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.
1 Apple Hill Drive
Natick, MA 01760-2098

Revision History

Contents

<div style="text-align: right;">

Setting Up an Optimization

</div>

2

Examining Results

3

Steps to Take After Running a Solver

4

Optimization App

5

Nonlinear algorithms and examples

6

Nonlinear Problem-Based

7

Multiobjective Algorithms and Examples

8

Linear Programming and Mixed-Integer Linear Programming

9

Quadratic Programming

11

Least Squares

12

Systems of Equations

13

14

<div style="text-align: right">

Parallel Computing for Optimization

</div>

15

<div style="text-align: right">

Argument and Options Reference

</div>

Acknowledgments

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

Solve linear, quadratic, conic, 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 (LP), mixed-integer linear programming (MILP), quadratic programming (QP), second-order cone programming (SOCP), nonlinear programming (NLP), constrained linear least squares, nonlinear least squares, and nonlinear equations.

You can define your optimization problem with functions and matrices or by specifying variable expressions that reflect the underlying mathematics. You can use automatic differentiation of objective and constraint functions for faster and more accurate solutions.

You can use the toolbox solvers to find optimal solutions to continuous and discrete problems, perform tradeoff analyses, and incorporate optimization methods into algorithms and applications. The toolbox lets you perform design optimization tasks, including parameter estimation, component selection, and parameter tuning. It enables you to find optimal solutions in applications such as portfolio optimization, energy management and trading, and production planning.

First Choose Problem-Based or Solver-Based Approach

Optimization Toolbox has two approaches to solving optimization problems or equations: problem-based and solver-based. Before you start to solve a problem, you must first choose the appropriate approach.

This table summarizes the main differences between the two approaches.

Approaches	Characteristics
"Problem-Based Optimization Setup"	**Easier to create and debug**
	Represents the objective and constraints symbolically
	Requires translation from problem form to matrix form, resulting in a [...] time
	Automatically calculates and uses gradients of objective and nonlinear [...] functions in many cases, but does not calculate Hessians; see "Automa[...] Differentiation" on page 16-473
	See the steps in "Problem-Based Optimization Workflow" on page 10-[...] "Problem-Based Workflow for Solving Equations" on page 10-4
	Basic linear example: "Mixed-Integer Linear Programming Basics: Pr[...] page 10-47 or the video Solve a Mixed-Integer Linear Programming P[...] Optimization Modeling
	Basic nonlinear example: "Solve a Constrained Nonlinear Problem, Pr[...] on page 1-5
	Basic equation-solving example: "Solve Nonlinear System of Equation[...] Based" on page 13-21
"Solver-Based Optimization Problem Setup"	**Harder to create and debug**
	Provides a visual interface; see **Optimize** Live Editor task
	Represents the objective and constraints as functions or matrices
	Does not require translation from problem form to matrix form, result[...] solution time
	Allows direct inclusion of gradient or Hessian, but does not calculate t[...] automatically
	Allows use of a Hessian multiply function or Jacobian multiply functio[...] memory in large problems
	See "Quadratic Minimization with Dense, Structured Hessian" on pag[...] "Jacobian Multiply Function with Linear Least Squares" on page 12-3[...]
	See the steps in "Solver-Based Optimization Problem Setup"
	Basic linear example: "Mixed-Integer Linear Programming Basics: Sol[...] page 9-54
	Basic nonlinear example: "Solve a Constrained Nonlinear Problem, So[...] page 1-11
	Basic equation-solving examples: "Examples" on page 16-0

See Also

More About
- "Problem-Based Optimization Setup"
- "Solver-Based Optimization Problem Setup"

Solve a Constrained Nonlinear Problem, Problem-Based

Typical Optimization Problem

This example shows how to solve a constrained nonlinear optimization problem using the problem-based approach. The example demonstrates the typical work flow: create an objective function, create constraints, solve the problem, and examine the results.

Note:

If your objective function or nonlinear constraints are not composed of elementary functions, you must convert the nonlinear functions to optimization expressions using `fcn2optimexpr.` See the last part of this example, Alternative Formulation Using fcn2optimexpr on page 1-0 , or "Convert Nonlinear Function to Optimization Expression" on page 7-8.

For the solver-based approach to this problem, see "Solve a Constrained Nonlinear Problem, Solver-Based" on page 1-11.

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*, meaning 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 subject to a nonlinear constraint.

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 because the function has a shallow minimum inside a deeply curved valley. The solution for this problem is not at the point `[1,1]` because that point does not satisfy the constraint.

This figure shows 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.

```
rosenbrock = @(x)100*(x(:,2) - x(:,1).^2).^2 + (1 - x(:,1)).^2; % Vectorized function

figure1 = figure('Position',[1 200 600 300]);
colormap('gray');
axis square;
R = 0:.002:1;
TH = 2*pi*(0:.002:1);
X = R'*cos(TH);
Y = R'*sin(TH);
Z = log(1 + rosenbrock([X(:),Y(:)]));
Z = reshape(Z,size(X));

% Create subplot
subplot1 = subplot(1,2,1,'Parent',figure1);
view([124 34]);
grid('on');
hold on;

% Create surface
```

```
surf(X,Y,Z,'Parent',subplot1,'LineStyle','none');

% Create contour
contour(X,Y,Z,'Parent',subplot1);

% Create subplot
subplot2 = subplot(1,2,2,'Parent',figure1);
view([234 34]);
grid('on');
hold on

% Create surface
surf(X,Y,Z,'Parent',subplot2,'LineStyle','none');

% Create contour
contour(X,Y,Z,'Parent',subplot2);

% Create textarrow
annotation(figure1,'textarrow',[0.4 0.31],...
    [0.055 0.16],...
    'String',{'Minimum at (0.7864,0.6177)'});

% Create arrow
annotation(figure1,'arrow',[0.59 0.62],...
    [0.065 0.34]);

title("Rosenbrock's Function: Two Views")

hold off
```
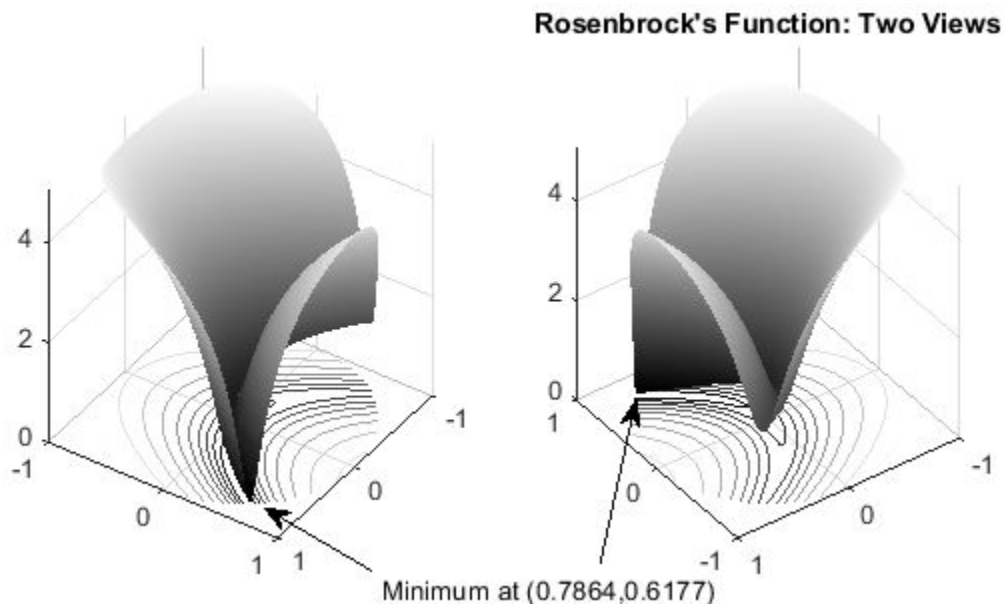

The `rosenbrock` function handle calculates Rosenbrock's function at any number of 2-D points at once. This "Vectorization" speeds the plotting of the function, and can be useful in other contexts for speeding evaluation of a function at multiple points.

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

Define Problem Using Optimization Variables

The problem-based approach to optimization uses optimization variables to define objective and constraints. There are two approaches for creating expressions using these variables:

- For polynomial or rational functions, write expressions directly in the variables.
- For other types of functions, convert functions to optimization expressions using `fcn2optimexpr`. See **Alternative Formulation Using `fcn2optimexpr`** at the end of this example.

For this problem, both the objective function and the nonlinear constraint are polynomials, so you can write the expressions directly in terms of optimization variables. Create a 2-D optimization variable named 'x'.

```
x = optimvar('x',1,2);
```

Create the objective function as a polynomial in the optimization variable.

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

Create an optimization problem named `prob` having `obj` as the objective function.

```
prob = optimproblem('Objective',obj);
```

Create the nonlinear constraint as a polynomial in the optimization variable.

```
nlcons = x(1)^2 + x(2)^2 <= 1;
```

Include the nonlinear constraint in the problem.

```
prob.Constraints.circlecons = nlcons;
```

Review the problem.

```
show(prob)
```

```
  OptimizationProblem :

    Solve for:
       x

    minimize :
       ((100 .* (x(2) - x(1).^2).^2) + (1 - x(1)).^2)


    subject to circlecons:
       (x(1).^2 + x(2).^2) <= 1
```

Solve Problem

To solve the optimization problem, call `solve`. The problem needs an initial point, which is a structure giving the initial value of the optimization variable. Create the initial point structure `x0` having an *x*-value of `[0 0]`.

```
x0.x = [0 0];
[sol,fval,exitflag,output] = solve(prob,x0)

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol = struct with fields:
    x: [0.7864 0.6177]


fval = 0.0457

exitflag =
    OptimalSolution


output = struct with fields:
           iterations: 24
            funcCount: 34
        constrviolation: 0
             stepsize: 6.9161e-06
            algorithm: 'interior-point'
        firstorderopt: 2.1625e-08
          cgiterations: 4
               message: '...'
          bestfeasible: [1x1 struct]
                solver: 'fmincon'
```

Examine Solution

The solution shows `exitflag = OptimalSolution`. This exit flag indicates that the solution is a local optimum. For information on trying to find a better solution, see "When the Solver Succeeds" on page 4-18.

The exit message indicates that the solution satisfies the constraints. You can check that the solution is indeed feasible in several ways.

• Check the reported infeasibility in the `constrviolation` field of the `output` structure.

```
infeas = output.constrviolation

infeas = 0
```

An infeasibility of 0 indicates that the solution is feasible.

• Compute the infeasibility at the solution.

```
infeas = infeasibility(nlcons,sol)

infeas = 0
```

Again, an infeasibility of 0 indicates that the solution is feasible.

- Compute the norm of `x` to ensure that it is less than or equal to 1.

```
nx = norm(sol.x)
```

```
nx = 1.0000
```

The `output` structure gives more information on the solution process, such as the number of iterations (24), the solver (`fmincon`), and the number of function evaluations (84). For more information on these statistics, see "Tolerances and Stopping Criteria" on page 2-68.

Alternative Formulation Using `fcn2optimexpr`

For more complex expressions, write function files for the objective or constraint functions, and convert them to optimization expressions using `fcn2optimexpr`. For example, the basis of the nonlinear constraint function is in the `disk.m` file:

```
type disk
```

```
function radsqr = disk(x)
```

```
radsqr = x(1)^2 + x(2)^2;
```

Convert this function file to an optimization expression.

```
radsqexpr = fcn2optimexpr(@disk,x);
```

Furthermore, you can also convert the `rosenbrock` function handle, which was defined at the beginning of the plotting routine, into an optimization expression.

```
rosenexpr = fcn2optimexpr(rosenbrock,x);
```

Create an optimization problem using these converted optimization expressions.

```
convprob = optimproblem('Objective',rosenexpr,'Constraints',radsqexpr <= 1);
```

View the new problem.

```
show(convprob)
```

```
  OptimizationProblem :

    Solve for:
       x

    minimize :
       anonymousFunction2(x)

       where:

         anonymousFunction2 = @(x)100*(x(:,2)-x(:,1).^2).^2+(1-x(:,1)).^2;


    subject to :
       disk(x) <= 1
```

Solve the new problem. The solution is essentially the same as before.

```
[sol,fval,exitflag,output] = solve(convprob,x0)
```

```
Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol = struct with fields:
    x: [0.7864 0.6177]


fval = 0.0457

exitflag =
    OptimalSolution


output = struct with fields:
         iterations: 24
          funcCount: 84
      constrviolation: 0
           stepsize: 6.9162e-06
          algorithm: 'interior-point'
       firstorderopt: 2.0234e-08
        cgiterations: 4
            message: '...'
         bestfeasible: [1x1 struct]
              solver: 'fmincon'
```

For the list of supported functions, see "Supported Operations on Optimization Variables and Expressions" on page 10-43.

See Also

More About

Solve a Constrained Nonlinear Problem, Solver-Based

In this section...
"Typical Optimization Problem" on page 1-11
"Problem Formulation: Rosenbrock's Function" on page 1-11
"Define and Solve Problem Using Optimize Live Editor Task" on page 1-12
"Define and Solve Problem at Command Line" on page 1-16
"Interpret Result" on page 1-19

Typical Optimization Problem

This example shows how to solve a constrained nonlinear problem using an Optimization Toolbox solver. The example demonstrates the typical workflow: create an objective function, create constraints, solve the problem, and examine the results.

This example provides two approaches to solving the problem. One uses the **Optimize** Live Editor task, a visual approach. The other uses the MATLAB® command line, a text-based approach. You can also solve this type of problem using the problem-based approach; see "Solve a Constrained Nonlinear Problem, Problem-Based" on page 1-5.

Problem Formulation: Rosenbrock's Function

The problem is to minimize Rosenbrock's function

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

over the unit disk, that is, 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 because the function has a shallow minimum inside a deeply curved valley. The solution for this problem is not at the point [1,1] because that point does not satisfy the constraint.

This figure shows 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 (0.7864,0.6177)
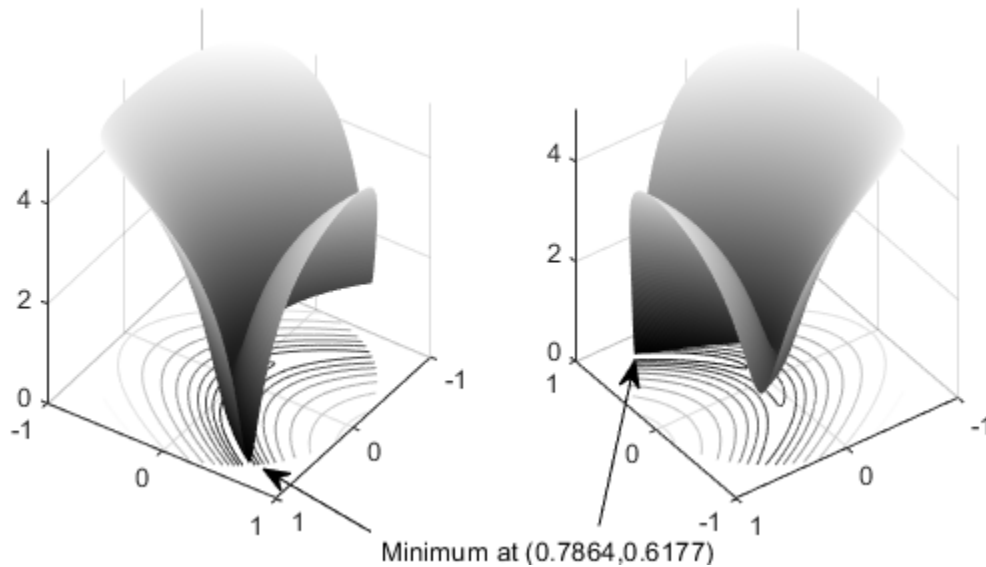
Rosenbrock's Function, Log-Scaled: Two Views

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

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 2-30.

Define and Solve Problem Using Optimize Live Editor Task

The **Optimize** Live Editor task lets you set up and solve the problem using a visual approach.

1 Create a new live script by clicking the **New Live Script** button on the **File** section of the **Home** tab.

2 Insert an **Optimize** Live Editor task. Click the **Insert** tab and then, in the **Code** section, select **Task > Optimize**.

3   In the **Specify problem type** section of the task, select **Objective > Nonlinear** and **Constraints > Nonlinear**. The task selects the solver `fmincon - Constrained nonlinear minimization`.

4   Include Rosenbrock's function as the objective function. In the **Select problem data** section of the task, select **Objective function > Local function** and then click the **New...** button. A new local function appears in a section below the task.

```
function f = objectiveFcn(optimInput)
% Example:
% Minimize Rosenbrock's function
% f = 100*(y - x^2)^2 + (1 - x)^2

% Edit the lines below with your calculation
x = optimInput(1);
y = optimInput(2);
f = 100*(y - x^2)^2 + (1 - x)^2;
end
```

This function implements Rosenbrock's function.

5   In the **Select problem data** section of the task, select **Objective function > objectiveFcn**.

6   Place the initial point `x0 = [0;0]` into the MATLAB workspace. Insert a new section above the **Optimize** task by clicking the task, then clicking the **Section Break** button on the **Insert** tab. In the new section above the task, enter the following code for the initial point.

```
x0 = [0;0];
```

7   Run the section by pressing **Ctrl+Enter**. This action places `x0` into the workspace.

8   In the **Select problem data** section of the task, select **Initial point (x0) > x0**.

9 In the **Select problem data** section, select **Constraints > Nonlinear > Local function** and then click the **New...** button. A new local function appears below the previous local function.

10 Edit the new local function as follows.

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

11 In the **Select problem data** section, select `unitdisk` as the constraint function.

12 To monitor the solver progress, in the **Display progress** section of the task, select **Text display > Each iteration**. Also, select **Objective value and feasibility** for the plot.

13 To run the solver, click the options button **:** at the top right of the task window, and select **Run Section**. The plot appears in a separate figure window and in the output area.

Best Function Value: 0.0456748
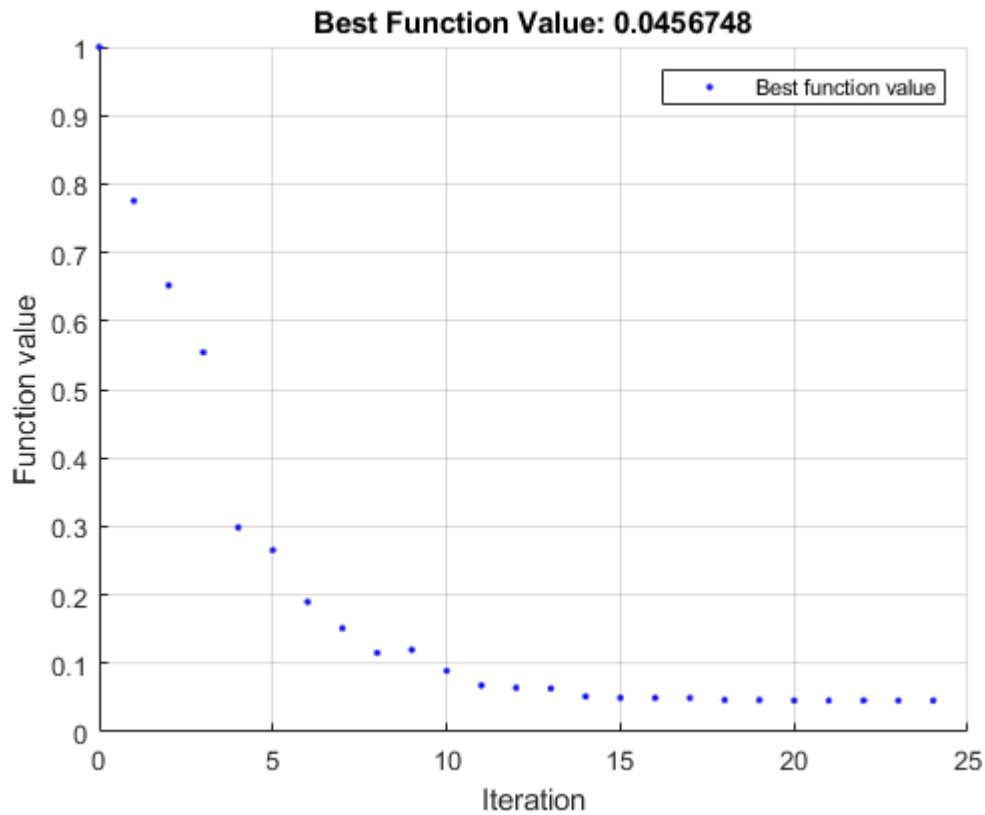
The output area shows a table of iterations, discussed in "Interpret Result" on page 1-19.

14 To find the solution, look at the top of the task.

Optimize

solution , objectiveValue = Minimize **objectiveFcn** using **fmincon** solver

The solver places the variables `solution` and `objectiveValue` in the workspace. View their values by inserting a new section break below the task and entering these lines.

```
11      disp(solution)
12      disp(objectiveValue)
```

15 Run the section by pressing **Ctrl+Enter**.

```
disp(solution)

    0.7864
    0.6177

disp(objectiveValue)|

    0.0457
```

To understand the `fmincon` process for obtaining the result, see "Interpret Result" on page 1-19.

16 To display the code that **Optimize** generates to solve the problem, click the options button ⋮ at the top right of the task window, and select **Controls and Code**.

```
⋮
● Controls and Code
  Controls Only
  Code Only
```
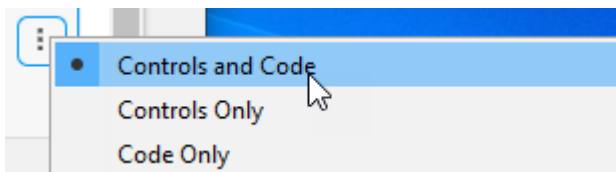
At the bottom of the task, the following code appears.

```
% Set nondefault solver options
options = optimoptions('fmincon','Display','iter','PlotFcn',...
    'optimplotfvalconstr');

% Solve
[solution,objectiveValue] = fmincon(@objectiveFcn,x0,[],[],[],[],[],[],...
    @unitdisk,options);
```

This code is the code you use to solve the problem at the command line, as described next.

Define and Solve Problem at Command Line

The first step in solving an optimization problem at the command line is to choose a solver. Consult the "Optimization Decision Table" on page 2-4. For a problem with a nonlinear objective function and a nonlinear constraint, generally you use the `fmincon` solver.

Consult the `fmincon` function reference page. The solver syntax is as follows.

```
[x,fval] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

The `fun` and `nonlcon` inputs represent the objective function and nonlinear constraint functions, respectively.

Express your problem as follows:

1 Define the objective function in the MATLAB language, as a function file or anonymous function. This example uses a function file.

2 Define the constraints as a separate file or anonymous function.

A function file is a text file that contains MATLAB commands and has the extension `.m`. Create a function file in any text editor, or use the built-in MATLAB Editor as in this example.

1 At the command line, enter:

```
edit rosenbrock
```

2 In the MATLAB Editor, enter:

```
%% ROSENBROCK(x) expects a two-column matrix and returns a column vector
% The output is the Rosenbrock function, which has a minimum at
% (1,1) of value 0, and is strictly positive everywhere else.

function f = rosenbrock(x)

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

Note `rosenbrock` is a vectorized function that can compute values for several points at once. See "Vectorization". A vectorized function is best for plotting. For a nonvectorized version, enter:

```
%% ROSENBROCK1(x) expects a two-element vector and returns a scalar
% The output is the Rosenbrock function, which has a minimum at
% (1,1) of value 0, and is strictly positive everywhere else.

function f = rosenbrock1(x)

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

---

3 Save the file with the name `rosenbrock.m`.

Constraint functions have the form $c(x) \leq 0$ or $ceq(x) = 0$. The constraint $x_1^2 + x_2^2 \leq 1$ is not in the form that the solver handles. To have the correct syntax, reformulate the constraint as $x_1^2 + x_2^2 - 1 \leq 0$.

The syntax for nonlinear constraints returns both equality and inequality constraints. This example includes 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`.

Now that you have defined the objective and constraint functions, create the other `fmincon` inputs.

1 Create options for `fmincon` to use the `'optimplotfvalconstr'` plot function and to return iterative display.

```
options = optimoptions('fmincon',...
    'PlotFcn','optimplotfvalconstr',...
    'Display','iter');
```

2 Create the initial point.

```
x0 = [0 0];
```

3 Create empty entries for the constraints that this example does not use.

```
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
```

Solve the problem by calling `fmincon`.

```
[x,fval] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

```
                                       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     6.440e-05    3.974e-05
   22      78   4.568281e-02   0.000e+00     8.000e-06    1.084e-07
   23      81   4.567641e-02   0.000e+00     1.601e-06    2.793e-05
   24      84   4.567482e-02   0.000e+00     2.023e-08    6.916e-06
```

```
Local minimum found that satisfies the constraints.

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

x =

    0.7864    0.6177


fval =

    0.0457
```
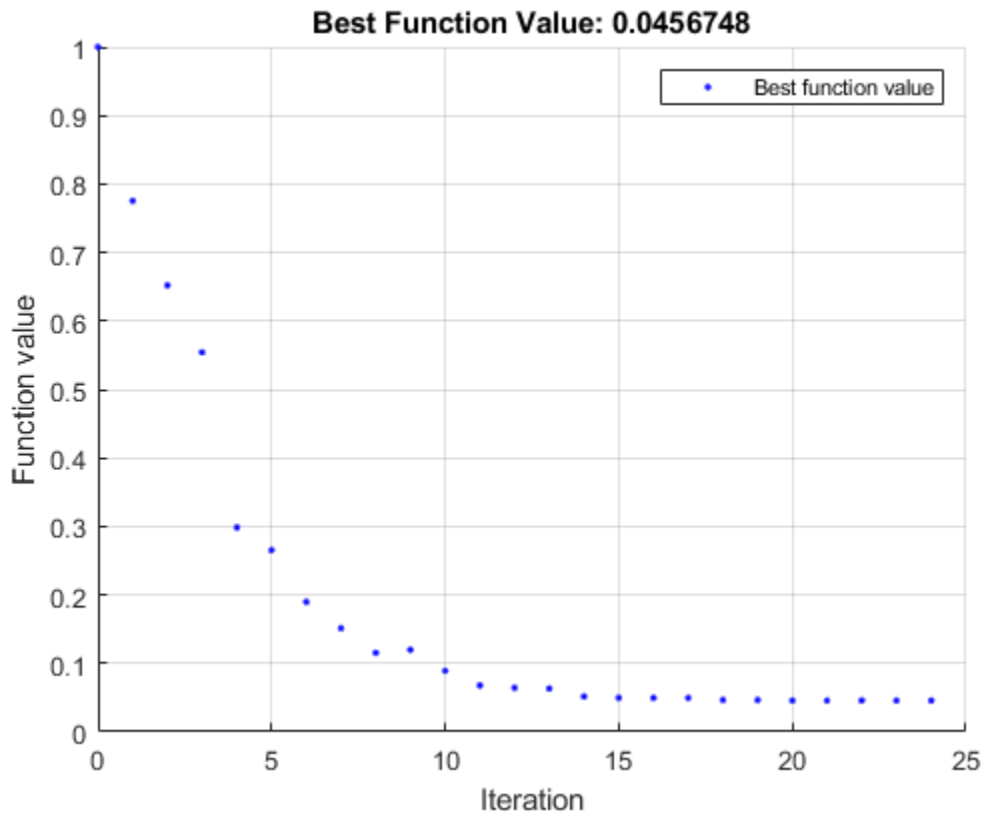
The exit message tells you that the search for a constrained optimum ended because the derivative of the objective function is nearly 0 in directions allowed by the constraint, and that the constraint is satisfied to the required accuracy. Several phrases in the message contain links to more information about the terms used in the message. For more details about these links, see "Enhanced Exit Messages" on page 3-4.

Interpret Result

The iteration table in both the Live Editor task output area and the MATLAB Command Window shows how MATLAB searched for the minimum value of Rosenbrock's function in the unit disk. Your table can differ, depending on toolbox version and computing platform. The following description applies to the table shown in this example.

- 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, `4.567482e-2`, is the minimum reported in the **Optimize** run, 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. Because 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 3-14.

See Also
Optimize | `fmincon`

More About

- "Solve a Constrained Nonlinear Problem, Problem-Based" on page 1-5
- "First Choose Problem-Based or Solver-Based Approach" on page 1-3
- "Get Started with Optimize Live Editor Task" on page 1-34
- "Use Optimize Live Editor Task Effectively" on page 1-39
- "Solver-Based Optimization Problem Setup"
- How to Use the Optimize Live Editor Task

Set Up a Linear Program, Solver-Based

In this section...
"Convert a Problem to Solver Form" on page 1-21
"Model Description" on page 1-21
"Solution Method" on page 1-22
"Bibliography" on page 1-27

Convert a Problem to Solver Form

This example shows how to convert a problem from mathematical form into Optimization Toolbox solver syntax using the solver-based approach. 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.

- Mathematical Modeling with Optimization, Part 1 shows the problem in pictorial form. It shows how to generate the mathematical expressions of "Model Description" on page 1-21 from the picture.
- Optimization Modeling, Part 2: Converting to Solver Form 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, Part 2: Converting to Solver Form. 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 1-23.

Model Description

The video Mathematical Modeling with Optimization, Part 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 Mathematical Modeling with Optimization, Part 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 ≤ P1 ≤ 6250
I1 ≤ 192,000
C ≤ 62,000
I1 - HE1 ≤ 132,000
I1 = LE1 + HE1 + C
1359.8 I1 = 1267.8 HE1 + 1251.4 LE1 + 192 C + 3413 P1
3000 ≤ P2 ≤ 9000
I2 ≤ 244,000
LE2 ≤ 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 ≥ 24,550
EP + PP ≥ 12,000
MPS ≥ 271,536
LPS ≥ 100,623
All variables are positive.

Solution Method

To solve the optimization problem, take the following steps.

1. "Choose a Solver" on page 1-22
2. "Combine Variables Into One Vector" on page 1-23
3. "Write Bound Constraints" on page 1-24
4. "Write Linear Inequality Constraints" on page 1-25
5. "Write Linear Equality Constraints" on page 1-25
6. "Write the Objective" on page 1-26
7. "Solve the Problem with linprog" on page 1-26
8. "Examine the Solution" on page 1-27

The steps are also shown in the video Optimization Modeling, Part 2: Converting to Solver Form.

Choose a Solver

To find the appropriate solver for this problem, consult the "Optimization Decision Table" on page 2-4. 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 2-12 or the `linprog` function reference page, the `linprog` solver solves problems of the form

$$\min_{x} f^T x \text{ such that } \begin{cases} A \cdot x \le b, \\ Aeq \cdot x = beq, \\ lb \le x \le ub. \end{cases} \tag{1-1}$$

- $f^T x$ means a row vector of constants f multiplying a column vector of variables x. In other words,

  $f^T x = f(1)x(1) + f(2)x(2) + \dots + f(n)x(n),$

  where n is the length of f.
- $A\,x \le b$ 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 2-35.
- $Aeq\,x = beq$ 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 2-36.
- $lb \le x \le ub$ 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 2-34.

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 1-21. 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 names for the variables.

```
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:

Workspace	
Name ▲	Value
BF1	8
BF2	9
C	7
EP	16
HE1	3
HE2	4
HPS	10
I1	1
I2	2
LE1	5
LE2	6
LPS	12
MPS	11
N	16
P1	13
P2	14
PP	15
v	16
{} variables	1x16 cell

These named variables represent index numbers for the components of x. You do not have to create named variables. The video Optimization Modeling, Part 2: Converting to Solver Form 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 1-21. 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 ≤ 6250
P2 ≤ 9000
I1 ≤ 192,000
I2 ≤ 244,000
C ≤ 62,000

LE2 ≤ 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 1-21:

```
I1 - HE1 ≤ 132,000
EP + PP ≥ 12,000
P1 + P2 + PP ≥ 24,550.
```

In order to have the equations in the form $A\,x \le b$, 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 ≤ 132,000
-EP - PP ≤ -12,000
-P1 - P2 - PP ≤ -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 1-21:

```
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

$$f^T x = 0.002614 \text{ HPS} + 0.0239 \text{ PP} + 0.009825 \text{ EP}.$$

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:

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

The result:

```
Optimal solution found.
    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+03
```

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, Part 2: Converting to Solver Form 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.

See Also

More About

- "Set Up a Linear Program, Problem-Based" on page 1-28
- "Solver-Based Optimization Problem Setup"

Set Up a Linear Program, Problem-Based

In this section...
"Convert Problem to Solver Form" on page 1-28
"Model Description" on page 1-28
"First Solution Method: Create Optimization Variable for Each Problem Variable" on page 1-29
"Second Solution Method: Create One Optimization Variable and Indices" on page 1-31
"Bibliography" on page 1-33

Convert Problem to Solver Form

This example shows how to convert a linear problem from mathematical form into Optimization Toolbox solver syntax using the problem-based approach.

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

- Mathematical Modeling with Optimization, Part 1 presents the problem in pictorial form, showing how to generate the mathematical expressions of the "Model Description" on page 1-28.

- Optimization Modeling, Part 2: Problem-Based Solution of a Mathematical Model 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.

This example, which closely follows the Part 2 video, focuses on transforming the problem to solver syntax.

Model Description

The Part 1 video suggests the following approach for converting a problem into mathematical form:

1. Get an overall idea of the problem.
2. Identify the goal (maximizing or minimizing something).
3. Identify (name) the variables.
4. Identify the 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 Part 1 video.

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 ≤ 192,000
C ≤ 62,000
I1 - HE1 ≤ 132,000
I1 = LE1 + HE1 + C
1359.8 I1 = 1267.8 HE1 + 1251.4 LE1 + 192 C + 3413 P1
3000 ≤ P2 ≤ 9000
I2 ≤ 244,000
LE2 ≤ 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 ≥ 24,550
EP + PP ≥ 12,000
MPS ≥ 271,536
LPS ≥ 100,623
```
All variables are positive.

First Solution Method: Create Optimization Variable for Each Problem Variable

The first solution method involves creating an optimization variable for each problem variable. As you create the variables, include their bounds.

```matlab
P1 = optimvar('P1','LowerBound',2500,'UpperBound',6250);
P2 = optimvar('P2','LowerBound',3000,'UpperBound',9000);
I1 = optimvar('I1','LowerBound',0,'UpperBound',192000);
I2 = optimvar('I2','LowerBound',0,'UpperBound',244000);
C = optimvar('C','LowerBound',0,'UpperBound',62000);
LE1 = optimvar('LE1','LowerBound',0);
LE2 = optimvar('LE2','LowerBound',0,'UpperBound',142000);
HE1 = optimvar('HE1','LowerBound',0);
HE2 = optimvar('HE2','LowerBound',0);
HPS = optimvar('HPS','LowerBound',0);
MPS = optimvar('MPS','LowerBound',271536);
LPS = optimvar('LPS','LowerBound',100623);
BF1 = optimvar('BF1','LowerBound',0);
BF2 = optimvar('BF2','LowerBound',0);
EP = optimvar('EP','LowerBound',0);
PP = optimvar('PP','LowerBound',0);
```

Create Problem and Objective

Create an optimization problem container. Include the objective function in the problem.

```matlab
linprob = optimproblem('Objective',0.002614*HPS + 0.0239*PP + 0.009825*EP);
```

Create and Include Linear Constraints

The problem expressions contain three linear inequalities:

I1 - HE1 ≤ 132,000   (1-2)

EP + PP ≥ 12,000
P1 + P2 + PP ≥ 24,550

Create these inequality constraints and include them in the problem.

```
linprob.Constraints.cons1 = I1 - HE1 <= 132000;
linprob.Constraints.cons2 = EP + PP >= 12000;
linprob.Constraints.cons3 = P1 + P2 + PP >= 24550;
```

The problem has eight linear equalities:

I2 = LE2 + HE2   (1-3)
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.

Include these constraints as well.

```
linprob.Constraints.econs1 = LE2 + HE2 == I2;
linprob.Constraints.econs2 = LE1 + LE2 + BF2 == LPS;
linprob.Constraints.econs3 = I1 + I2 + BF1 == HPS;
linprob.Constraints.econs4 = C + MPS + LPS == HPS;
linprob.Constraints.econs5 = LE1 + HE1 + C == I1;
linprob.Constraints.econs6 = HE1 + HE2 + BF1 == BF2 + MPS;
linprob.Constraints.econs7 = 1267.8*HE1 + 1251.4*LE1 + 192*C + 3413*P1 == 1359.8*I1;
linprob.Constraints.econs8 = 1267.8*HE2 + 1251.4*LE2 + 3413*P2 == 1359.8*I2;
```

Solve Problem

The problem formulation is complete. Solve the problem using `solve`.

```
linsol = solve(linprob);
```

```
Optimal solution found.
```

Examine Solution

Evaluate the objective function. (You can also for this value when you first call `solve`.)

```
evaluate(linprob.Objective,linsol)
```

```
ans =

   1.2703e+03
```

The lowest-cost method of operating the plant costs $1,207.30.

Examine the solution variable values.

```
tbl = struct2table(linsol)
```

```
tbl =

  1×16 table
```

BF1	BF2	C	EP	HE1	HE2	HPS	I1	
0	0	8169.7	760.71	1.2816e+05	1.4338e+05	3.8033e+05	1.3633e+05	2.4

This table is too wide to view its contents easily. Stack the variables to arrange them vertically.

```
vars = {'P1','P2','I1','I2','C','LE1','LE2','HE1','HE2',...
'HPS','MPS','LPS','BF1','BF2','EP','PP'};
outputvars = stack(tbl,vars,'NewDataVariableName','Amt','IndexVariableName','Var')
```

```
outputvars =

  16×2 table

    Var        Amt
    ___     _____

    P1            6250
    P2          7060.7
    I1      1.3633e+05
    I2        2.44e+05
    C           8169.7
    LE1              0
    LE2     1.0062e+05
    HE1     1.2816e+05
    HE2     1.4338e+05
    HPS     3.8033e+05
    MPS     2.7154e+05
    LPS     1.0062e+05
    BF1              0
    BF2              0
    EP          760.71
    PP           11239
```

- `BF1`, `BF2`, and `LE1` are `0`, their lower bounds.
- `I2` is `244,000`, its upper bound.
- The nonzero components of the objective function (cost) are

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

The Part 2 video interprets these characteristics in terms of the original problem.

Second Solution Method: Create One Optimization Variable and Indices

Alternatively, you can solve the problem using just one optimization variable that has indices with the names of the problem variables. This method enables you to give a lower bound of zero to all problem variables at once.

```
vars = {'P1','P2','I1','I2','C','LE1','LE2','HE1','HE2',...
    'HPS','MPS','LPS','BF1','BF2','EP','PP'};
x = optimvar('x',vars,'LowerBound',0);
```

Set Variable Bounds

Include the bounds on the variables using dot notation.

```
x('P1').LowerBound = 2500;
x('P2').LowerBound = 3000;
x('MPS').LowerBound = 271536;
x('LPS').LowerBound = 100623;
x('P1').UpperBound = 6250;
x('P2').UpperBound = 9000;
x('I1').UpperBound = 192000;
x('I2').UpperBound = 244000;
x('C').UpperBound = 62000;
x('LE2').UpperBound = 142000;
```

Create Problem, Linear Constraints, and Solution

The remainder of the problem setup is similar to the setup using separate variables. The difference is that, instead of addressing a variable by its name, such as P1, you address it using its index, x('P1').

Create the problem object, include the linear constraints, and solve the problem.

```
linprob = optimproblem('Objective',0.002614*x('HPS') + 0.0239*x('PP') + 0.009825*x('EP'));

linprob.Constraints.cons1 = x('I1') - x('HE1') <= 132000;
linprob.Constraints.cons2 = x('EP') + x('PP') >= 12000;
linprob.Constraints.cons3 = x('P1') + x('P2') + x('PP') >= 24550;

linprob.Constraints.econs1 = x('LE2') + x('HE2') == x('I2');
linprob.Constraints.econs2 = x('LE1') + x('LE2') + x('BF2') == x('LPS');
linprob.Constraints.econs3 = x('I1') + x('I2') + x('BF1') == x('HPS');
linprob.Constraints.econs4 = x('C') + x('MPS') + x('LPS') == x('HPS');
linprob.Constraints.econs5 = x('LE1') + x('HE1') + x('C') == x('I1');
linprob.Constraints.econs6 = x('HE1') + x('HE2') + x('BF1') == x('BF2') + x('MPS');
linprob.Constraints.econs7 = 1267.8*x('HE1') + 1251.4*x('LE1') + 192*x('C') + 3413*x('P1') == 13!
linprob.Constraints.econs8 = 1267.8*x('HE2') + 1251.4*x('LE2') + 3413*x('P2') == 1359.8*x('I2');

[linsol,fval] = solve(linprob);
```

```
Optimal solution found.
```

Examine Indexed Solution

Examine the solution as a vertical table.

```
tbl = table(vars',linsol.x')
```

```
tbl =

  16×2 table

    Var1         Var2
    _____     _____

    'P1'            6250
    'P2'          7060.7
    'I1'       1.3633e+05
```

```
'I2'        2.44e+05
'C'          8169.7
'LE1'             0
'LE2'     1.0062e+05
'HE1'     1.2816e+05
'HE2'     1.4338e+05
'HPS'     3.8033e+05
'MPS'     2.7154e+05
'LPS'     1.0062e+05
'BF1'             0
'BF2'             0
'EP'         760.71
'PP'          11239
```

Bibliography

[1] Edgar, Thomas F., and David M. Himmelblau. *Optimization of Chemical Processes.* New York: McGraw-Hill, 1987.

See Also

More About

- "Set Up a Linear Program, Solver-Based" on page 1-21
- "Problem-Based Optimization Setup"

Get Started with Optimize Live Editor Task

This example script helps you to use the Optimize Live Editor task for optimization or equation solving. Modify the script for your own problem.

The script solves a nonlinear optimization problem with nonlinear constraints.

Include Parameters or Data

Typically, you have data or values to pass to the solver. Place those values in the input section (where you see x0) and run the section by choosing **Section > Run Section** or pressing `Control+Enter`.

Set the initial point x0 and scale a for the optimization.

```
x0 = [2;1];
a = 100;
```

Place the x0 value and any other problem data into the workspace by running this section before proceeding.

Optimize Live Editor Task

This task has objective and nonlinear constraint functions included. To change these functions, edit the function listings below the task.
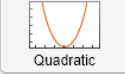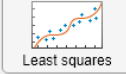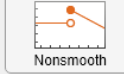
To change the constraints, select appropriate constraint types and enter values in the input boxes. You might need to enter values in the section containing x0 above, and run the section to put values in the workspace.

Run the task by clicking the striped bar to the left, or by choosing **Run** or **Section > Run Section**, or by pressing `Control+Enter`.

▼ Specify problem type

Objective

| Linear | Quadratic | Least squares | Nonlinear | Nonsmooth |

Examples: $f(x, y) = x/y$, $f(x) = \cos(x)$, $f(x) = \log(x)$, $f(x) = e^x$, $f(x) = x^3$, Solve $F(x) = 0, ...$

| Unconstrained | Lower bounds | Upper bounds | Linear inequality |

Constraints

| Linear equality | Second-order cone | Nonlinear | Integer |

Examples: $\cos(x) \leq 0$, $x^2 = 0$

Solver | fmincon - Constrained nonlinear minimization (recommended) ▼ | ❓

▼ Select problem data

Objective function | Local function ▼ | objectiveFcn ▼ | New... | ❓

▼ Function inputs

Optimization input | x ▼

Fixed input: a | a ▼

Initial point (x0) | x0 ▼

Constraints | Nonlinear | Local function ▼ | constraintFcn ▼ | New... | ❓

▷ Specify solver options

▼ Display progress

Text display | Final output ▼

Plot ☐ Current point   ☐ Evaluation count   ☑ Objective value and feasibility   ☐ Objective value

☐ Max constraint violation   ☐ Step size   ☐ Optimality measure
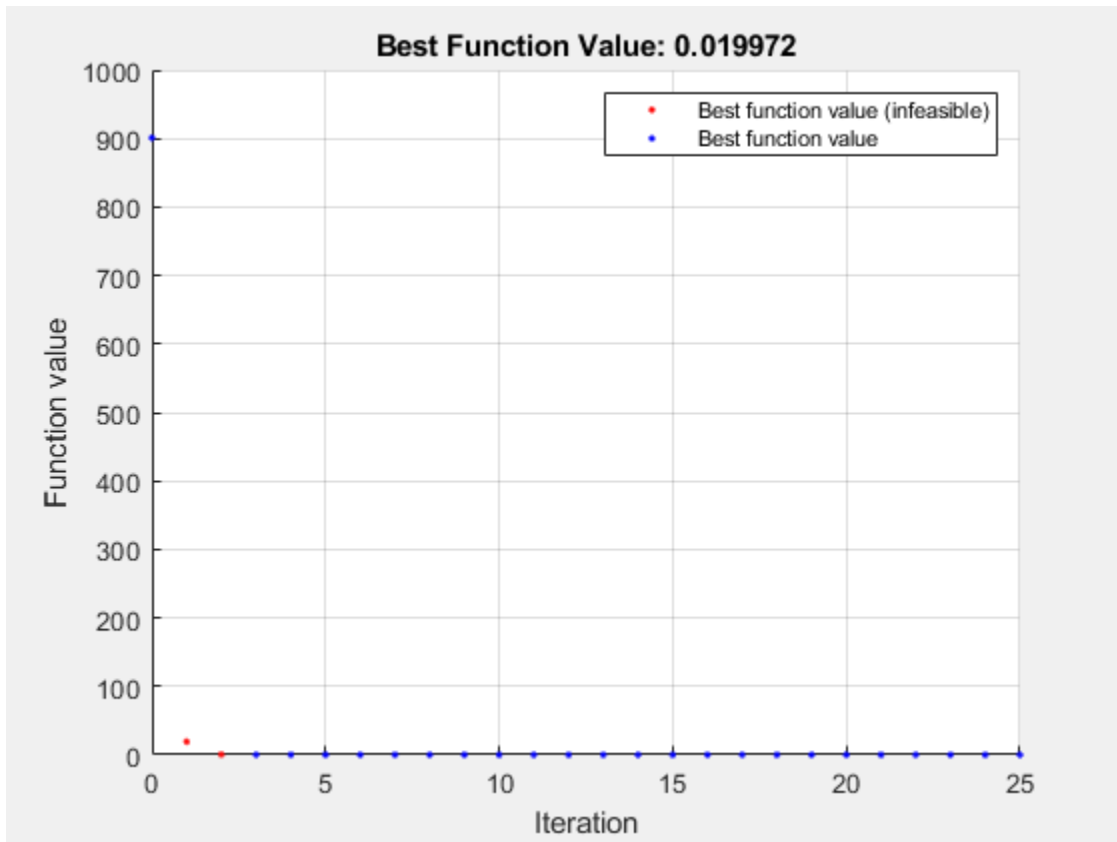
Local minimum found that satisfies the constraints.

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

<stopping criteria details>

Results

`Optimize` saves the solution to the workspace variable `solution`, and saves the objective function value at the solution to the workspace variable `objectiveValue`. You can see and modify these variable names at the top of the `Optimize` task.

View these variables.

`solution`

solution = *2×1*

    1.1413
    1.3029

`objectiveValue`

objectiveValue = 0.0200

View the nonlinear constraint function values at the solution.

```
[ccons,ceqcons] = constraintFcn(solution)

ccons = 1×2

   -2.0000   -0.0000


ceqcons =

     []
```

Helper Functions — Local Functions

The following code creates the objective function. Modify this code for your problem.

```
function f = objectiveFcn(x,a)
f = a*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
end
```

The following code creates the constraint function. Modify this code for your problem.

```
function [c,ceq] = constraintFcn(x)
c(1) = x(1)^2 + x(2)^2 - 5;
c(2) = 3 - x(1)^2 - x(2)^2;
ceq = [];  % No equality constraints
end
```

See Also
Optimize

More About
- "Use Optimize Live Editor Task Effectively" on page 1-39
- "Add Interactive Tasks to a Live Script"
- How to Use the Optimize Live Editor Task

Use Optimize Live Editor Task Effectively

In this section...
"Organize the Task Effectively" on page 1-39
"Place Optimization Variables in One Vector and Data in Other Variables" on page 1-40
"Specify Problem Type to Obtain Recommended Solver" on page 1-41
"Ways to Run the Task" on page 1-41
"View Solver Progress" on page 1-43
"View Equivalent Code" on page 1-43

Organize the Task Effectively

Place the **Optimize** Live Editor task in a live script with a section above and two or more sections below the task. To open the **Optimize** task in the Live Editor, click the **Insert** tab and then select **Task > Optimize**. Use the **Section Break** button on the **Insert** tab to insert a new section.
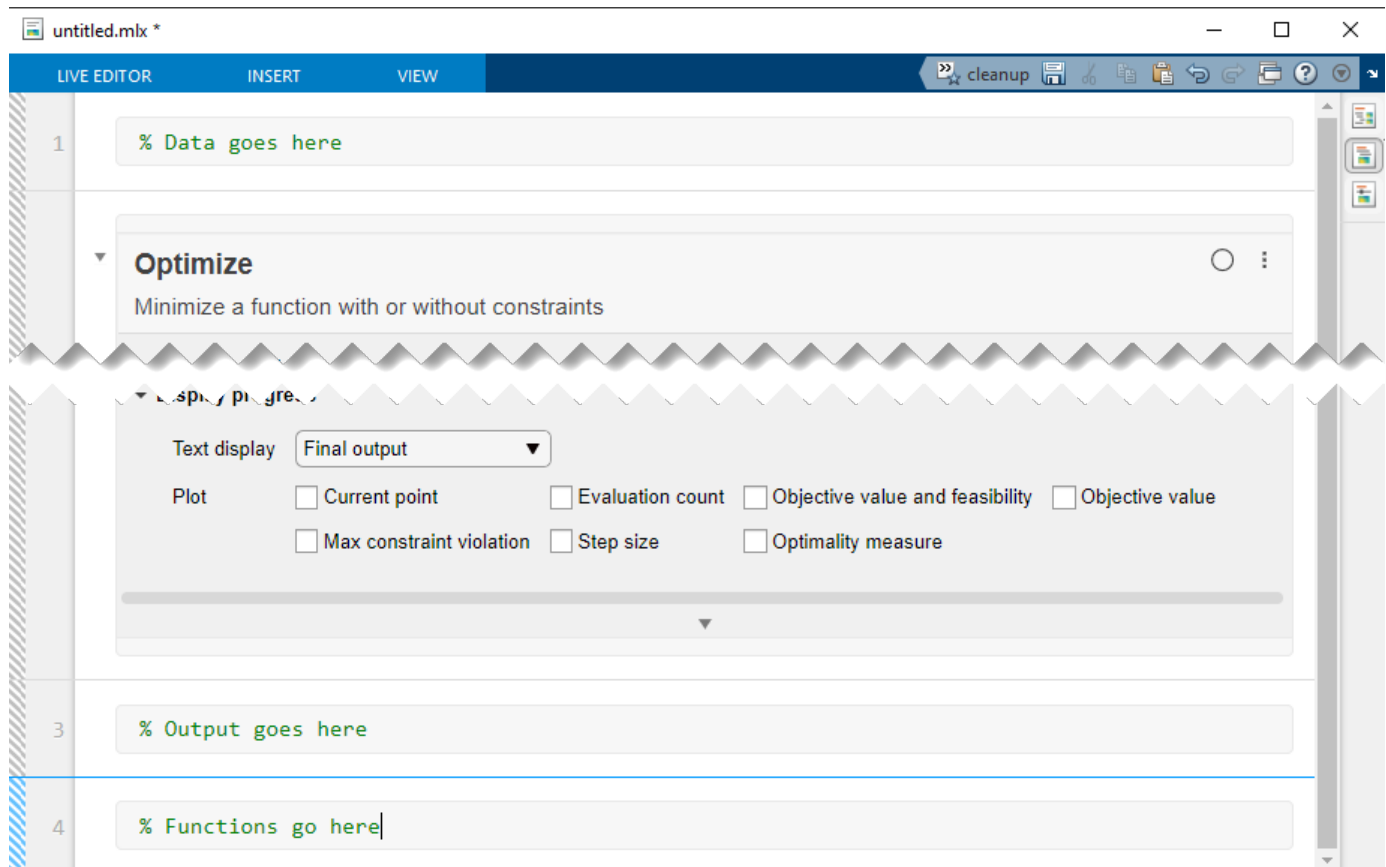
By default, the **Output on right** button is selected to the right of the task window.

Output on right
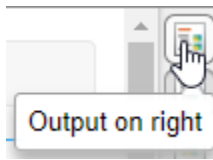
This selection places the output to the right of the task. To place the output below the task, select the **Output inline** button.

Output inline
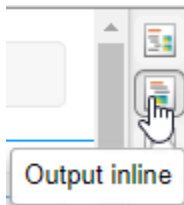
- Above the task, include a section for the data that you need for the optimization. For example, the initial point x0, any constraint matrices such as Aeq or beq, and extra parameters for objective or nonlinear constraint functions belong in the section above the task. The data must be included in a section above the task so that you can run the entire script successfully, for example, after saving and reloading it. The data loads into the workspace before the script needs to access it.
- Place outputs of the task in a section below the task. For example, display the solution and objectiveValue outputs in this section, after the task writes them to the workspace. You can include multiple sections below the task to view and work with the results of the task.
- The final section contains any local functions for the problem. Local functions must be included at the end of the live script. However, if you have functions that you access from more than one script, including them as separate files on the MATLAB path can be more convenient.

Place Optimization Variables in One Vector and Data in Other Variables

Optimize is a front end for solver-based optimization and equation solving. As such, it requires all variables to be placed in one vector, as documented in "Writing Scalar Objective Functions" on page 2-17. For example, suppose that your objective function is

$$f(x, y, z, w) = \left(x^2 + y^4\right)\exp(-(z/(1 + x^2)))w\exp(-z).$$

In this example, the variables x and z are the optimization variables, and the variables y and w are fixed data. You can represent your function in a section below the **Optimize** task as follows.

```
function f = myfun(vars,y,w)
x = vars(1);
z = vars(2);
f = (x^2 + y^4)*exp(-z/(1 + x^2))*w*exp(-z);
end
```

Define the values of the variables y and w in a section above the task.

```
y = log(pi);
w = 2/3;
```

Run the section above the task by pressing **Ctrl+Enter** to put y and w into the workspace. Then select the appropriate inputs in the **Select problem data** section of the task.

Specify Problem Type to Obtain Recommended Solver

The **Specify problem type** section of the task provides buttons for choosing the objective function type and the constraint types. After you select these items, **Optimize** reduces the number of available solvers and shows one solver as recommended. For example, for a problem with a least-squares objective and upper and lower bounds, **Optimize** shows that the `lsqnonlin` solver is recommended.

To use a solver that is not available with the current selections, deselect all of the problem type buttons by clicking each selected button.

Ways to Run the Task

You can run the **Optimize** Live Editor task in various ways:

- Click the options button ⋮ at the top right of the task window, and select **Run Section**.
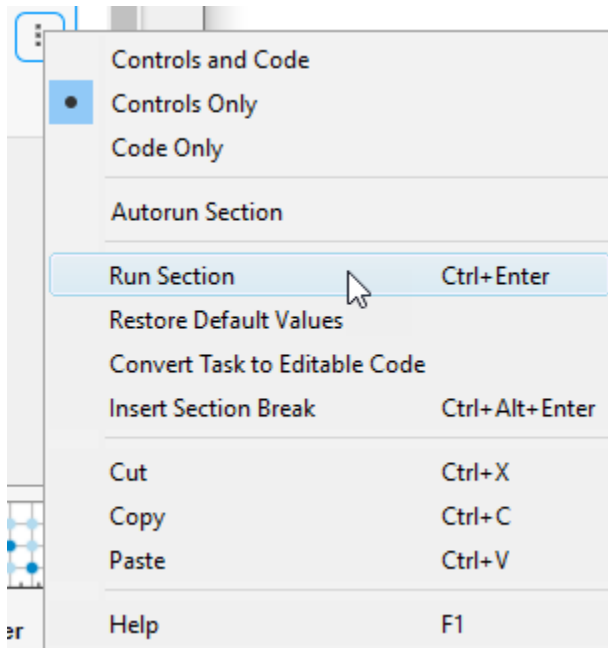
- Click in the task and then press **Ctrl+Enter**.
- Set the task to autorun after any change by selecting the **autorun** button (next to the options button at the top right of the task window). If your task is time consuming, do not choose this setting.
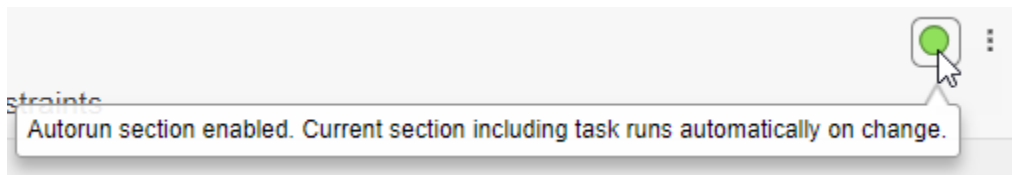
- Run the section containing the task by clicking the striped bar to the left of the task.
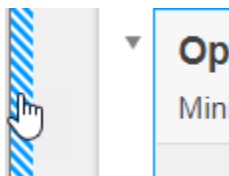
- Run the entire live script from the **Live Editor** tab by clicking the **Run** button, or by pressing **F5**.
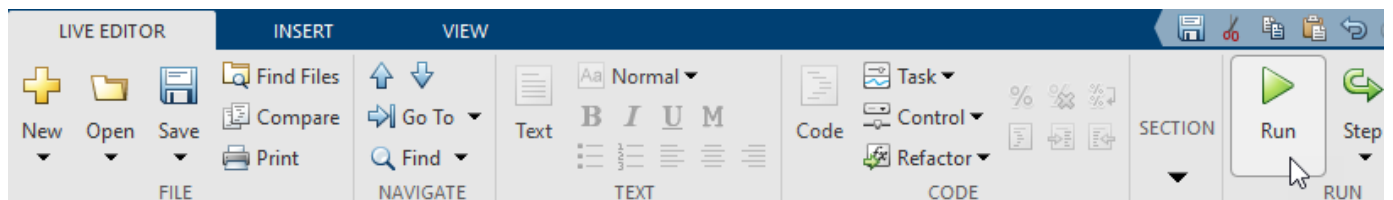
View Solver Progress

The Live Editor task enables you to monitor the solver progress easily. To ensure that the solver is performing properly, view at least the objective function value plot. Also, by using a plot function you can stop the solver without losing any data.

View Equivalent Code

Optimize internally creates code to match the visual selections. You can view the code by clicking the options button **⋮** and selecting **Controls and Code** or **Code Only**.
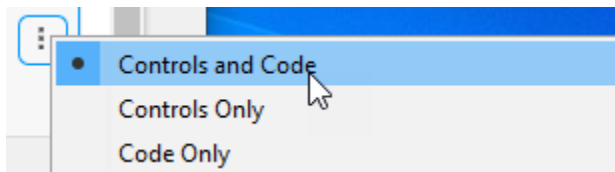
The code appears below the task.

```
% Set nondefault solver options
options2 = optimoptions('lsqnonlin','PlotFcn','optimplotx');

% Solve
[solution,objectiveValue] = lsqnonlin(@lsqnonlinObjFcn,x0,lb,ub,options2);
```

You can select and copy this code to modify it for use in other contexts.

To convert the task from a visual interface to usable code, choose **Convert Task to Editable Code**. This choice removes the visual **Optimize** interface and allows you to proceed using code.
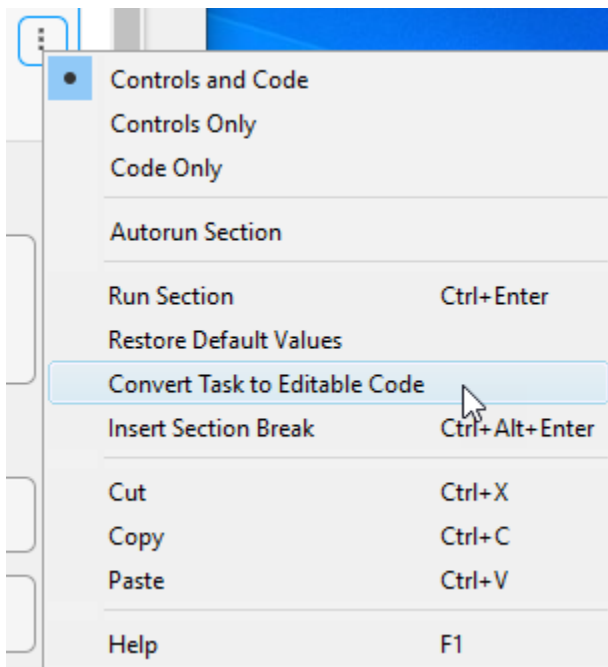
See Also
Optimize

More About

- "Get Started with Optimization Toolbox"
- "Get Started with Optimize Live Editor Task" on page 1-34
- "Add Interactive Tasks to a Live Script"
- How to Use the Optimize Live Editor Task

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 process 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 one or more of these forms:

- Equality constraints, $G_i(x) = 0$ ($i = 1,...,m_e$)
- Inequality constraints, $G_i(x) \leq 0$ ($i = m_e + 1,...,m$)
- Parameter bounds, x_l, x_u, where $x_l \leq x \leq x_u$, some x_l can be $-\infty$, and some x_u can be ∞

A General Problem (GP) description is stated as

$$\min_x f(x),\tag{2-1}$$

subject to

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

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 solution is usually achieved by the solution of an LP, QP, or unconstrained subproblem.

All optimization takes place in real numbers. However, unconstrained least-squares problems and equation solving can be formulated and solved using complex analytic functions. See "Complex Numbers in Optimization Toolbox Solvers" on page 2-14.

Optimization Toolbox Solvers

Optimization Toolbox solvers are grouped into four general categories:

- Minimizers on page 2-12

  Solvers in this group attempt to find a local minimum of the objective function near a starting point `x0`. They address problems of unconstrained optimization, linear programming, quadratic programming, cone programming, and general nonlinear programming.

- Multiobjective minimizers on page 2-13

  Solvers in this group attempt 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 specified values (`fgoalattain`).

- Equation solvers on page 2-13

  Solvers in this group attempt 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 on page 2-13

  Solvers in this group attempt 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, finding bounded or linearly constrained solutions, and fitting parameterized nonlinear models to data.

For more information see "Problems Handled by Optimization Toolbox Functions" on page 2-12. See "Optimization Decision Table" on page 2-4 for help choosing a solver 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 2-30.

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 "Write Objective Function". Constraints have a special form, as described in "Write Constraints".

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 2-12.

In this table:

- * means relevant solvers are found in Global Optimization Toolbox (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` and `surrogateopt` functions can address mixed-integer nonlinear programming problems.
- The Statistics and Machine Learning Toolbox™ `bayesopt` function can address low-dimensional deterministic or stochastic optimization problems with combinations of continuous, integer, or categorical variables.

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	mldivide, 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	*
Cone	coneprog, Information	fmincon, Information	fmincon, Information	fmincon, Information	*
General Smooth	fmincon, Information	fmincon, Information	fmincon, Information	fmincon, fseminf, Information	*
Discrete, with Bound or Linear	intlinprog, Information	*	*	*	*

Note This table does not list multiobjective solvers nor equation solvers. See "Problems Handled by Optimization Toolbox Functions" on page 2-12 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 2-6.

Choosing the Algorithm

In this section...

fmincon Algorithms

`fmincon` has five algorithm options:

- `'interior-point'` (default)
- `'trust-region-reflective'`
- `'sqp'`
- `'sqp-legacy'`
- `'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 4-12.   • 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 2-10.

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 2-10. 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 2-10.
- `'sqp-legacy'` is similar to `'sqp'`, but usually is slower and uses more memory.

- `'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 2-10.
- `'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 2-10. 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`.

For descriptions of the algorithms, see "Constrained Nonlinear Optimization Algorithms" on page 6-19.

fsolve Algorithms

fsolve has three algorithms:

- `'trust-region-dogleg'` (default)
- `'trust-region'`
- `'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 4-12.   - To solve equations again if you have a Jacobian multiply function, or want to tune the internal algorithm (see **Trust-Region Algorithm** in fsolve `options`), try `'trust-region'`.   - 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'` algorithm is effective on sparse problems. It can use special techniques such as a Jacobian multiply function for large-scale problems.

For descriptions of the algorithms, see "Equation Solving Algorithms" on page 13-2.

fminunc Algorithms

fminunc has two algorithms:

- `'quasi-newton'` (default)
- `'trust-region'`

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 `SpecifyObjectiveGradient` option to `true`.
• 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 4-12.

For descriptions of the algorithms, see "Unconstrained Nonlinear Optimization Algorithms" on page 6-2.

Least Squares Algorithms

lsqlin

`lsqlin` has three algorithms:

- `'interior-point'`, the default
- `'trust-region-reflective'`
- `'active-set'`

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

Recommendations
• Try `'interior-point'` first.
Tip For better performance when your input matrix `C` has a large fraction of nonzero entries, specify `C` as an ordinary double matrix. Similarly, for better performance when `C` has relatively few nonzero entries, specify `C` as sparse. For data type details, see "Sparse Matrices". You can also set the internal linear algebra type by using the `'LinearSolver'` option.
• If you have no constraints or only bound constraints, and want higher accuracy, more speed, or want to use a "Jacobian Multiply Function with Linear Least Squares" on page 12-30, try `'trust-region-reflective'`.
• If you have a large number of linear constraints and not a large number of variables, try `'active-set'`.
For help if the minimization fails, see "When the Solver Fails" on page 4-3 or "When the Solver Might Have Succeeded" on page 4-12.
See "Potential Inaccuracy with Interior-Point Algorithms" on page 2-10.

For descriptions of the algorithms, see "Least-Squares (Model Fitting) Algorithms" on page 12-2.

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
• Generally, try `'trust-region-reflective'` first. • 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 4-12.

For descriptions of the algorithms, see "Least-Squares (Model Fitting) Algorithms" on page 12-2.

Linear Programming Algorithms

`linprog` has three algorithms:

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

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

Recommendations
Use the `'dual-simplex'` algorithm or 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 4-12. See "Potential Inaccuracy with Interior-Point Algorithms" on page 2-10.

Reasoning Behind the Recommendations

- Often, the `'dual-simplex'` and `'interior-point'` algorithms are fast, and use the least memory.
- The `'interior-point-legacy'` algorithm is similar to `'interior-point'`, but `'interior-point-legacy'` can be slower, less robust, or use more memory.

For descriptions of the algorithms, see "Linear Programming Algorithms" on page 9-2.

Quadratic Programming Algorithms

`quadprog` has three algorithms:

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

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'`.
• **Tip** For better performance when your Hessian matrix H has a large fraction of nonzero entries, specify H as an ordinary double matrix. Similarly, for better performance when H has relatively few nonzero entries, specify H as sparse. For data type details, see "Sparse Matrices". You can also set the internal linear algebra type by using the `'LinearSolver'` option.
• If you have a nonconvex problem with only bounds, or with only linear equalities, use `'trust-region-reflective'`.
• If you have a positive semidefinite problem with a large number of linear constraints and not a large number of variables, try `'active-set'`.
For help if the minimization fails, see "When the Solver Fails" on page 4-3 or "When the Solver Might Have Succeeded" on page 4-12.
See "Potential Inaccuracy with Interior-Point Algorithms" on page 2-10.

For descriptions of the algorithms, see "Quadratic Programming Algorithms" on page 11-2.

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`, `lsqlin`, 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 `StepTolerance`, `OptimalityTolerance`, and possibly `ConstraintTolerance` tolerances (but keep the tolerances sensible.) See "Tolerances and Stopping Criteria" on page 2-68).

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

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

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

x =

   2.0833e-13
```

Using the `linprog` `dual-simplex` algorithm:

```
opts.Algorithm = 'dual-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, multiobjective optimization, equation solving, and solving least-squares (model-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 $A \cdot x \leq b$, $Aeq \cdot x = beq$, $lb \leq x \leq ub$	`linprog`
Mixed-integer linear programming	$\min\limits_{x} f^T x$ such that $A \cdot x \leq b$, $Aeq \cdot x = beq$, $lb \leq x \leq ub$, x(intcon) is integer-valued	`intlinprog`
Quadratic programming	$\min\limits_{x} \frac{1}{2} x^T H x + c^T x$ such that $A \cdot x \leq b$, $Aeq \cdot x = beq$, $lb \leq x \leq ub$	`quadprog`
Cone programming	$\min\limits_{x} f^T x$ such that $\|A \cdot x - b\| \leq d^T \cdot x - \gamma$, $A \cdot x \leq b$, $Aeq \cdot x = beq$, $lb \leq x \leq ub$	`coneprog`
Constrained minimization	$\min\limits_{x} f(x)$ such that $c(x) \leq 0$, $ceq(x) = 0$, $A \cdot x \leq b$, $Aeq \cdot x = beq$, $lb \leq x \leq ub$	`fmincon`
Semi-infinite minimization	$\min\limits_{x} f(x)$ such that $K(x,w) \leq 0$ for all w, $c(x) \leq 0$, $ceq(x) = 0$, $A \cdot x \leq b$, $Aeq \cdot x = beq$, $lb \leq x \leq ub$	`fseminf`

Multiobjective Optimization Problems

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

Equation Solving Problems

Type	Formulation	Solver
Linear equations	$C \cdot x = d$, n equations, n variables	`mldivide` (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	$\min_{x} \frac{1}{2} \lVert C \cdot x - d \rVert_2^2$ m equations, n variables	`mldivide` (matrix left division)
Nonnegative linear least squares	$\min_{x} \frac{1}{2} \lVert C \cdot x - d \rVert_2^2$ such that $x \geq 0$	`lsqnonneg`
Constrained linear least squares	$\min_{x} \frac{1}{2} \lVert C \cdot x - d \rVert_2^2$ such that $A \cdot x \leq b$, $Aeq \cdot x = beq$, $lb \leq x \leq ub$	`lsqlin`
Nonlinear least squares	$\min_{x} \lVert F(x) \rVert_2^2 = \min_{x} \sum_{i} F_i^2(x)$ such that $lb \leq x \leq ub$	`lsqnonlin`
Nonlinear curve fitting	$\min_{x} \lVert F(x, xdata) - ydata \rVert_2^2$ such that $lb \leq x \leq ub$	`lsqcurvefit`

Complex Numbers in Optimization Toolbox Solvers

Generally, Optimization Toolbox solvers do not accept or handle objective functions or constraints with complex values. However, the least-squares solvers `lsqcurvefit`, `lsqnonlin`, and `lsqlin`, and the `fsolve` solver can handle these objective functions under the following restrictions:

*   The objective function must be analytic in the complex function sense (for details, see Nevanlinna and Paatero [1]). For example, the function $f(z) = \text{Re}(z) - i\text{Im}(z)$ is not analytic, but the function $f(z) = \exp(z)$ is analytic. This restriction automatically holds for `lsqlin`.
*   There must be no constraints, not even bounds. Complex numbers are not well ordered, so it is not clear what "bounds" might mean. When there are problem bounds, nonlinear least-squares solvers disallow steps leading to complex values.
*   Do not set the *FunValCheck* option to `'on'`. This option immediately halts a solver when the solver encounters a complex value.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

The least-squares solvers and `fsolve` try to minimize the squared norm of a vector of function values. This makes sense even in the presence of complex values.

If you have a non-analytic function or constraints, split the real and imaginary parts of the problem. For an example, see "Fit a Model to Complex-Valued Data" on page 12-50.

To get the best (smallest norm) solution, try setting a complex initial point. For example, solving $1 + x^4 = 0$ fails if you use a real start point:

```
f = @(x)1+x^4;
x0 = 1;
x = fsolve(f,x0)
```

```
No solution found.

fsolve stopped because the problem appears regular as measured by the gradient,
but the vector of function values is not near zero as measured by the
default value of the function tolerance.

x =

   1.1176e-08
```

However, if you use a complex initial point, `fsolve` succeeds:

```
x0 = 1 + 1i/10;
x = fsolve(f,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 =

   0.7071 + 0.7071i
```

References

[1] Nevanlinna, Rolf, and V. Paatero. *Introduction to Complex Analysis*. Addison-Wesley, 1969.

See Also

Related Examples

- "Fit a Model to Complex-Valued Data" on page 12-50

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 2-17

Nonlinear least squares	`lsqcurvefit`
	
	`lsqnonlin`
		"Writing Vector and Matrix Objective Functions" on page 2-26

Multivariable equation solving	`fsolve`	

Multiobjective	`fgoalattain`
	
	`fminimax`	

Linear programming	`linprog`	"Writing Objective Functions for Linear or Quadratic Problems" on page 2-29
Mixed-integer linear programming	`intlinprog`	

Linear least squares	`lsqlin`
	
	`lsqnonneg`	

Quadratic programming	`quadprog`	

Writing Scalar Objective Functions

In this section...
"Function Files" on page 2-17
"Anonymous Function Objectives" on page 2-18
"Including Gradients and Hessians" on page 2-19

Function Files

A scalar objective function file accepts one input, say `x`, and returns one real scalar output, say `f`. The input `x` can be a scalar, vector, or matrix on page 2-31. A function file can return more outputs (see "Including Gradients and Hessians" on page 2-19).

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 2-57. For more complex examples of function files, see "Minimization with Gradient and Hessian Sparsity Pattern" on page 6-16 or "Minimization with Bound Constraints and Banded Preconditioner" on page 6-84.

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 2-58.

For example, suppose you want to minimize the `myObjective.m` objective function, described in "Function Files" on page 2-17, subject to the `ellipseparabola.m` constraint, described in "Nonlinear Constraints" on page 2-37. 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?". 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 =
  1.2266e-10
```

Including Gradients and Hessians

- "Provide Derivatives For Solvers" on page 2-19
- "How to Include Gradients" on page 2-19
- "Including Hessians" on page 2-21
- "Benefits of Including Derivatives" on page 2-24
- "Choose Input Hessian Approximation for interior-point fmincon" on page 2-24

Provide Derivatives For Solvers

For `fmincon` and `fminunc`, you can include gradients in the objective function. Generally, solvers are more robust, and can be slightly faster when you include gradients. See "Benefits of Including Derivatives" on page 2-24. To also include second derivatives (Hessians), see "Including Hessians" on page 2-21.

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

Solver	Algorithm	Gradient	Hessian
fmincon	active-set	Optional	No
	interior-point	Optional	Optional (see "Hessian for fmincon interior-point algorithm" on page 2-21)
	sqp	Optional	No
	trust-region-reflective	Required	Optional (see "Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms" on page 2-21)
fminunc	quasi-newton	Optional	No
	trust-region	Required	Optional (see "Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms" on page 2-21)

How to Include Gradients

1   Write code that returns:

- The objective function (scalar) as the first output
- The gradient (vector) as the second output

2   Set the `SpecifyObjectiveGradient` option to `true` using `optimoptions`. If appropriate, also set the `SpecifyConstraintGradient` option to `true`.

3   Optionally, check if your gradient function matches a finite-difference approximation. See "Checking Validity of Gradients or Jacobians" on page 2-74.

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 `SpecifyObjectiveGradient` option. Unconditionalized code requires you to set 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, Solver-Based" on page 1-11. The gradient of $f(x)$ is

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

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

```
function [f,g] = rosentwo(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
```

`nargout` checks the number of arguments that a calling function specifies. See "Find Number of Function Arguments".

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',...
    'SpecifyObjectiveGradient',true);
```

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

```
[x fval] = fminunc(@rosentwo,[-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 =
  1.9886e-17
```

If you have a Symbolic Math Toolbox™ license, you can calculate gradients and Hessians automatically, as described in "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

Including Hessians

You can include second derivatives with the fmincon `'trust-region-reflective'` and `'interior-point'` algorithms, and with the fminunc `'trust-region'` algorithm. There are several ways to include Hessian information, depending on the type of information and on the algorithm.

You must also include gradients (set `SpecifyObjectiveGradient` to `true` and, if applicable, `SpecifyConstraintGradient` to `true`) in order to include Hessians.

- "Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms" on page 2-21
- "Hessian for fmincon interior-point algorithm" on page 2-21
- "Hessian Multiply Function" on page 2-23

Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms

These algorithms either have no constraints, or have only bound or linear equality constraints. Therefore the Hessian is the matrix of second derivatives of the objective function.

Include the Hessian matrix as the third output of the objective function. For example, the Hessian $H(x)$ of Rosenbrock's function is (see "How to Include Gradients" on page 2-19)

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

Include this Hessian in the objective:

```
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
```

Set `HessianFcn` to `'objective'`. For example,

```
options = optimoptions('fminunc','Algorithm','trust-region',...
    'SpecifyObjectiveGradient',true,'HessianFcn','objective');
```

Hessian for fmincon interior-point algorithm

The Hessian is the Hessian of the Lagrangian, where the Lagrangian $L(x,\lambda)$ is

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

g and h are vector functions representing all inequality and equality constraints respectively (meaning bound, linear, and nonlinear constraints), so the minimization problem is

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

For details, see "Constrained Optimality Theory" on page 3-12. The Hessian of the Lagrangian is

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

To include a Hessian, write a function with the syntax

```
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` and passes it to your Hessian function. `hessianfcn` must calculate the sums in "Equation 2-2". Indicate that you are supplying a Hessian by setting these options:

```
options = optimoptions('fmincon','Algorithm','interior-point',...
    'SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true,...
    'HessianFcn',@hessianfcn);
```

For example, to include a Hessian for Rosenbrock's function constrained to the unit disk $x_1^2 + x_2^2 \le 1$, notice that the constraint function $g(x) = x_1^2 + x_2^2 - 1 \le 0$ has gradient and second derivative matrix

$$\nabla g(x) = \begin{bmatrix} 2x_1 \\ 2x_2 \end{bmatrix}$$

$$H_g(x) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}.$$

Write the Hessian function as

```
function Hout = hessianfcn(x,lambda)
% Hessian of objective
H = [1200*x(1)^2-400*x(2)+2, -400*x(1);
            -400*x(1), 200];
% Hessian of nonlinear inequality constraint
Hg = 2*eye(2);
Hout = H + lambda.ineqnonlin*Hg;
```

Save `hessianfcn` on your MATLAB path. To complete the example, the constraint function including gradients is

```
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
```

Solve the problem including gradients and Hessian.

```
fun = @rosenboth;
nonlcon = @unitdisk2;
x0 = [-1;2];
options = optimoptions('fmincon','Algorithm','interior-point',...
    'SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true,...
    'HessianFcn',@hessianfcn);
[x,fval,exitflag,output] = fmincon(fun,x0,[],[],[],[],[],[],@unitdisk2,options);
```

For other examples using an interior-point Hessian, see "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-66 and "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

Hessian Multiply Function

Instead of a complete Hessian function, both the fmincon `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 SubproblemAlgorithm option must be `'cg'` for a Hessian multiply function to work; this is the `trust-region-reflective` default.

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 16-1"), 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','SpecifyObjectiveGradient',true,
      'SpecifyConstraintGradient',true,'SubproblemAlgorithm','cg','HessianMultiplyFcn',@HessMult
  ```

  Supply the function HessMultFcn, which returns an *n*-by-1 vector, where *n* is the number of dimensions of *x*. The HessianMultiplyFcn 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 "Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms" on page 2-21). 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',...
      'SpecifyObjectiveGradient',true,'HessianMultiplyFcn',@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 6-93.

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, although a finite-difference estimate can be faster for complicated derivatives. 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 example, see the end of "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

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 `SpecifyConstraintGradient` option to `true`. See "Nonlinear Constraints" on page 2-37.

Choose Input Hessian Approximation for interior-point fmincon

The `fmincon interior-point` algorithm has many options for selecting an input Hessian approximation. For syntax details, see "Hessian as an Input" on page 16-103. Here are the options, along with estimates of their relative characteristics.

Hessian	Relative Memory Usage	Relative Efficiency
`'bfgs'` (default)	High (for large problems)	High
`'lbfgs'`	Low to Moderate	Moderate
`'fin-diff-grads'`	Low	Moderate
`'HessianMultiplyFcn'`	Low (can depend on your code)	Moderate
`'HessianFcn'`	? (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 `SpecifyObjectiveGradient` and `SpecifyConstraintGradient` options to `true`.

- Want more efficiency — Provide your own gradients and Hessian. See "Including Hessians" on page 2-21, "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-66, and "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

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 `HessianMultiplyFcn` 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 6-30, and its discussion of the LDL approach and the conjugate gradient approach to solving "Equation 6-50".

See Also

More About

- "Checking Validity of Gradients or Jacobians" on page 2-74

Writing Vector and Matrix Objective Functions

In this section...
"What Are Vector and Matrix Objective Functions?" on page 2-26
"Jacobians of Vector Functions" on page 2-26
"Jacobians of Matrix Functions" on page 2-27
"Jacobians with Matrix-Valued Independent Variables" on page 2-27

What Are 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 on page 2-17 is how you 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.

For information on complex-valued objective functions, see "Complex Numbers in Optimization Toolbox Solvers" on page 2-14.

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

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

then $J(x)$ is

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

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
```

To indicate to the solver that your objective function includes a Jacobian, set the `SpecifyObjectiveGradient` option to `true`. For example:

```
options = optimoptions('lsqnonlin','SpecifyObjectiveGradient',true);
```

Jacobians of Matrix Functions

To define the Jacobian of a matrix $F(x)$, change 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 defined in terms of 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 f having the vector form of F, the Jacobian of $F(X)$ is defined as the Jacobian of $f(x)$:

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

So, for example,

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

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

See Also

More About

*   "Checking Validity of Gradients or Jacobians" on page 2-74

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 "Nonnegative Linear Least Squares, Solver-Based" on page 12-25.

- `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 and Cone 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 value of the optimality 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 because, to five digits, the maximum is $\tan(1) = 1.5574$, which occurs at $x = 2\pi = 6.2832$.

Matrix Arguments

Optimization Toolbox solvers accept vectors for many arguments, such as the initial point `x0`, lower bounds `lb`, and upper bounds `ub`. They also accept matrices for these arguments, where matrix means an array of any size. When your solver arguments are naturally arrays, not vectors, feel free to provide the arguments as arrays.

Here is 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`, or the "Indexing with a Single Index" section of "Array Indexing".

- 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" on page 2-35, 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.

See Also
`colon`

More About
- "Writing Scalar Objective Functions" on page 2-17
- "Bound Constraints" on page 2-34
- "Linear Constraints" on page 2-35
- "Nonlinear Constraints" on page 2-37
- "Array Indexing"

Types of Constraints

Optimization Toolbox solvers have special forms for constraints:

- "Bound Constraints" on page 2-34 — Lower and upper bounds on individual components; $x \geq l$ and $x \leq u$.
- "Linear Inequality Constraints" on page 2-35 — $A \cdot x \leq b$. A is an m-by-n matrix, which represents m constraints for an n-dimensional vector x. b is m-dimensional.
- "Linear Equality Constraints" on page 2-36 — $Aeq \cdot x = beq$. Equality constraints have the same form as inequality constraints.
- "Nonlinear Constraints" on page 2-37 — $c(x) \leq 0$ and $ceq(x) = 0$. Both c and ceq are scalars or vectors representing several constraints.

Optimization Toolbox functions assume that inequality constraints have the form $c_i(x) \leq 0$ or $A \cdot x \leq b$. 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 $A \cdot x \geq b$ is equivalent to the constraint $-A \cdot x \leq -b$. For more information, see "Linear Inequality Constraints" on page 2-35 and "Nonlinear Constraints" on page 2-37.

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 $5 \, x \leq 20$, 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 2-57.

Iterations Can Violate Constraints

In this section...
"Intermediate Iterations can Violate Constraints" on page 2-33
"Algorithms That Satisfy Bound Constraints" on page 2-33
"Solvers and Algorithms That Can Violate Bound Constraints" on page 2-33

Intermediate 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

See Also

More About

*   "Bound Constraints" on page 2-34

Bound Constraints

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

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

Specify bounds as vectors with the same length as x, or as matrices on page 2-31 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 only need to supply a vector of length m containing bounds. However, this shortcut causes solvers to issue 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] (issues a warning) or u = [Inf; 3; Inf].

Tip To lower memory usage and increase solver speed, use Inf or −Inf instead of a large, arbitrary bound. For more information, see "Use Inf Instead of a Large, Arbitrary Bound" on page 4-10.

You do not have to 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, Solver-Based" on page 1-21.

See Also

More About

- "Iterations Can Violate Constraints" on page 2-33

Linear Constraints

What Are Linear Constraints?

Several optimization solvers accept linear constraints, which are restrictions on the solution x to satisfy linear equalities or inequalities. Solvers that accept linear constraints include `fmincon`, `intlinprog`, `linprog`, `lsqlin`, `quadprog`, multiobjective solvers, and some Global Optimization Toolbox solvers.

Linear Inequality Constraints

Linear inequality constraints have the form $A \cdot x \leq b$. 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.

Pass linear inequality constraints in the `A` and `b` arguments.

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

$x_1 + x_3 \leq 4$,
$2x_2 - x_3 \geq -2$,
$x_1 - x_2 + x_3 - x_4 \geq 9$.

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

Write these constraints 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 are first multiplied by –1 to put them in "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.

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 2-31.

For a more complex example of linear constraints, see "Set Up a Linear Program, Solver-Based" on page 1-21.

Intermediate iterations can violate linear constraints. See "Iterations Can Violate Constraints" on page 2-33.

Linear Equality Constraints

Linear equalities have the form $Aeq \cdot x = beq$, which represents m equations with n-component vector x. You supply the m-by-n matrix Aeq and the m-component vector beq.

Pass linear equality constraints in the Aeq and beq arguments in the same way as described for the A and b arguments in "Linear Inequality Constraints" on page 2-35.

See Also

More About

*   "Write Constraints"

Nonlinear Constraints

Several optimization solvers accept nonlinear constraints, including `fmincon`, `fseminf`, `fgoalattain`, `fminimax`, and the Global Optimization Toolbox solvers `ga`, `gamultiobj`, `patternsearch`, `paretosearch`, `GlobalSearch`, and `MultiStart`. Nonlinear constraints allow you to restrict the solution to any region that can be described in terms of smooth functions.

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 have 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 an empty entry `[]` 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 an empty entry `[]` for `ceq`, the nonlinear equality constraint function. Also, the second inequality is rewritten to ≤ 0 form.

Minimize the function `exp(x(1) + 2*x(2))` subject to the `ellipseparabola` constraints.

```
fun = @(x)exp(x(1) + 2*x(2));
nonlcon = @ellipseparabola;
x0 = [0 0];
A = []; % No other constraints
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x =

   -0.2500   -0.9375
```

Including Gradients in Constraint Functions

If you provide gradients for *c* and *ceq*, the 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:

```matlab
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 2-17 for information on conditionalized functions. The gradient matrix has the form

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

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

To have a solver use gradients of nonlinear constraints, indicate that they exist by using `optimoptions`:

```matlab
options = optimoptions(@fmincon,'SpecifyConstraintGradient',true);
```

Make sure to pass the options structure to the solver:

```matlab
[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 "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

Anonymous Nonlinear Constraint Functions

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 that you have the nonlinear inequalities

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

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

Suppose that you have the nonlinear equality

$x_2 = \tanh(x_1)$.

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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

z = 2×1

  -0.6530
  -0.5737
```

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

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

```
cout = 2×1

  -0.8704
        0


ceqout = 0
```

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

For information on anonymous objective functions, see "Anonymous Function Objectives" on page 2-18.

See Also

GlobalSearch | MultiStart | fgoalattain | fmincon | ga | patternsearch

More About

- "Tutorial for Optimization Toolbox™" on page 6-36
- "Nonlinear Constraints with Gradients" on page 6-63

- "Or Instead of And Constraints" on page 2-41
- "How to Use All Types of Constraints" on page 2-45

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 2-37: you give them as a set of functions that must be negative at a feasible point. If your constraints are

$F_1(x) \leq 0$ OR $F_2(x) \leq 0$ OR $F_3(x) \leq 0$,

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 $-1 \leq x(1) \leq 1$, $0 \leq x(2) \leq 1$ OR x is in the rectangle $0 \leq x(1) \leq 1$, $-1 \leq x(2) \leq 1$.
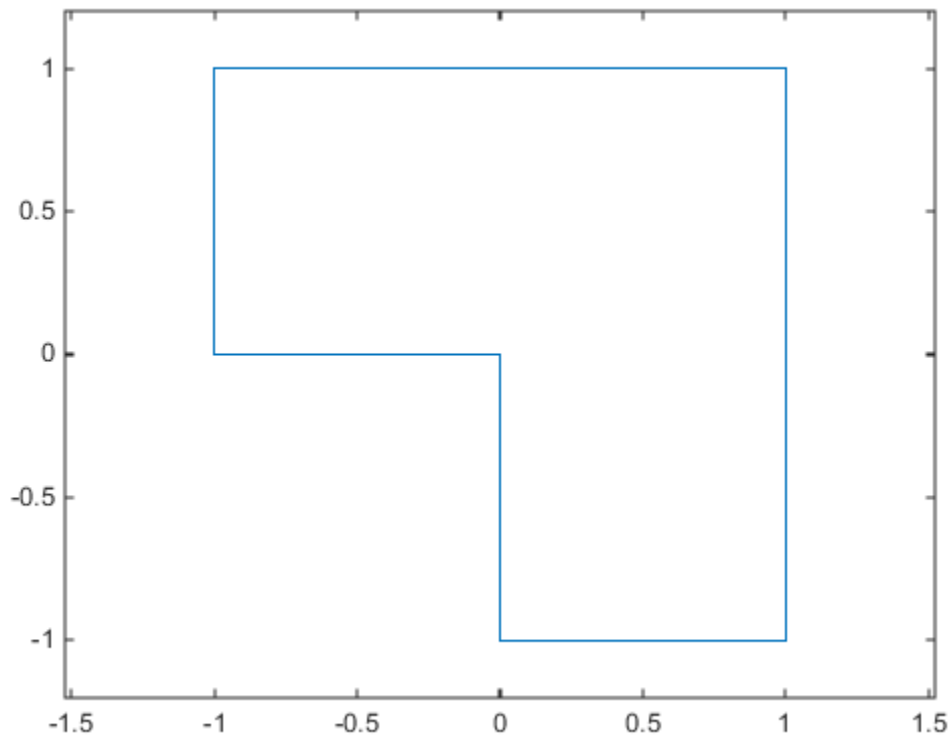
Code for creating the figure

```
% 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 $a \le x(1) \le b, c \le x(2) \le d$:

```
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
```
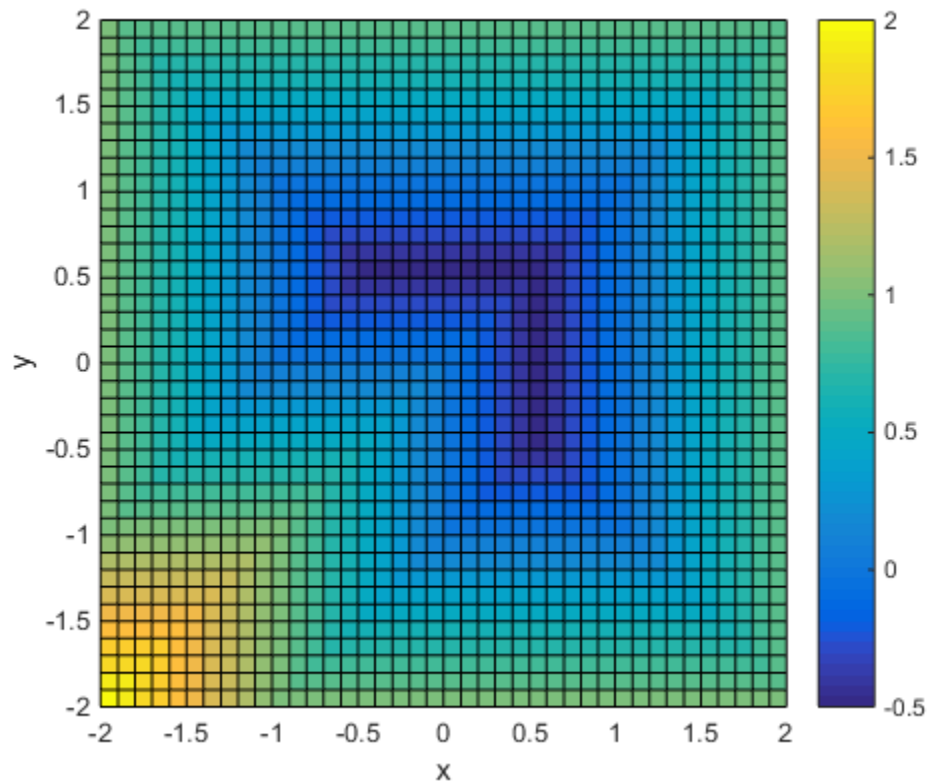

Code for creating the figure

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

```
[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] = fmincon(fun,x0,[],[],[],[],[],[],@rectconstrfcn,opts)

xsol =

    0.4998   -0.9996


fval =

   2.4650e-07


eflag =

     1
```

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

See Also
`fmincon`

More About
- "Nonlinear Constraints" on page 2-37

How to Use All Types of Constraints

This section contains an example of a nonlinear minimization problem with all possible types of constraints. This example does not use gradients.

The problem has five variables, x(1) through x(5). The objective function is a polynomial in the variables.

$$f(x) = 6x_2x_5 + 7x_1x_3 + 3x_2^2.$$

The objective function is in the local function myobj(x), which is nested inside the function fullexample. The code for fullexample appears at the end of this section on page 2-0 .

The nonlinear constraints are likewise polynomial expressions.

$$x_1 - 0.2x_2x_5 \le 71$$

$$0.9x_3 - x_4^2 \le 67$$

$$3x_2^2x_5 + 3x_1^2x_3 = 20.875.$$

The nonlinear constraints are in the local function myconstr(x), which is nested inside the function fullexample.

There are bounds on x_3 and x_5.

$$0 \le x_3 \le 20, x_5 \ge 1.$$

There is a linear equality constraint $x_1 = 0.3x_2$, which you can write as $x_1 - 0.3x_2 = 0$.

There are three linear inequality constraints:

$$0.1x_5 \le x_4$$

$$x_4 \le 0.5x_5$$

$$0.9x_5 \le x_3.$$

Represent the bounds and linear constraints as matrices and vectors. The code that creates these arrays is in the fullexample function. As described in the fmincon "Input Arguments" on page 16-87 section, the lb and ub vectors represent the constraints

lb ≤ x ≤ ub

The matrix A and vector b represent the linear inequality constraints

A*x ≤ b,

and the matrix Aeq and vector beq represent the linear equality constraints

Aeq*x = b.

Call fullexample to solve the minimization problem subject to all types of constraints.

```
[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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x = *5×1*

```
    0.6114
    2.0380
    1.3948
    0.1572
    1.5498
```

fval = 37.3806

exitflag = 1

The exit flag value of 1 indicates that `fmincon` converges to a local minimum that satisfies all of the constraints.

This code creates the `fullexample` function, which contains the nested functions `myobj` and `myconstr`.

```matlab
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;
end

%-------------------------------------------------------
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;
end
end
```

See Also

More About

- "Write Constraints"
- "Solver-Based Nonlinear Optimization"

Objective and Nonlinear Constraints in the Same Function

This example shows how to avoid calling a function twice when it computes values for both the objective and constraints using the solver-based approach. To avoid calling a function twice using the problem-based approach, see "Objective and Constraints Having a Common Function in Serial or Parallel, Problem-Based" on page 2-52.

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, use a nested function to evaluate the objective and constraint functions only when needed, by retaining the values of time-consuming calculations. This approach avoids using global variables, while retaining intermediate results be retained and sharing them between the objective and constraint functions.

Note Because of the way `ga` calls nonlinear constraint functions, the technique in this example usually does not reduce the number of calls to the objective or constraint functions.

Step 1. Write a function that computes the objective and constraints.

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

Write a function that computes a portion of Rosenbrock's function `f1` and includes 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, Solver-Based" on page 1-11.

This example has 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 the function in a 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:

```
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 function
        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. Determine the time to run with the nested function.

Run the function, 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. Determine the time to run without the 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 reduces the computational time in half, compared to the serial run with the nested function.

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.

See Also

More About

- "Objective and Constraints Having a Common Function in Serial or Parallel, Problem-Based" on page 2-52
- "Solve a Constrained Nonlinear Problem, Solver-Based" on page 1-11
- "Optimizing a Simulation or Ordinary Differential Equation" on page 4-26
- "Parallel Computing"

Objective and Constraints Having a Common Function in Serial or Parallel, Problem-Based

This example shows how to avoid calling a function twice when it computes values for both the objective and the constraints using the problem-based approach. For the solver-based approach, see "Objective and Nonlinear Constraints in the Same Function" on page 2-48.

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

This example also shows the effect of parallel computation on solver speed. For time-consuming functions, computing in parallel can speed the solver, as can avoiding calling the time-consuming function repeatedly at the same point. Using both techniques together speeds the solver the most.

Create Time-Consuming Function That Computes Several Quantities

The `computeall` function returns outputs that are part of the objective and nonlinear constraints.

```
type computeall
```

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

The function includes a `pause(1)` statement to simulate a time-consuming function.

Create Optimization Variables

This problem uses a four-element optimization variable.

```
x = optimvar('x',4);
```

Convert Function Using 'ReuseEvaluation'

Convert the `computeall` function to an optimization expression. To save time during the optimization, use the `'ReuseEvaluation'` name-value pair. To save time for the solver to determine the output expression sizes (this happens only once), set the `'OutputSize'` name-value pair to `[1 1]`, indicating that both `f` and `c` are scalar.

```
[f,c] = fcn2optimexpr(@computeall,x,'ReuseEvaluation',true,'OutputSize',[1 1]);
```

Create Objective, Constraint, and Problem

Create the objective function from the `f` expression.

```
obj = f + 20*(x(3) - x(4)^2)^2 + 5*(1 - x(4))^2;
```

Create the nonlinear inequality constraint from the `c` expression.

```
cons = c <= 0;
```

Create an optimization problem and include the objective and constraint.

```
prob = optimproblem('Objective',obj);
prob.Constraints.cons = cons;
show(prob)

  OptimizationProblem :

    Solve for:
       x

    minimize :
       ((arg3 + (20 .* (x(3) - x(4).^2).^2)) + (5 .* (1 - x(4)).^2))

       where:

         [arg3,~] = computeall(x);


    subject to cons:
       arg_LHS <= 0

       where:

         [~,arg_LHS] = computeall(x);
```

Solve Problem

Monitor the time it takes to solve the problem starting from the initial point `x0.x = [-1;1;1;2]`.

```
x0.x = [-1;1;1;2];
x0.x = x0.x/norm(x0.x); % Feasible initial point
tic
[sol,fval,exitflag,output] = solve(prob,x0)

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

<stopping criteria details>

sol = struct with fields:
    x: [4×1 double]


fval = 0.7107

exitflag =
    OptimalSolution


output = struct with fields:
         iterations: 25
          funcCount: 149
      constrviolation: 0
           stepsize: 1.2914e-07
```

```
         algorithm: 'interior-point'
     firstorderopt: 4.0000e-07
      cgiterations: 7
           message: '↵Local minimum found that satisfies the constraints.↵Optimization complete
            solver: 'fmincon'
```

```
time1 = toc
```

```
time1 = 149.9299
```

The number of seconds for the solution is just over the number of function evaluations, which indicates that the solver computed each evaluation only once.

```
fprintf("The number of seconds to solve was %g, and the number of evaluation points was %g.\n",t:
```

```
The number of seconds to solve was 149.93, and the number of evaluation points was 149.
```

If, instead, you do not call `fcn2optimexpr` using `'ReuseEvaluation'`, then the solution time doubles.

```
[f2,c2] = fcn2optimexpr(@computeall,x,'ReuseEvaluation',false);
obj2 = f2 + 20*(x(3) - x(4)^2)^2 + 5*(1 - x(4))^2;
cons2 = c2 <= 0;
prob2 = optimproblem('Objective',obj2);
prob2.Constraints.cons2 = cons2;
tic
[sol2,fval2,exitflag2,output2] = solve(prob2,x0);
```

```
Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

<stopping criteria details>
```

```
time2 = toc
```

```
time2 = 298.4493
```

Parallel Processing

If you have a Parallel Computing Toolbox™ license, you can save even more time by computing in parallel. To do so, set options to use parallel processing, and call `solve` with options.

```
options = optimoptions(prob,'UseParallel',true);
tic
[sol3,fval3,exitflag3,output3] = solve(prob,x0,'Options',options);
```

```
Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

```
<stopping criteria details>
```

```
time3 = toc
```

```
time3 = 74.7043
```

Using parallel processing and `'ReuseEvaluation'` together provides a faster solution than using `'ReuseEvaluation'` alone. See how long it takes to solve the problem using parallel processing alone.

```
tic
[sol4,fval4,exitflag4,output4] = solve(prob2,x0,'Options',options);
```

```
Solving problem using fmincon.
```

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

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

```
<stopping criteria details>
```

```
time4 = toc
```

```
time4 = 145.5278
```

Summary of Timing Results

Combine the timing results into one table.

```
timingtable = table([time1;time2;time3;time4],...
    'RowNames',["Reuse Serial";"No Reuse Serial";"Reuse Parallel";"No Reuse Parallel"])
```

```
timingtable=4×1 table
                          Var1
                         _____

    Reuse Serial         149.93
    No Reuse Serial      298.45
    Reuse Parallel       74.704
    No Reuse Parallel    145.53
```

For this problem, on a computer with a 6-core processor, computing in parallel takes about half the time of computing in serial, and computing with `'ReuseEvaluation'` takes about half the time of computing without `'ReuseEvaluation'`. Computing in parallel with `'ReuseEvaluation'` takes about a quarter of the time of computing in serial without `'ReuseEvaluation'`.

See Also
`fcn2optimexpr`

More About
- "Objective and Nonlinear Constraints in the Same Function" on page 2-48

- "Convert Nonlinear Function to Optimization Expression" on page 7-8
- "Using Parallel Computing in Optimization Toolbox" on page 14-5

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 2-57
- "Nested Functions" on page 2-58
- "Global Variables" on page 2-59

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.

Generally, for problem-based optimization, you pass extra parameters in a natural manner. See "Pass Extra Parameters in Problem-Based Approach" on page 10-11.

For example, suppose you want to minimize the function

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

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 2-3" 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 2-57.

Global Variables

Global variables can be troublesome, so it is better to avoid using them. Also, global variables fail in parallel computations. See "Factors That Affect Results" on page 14-13.

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 2-57.

See Also

More About

*   "Solver-Based Optimization Problem Setup"
*   "Pass Extra Parameters in Problem-Based Approach" on page 10-11

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 2-62.

For an overview of all options, including which solvers use each option, see "Optimization Options Reference" on page 15-6.

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 2-6 — `Algorithm`
- "Tolerances and Stopping Criteria" on page 2-68 — `OptimalityTolerance`, `StepTolerance`, `MaxFunctionEvaluations`, and `MaxIterations`
- Finite differences — `FiniteDifferenceType` and `FiniteDifferenceStepSize`

To diagnose and troubleshoot, try setting these options first:

- "Iterative Display" on page 3-14 — `Display`
- Function evaluation errors — `FunValCheck`
- "Plot Functions" on page 3-27 and "Output Functions for Optimization Toolbox™" on page 3-30 — `PlotFcn` and `OutputFcn`

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, the following code sets the `fmincon` algorithm to `sqp`, specifies iterative display, and sets a small value for the `ConstraintTolerance` tolerance.

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

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

You can change options in several ways. For example, you can use dot notation.

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

Or, you can change options using `optimoptions`.

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

To reset an option to its default, use `resetoptions`.

```
options = resetoptions(options,'StepTolerance');
```

Reset more than one option at a time by passing a cell array of option names, such as `{'Algorithm','StepTolerance'}`.

Note Ensure that you pass `options` in your solver call, as shown in this example.

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

You can also set and change options using the **Optimize** Live Editor Task.

See Also

Optimize | optimoptions | resetoptions

More About

• "Optimization Options Reference" on page 15-6

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 2-62 and "View Options" on page 2-66.

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

```
options = optimoptions(@fminunc,'SpecifyObjectiveGradient',true)

options =

  fminunc options:

  Options used by current Algorithm ('trust-region'):
  (Other available algorithms: 'quasi-newton')

  Set properties:
   SpecifyObjectiveGradient: 1

  Default properties:
                   Algorithm: 'trust-region'
             CheckGradients: 0
                     Display: 'final'
    FiniteDifferenceStepSize: 'sqrt(eps)'
        FiniteDifferenceType: 'forward'
           FunctionTolerance: 1.0000e-06
                   HessianFcn: []
```

```
               HessianMultiplyFcn: []
         MaxFunctionEvaluations: '100*numberOfVariables'
                   MaxIterations: 400
              OptimalityTolerance: 1.0000e-06
                        OutputFcn: []
                          PlotFcn: []
                    StepTolerance: 1.0000e-06
             SubproblemAlgorithm: 'cg'
                         TypicalX: 'ones(numberOfVariables,1)'

    Show options not used by current Algorithm ('trust-region')

options = optimset('GradObj','on')

options =

  struct with fields:

                       Display: []
                    MaxFunEvals: []
                        MaxIter: []
                         TolFun: []
                           TolX: []
                     FunValCheck: []
                       OutputFcn: []
                        PlotFcns: []
                  ActiveConstrTol: []
                       Algorithm: []
         AlwaysHonorConstraints: []
                  DerivativeCheck: []
                      Diagnostics: []
                    DiffMaxChange: []
                    DiffMinChange: []
                   FinDiffRelStep: []
                      FinDiffType: []
               GoalsExactAchieve: []
                       GradConstr: []
                          GradObj: 'on'
                         HessFcn: []
                         Hessian: []
                        HessMult: []
                     HessPattern: []
                      HessUpdate: []
                 InitBarrierParam: []
            InitTrustRegionRadius: []
                        Jacobian: []
                        JacobMult: []
                    JacobPattern: []
                      LargeScale: []
                        MaxNodes: []
                       MaxPCGIter: []
                     MaxProjCGIter: []
                        MaxSQPIter: []
                          MaxTime: []
                   MeritFunction: []
                        MinAbsMax: []
              NoStopIfFlatInfeas: []
                   ObjectiveLimit: []
```

```
     PhaseOneTotalScaling: []
            Preconditioner: []
         PrecondBandWidth: []
            RelLineSrchBnd: []
    RelLineSrchBndDuration: []
              ScaleProblem: []
                   Simplex: []
        SubproblemAlgorithm: []
                    TolCon: []
                 TolConSQP: []
                TolGradCon: []
                    TolPCG: []
                 TolProjCG: []
              TolProjCGAbs: []
                  TypicalX: []
               UseParallel: []
```

See Also

More About

*   "Set Options"

View Options

`optimoptions` "hides" some options, meaning it does not display their values. For example, it hides the `DiffMinChange` option.

```
options = optimoptions('fsolve','DiffMinChange',1e-3)

options =

  fsolve options:

   Options used by current Algorithm ('trust-region-dogleg'):
   (Other available algorithms: 'levenberg-marquardt', 'trust-region')

   Set properties:
     No options set.

   Default properties:
                    Algorithm: 'trust-region-dogleg'
               CheckGradients: 0
                      Display: 'final'
      FiniteDifferenceStepSize: 'sqrt(eps)'
         FiniteDifferenceType: 'forward'
            FunctionTolerance: 1.0000e-06
        MaxFunctionEvaluations: '100*numberOfVariables'
                 MaxIterations: 400
            OptimalityTolerance: 1.0000e-06
                      OutputFcn: []
                        PlotFcn: []
       SpecifyObjectiveGradient: 0
                  StepTolerance: 1.0000e-06
                        TypicalX: 'ones(numberOfVariables,1)'
                     UseParallel: 0

   Show options not used by current Algorithm ('trust-region-dogleg')
```

You can view the value of any option, including "hidden" options, by using dot notation. For example,

```
options.DiffMinChange

ans =

   1.0000e-03
```

Solver reference pages list "hidden" options in italics.

There are two reason that some options are "hidden":

* There are better ways. For example, the `FiniteDifferenceStepSize` option supersedes both the `DiffMinChange` and `DiffMaxChange` options. Therefore, both `DiffMinChange` and `DiffMaxChange` are "hidden".
* They are rarely used, or are difficult to set appropriately. For example, the `fmincon MaxSQPIter` option is recondite and hard to choose, and so is "hidden".
* For a list of hidden options, see "Hidden Options" on page 15-17.

See Also

More About

*   "Optimization Options Reference" on page 15-6

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 2-62.

Tip Generally set tolerances such as `OptimalityTolerance` and `StepTolerance` 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. This tip does not apply to `fzero`, which uses a default value of `eps` for the `TolX` tolerance.

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.
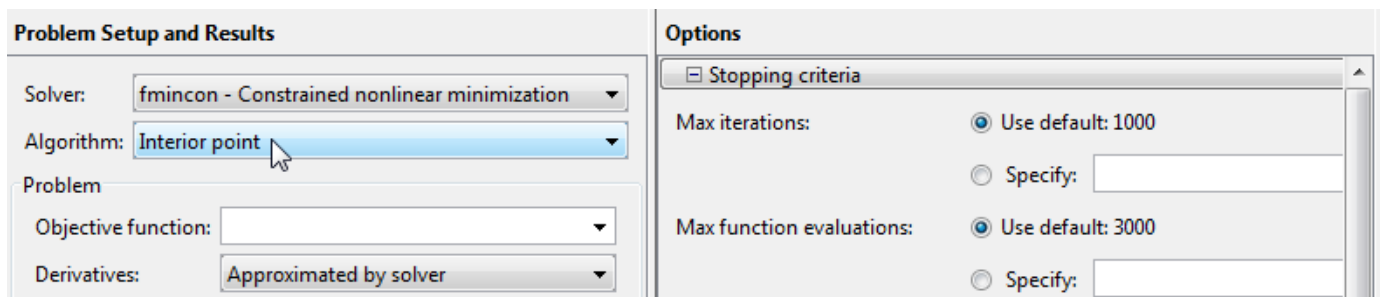
`optimoptions` displays tolerances. For example,

```
options = optimoptions('fmincon');
[options.OptimalityTolerance,options.FunctionTolerance,options.StepTolerance]
```

```
ans =

   1.0e-06 *

   1.0000    1.0000    0.0001
```

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

- `StepTolerance` 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 `StepTolerance`, the iterations end. `StepTolerance` is generally used as a *relative* bound, meaning iterations end when $|(x_i - x_{i+1})| <$ `StepTolerance`$*(1 + |x_i|)$, or a similar relative measure. See "Tolerance Details" on page 2-71.
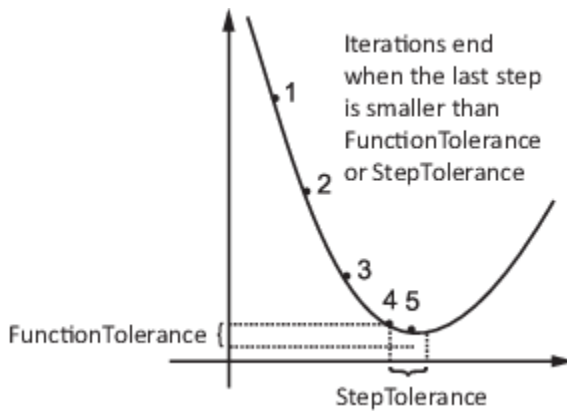
- For some algorithms, `FunctionTolerance` is a lower bound on the change in the value of the objective function during a step. For those algorithms, if $|f(x_i) - f(x_{i+1})| <$ `FunctionTolerance`, the iterations end. `FunctionTolerance` is generally used as a *relative* bound, meaning iterations end when $|f(x_i) - f(x_{i+1})| <$ `FunctionTolerance`$*(1 + |f(x_i)|)$, or a similar relative measure. See "Tolerance Details" on page 2-71.

> **Note** Unlike other solvers, `fminsearch` stops when it satisfies *both* `TolFun` (the function tolerance) and `TolX` (the step tolerance).

- `OptimalityTolerance` is a tolerance for the first-order optimality measure. If the optimality measure is less than `OptimalityTolerance`, the iterations end. `OptimalityTolerance` can also be a relative bound on the first-order optimality measure. See "Tolerance Details" on page 2-71. First-order optimality measure is defined in "First-Order Optimality Measure" on page 3-11.

- `ConstraintTolerance` is an upper bound on the magnitude of any constraint functions. If a solver returns a point x with $c(x) >$ `ConstraintTolerance` or $|ceq(x)|$ > `ConstraintTolerance`, the solver reports that the constraints are violated at x. `ConstraintTolerance` can also be a relative bound. See "Tolerance Details" on page 2-71.

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

- `MaxIterations` is a bound on the number of solver iterations. `MaxFunctionEvaluations` is a bound on the number of function evaluations. Iterations and function evaluations are discussed in "Iterations and Function Counts" on page 3-9.

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 6-21.

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 9-53 and "Branch and Bound" on page 9-48.

See Also

More About

- "Tolerance Details" on page 2-71
- "Optimization Options Reference" on page 15-6

Tolerance Details

Optimization Toolbox solvers use tolerances to decide when to stop iterating and to measure solution quality. See "Tolerances and Stopping Criteria" on page 2-68.

For the four most important tolerances, this section describes which tolerances are relative, meaning scale in some sense with problem size or values, and which are absolute, meaning do not scale with the problem. In the following table,

- **R** means Relative.
- **A** means Absolute.
- . means inapplicable.
- **A*** means Absolute when the tolerances are checked; however, preprocessing can scale the entries to some extent, so the tolerances can be considered relative.
- **A***, **R** means the constraints are first checked as Absolute. If this check passes, the solver returns a positive exit flag. If this check fails then the constraints are checked as Relative. If this check passes, the solver returns a positive exit flag with "poor feasibility". If this check fails, the solver returns a negative exit flag.

Tolerances by Solver and Algorithm

Solver	Algorithm	Optimality Tolerance	Function Tolerance	Step Tolerance	Constraint Tolerance
fmincon	'interior-point'	**R**	.	**R**	**R**
	'sqp'	**R**	.	**R**	**R**
	'sqp-legacy'	**R**	.	**R**	**R**
	'active-set'	**A**	**A**	**A**	**A**
	'trust-region-reflective'	**A**	**R**	**A**	.
fminunc	'quasi-newton'	**R**	.	**R**	.
	'trust-region'	**A**	**R**	**A**	.
fminsearch		.	**A**	**A**	.
fminbnd		.	**A**	**R**	.
fseminf		**A**	**A**	**A**	**A**
fgoalattain		**A**	**A**	**A**	**A**
fminimax		**A**	**A**	**A**	**A**
linprog	'dual-simplex'	**A***	.	.	**A***, **R**
	'interior-point'	**R**	.	.	**R**
	'interior-point-legacy'	**R**	.	.	.
intlinprog		**A***	.	.	**A***, **R**
quadprog	'interior-point-convex'	**R**	.	**R**	**R**
	'trust-region-reflective', bounds	**A**	**R**	**A**	.
	'trust-region-reflective', linear equalities	.	.	.	.
	'active-set'	**R**	.	**A**	**R**
coneprog		**R**			**R**
lsqlin	'interior-point'	**R**	.	**R**	**R**

Solver	Algorithm	Optimality Tolerance	Function Tolerance	Step Tolerance	Constraint Tolerance
	'trust-region-reflective'	**A**	**R**	**A**	.
lsqnonneg		.	.	**R**	.
lsqnonlin	'trust-region-reflective'	**A**	**R**	**A**	.
	'levenberg-marquardt'	**R**	**R**	**R**	.
lsqcurvefit	'trust-region-reflective'	**A**	**R**	**A**	.
	'levenberg-marquardt'	**R**	**R**	**R**	.
fsolve	'trust-region-dogleg'	**A**	**R**	**R**	.
	'trust-region'	**A**	**R**	**A**	.
	'levenberg-marquardt'	**R**	**R**	**R**	.
fzero		.	.	**R**	.

See Also

More About

- "Tolerances and Stopping Criteria" on page 2-68

Checking Validity of Gradients or Jacobians

In this section...
"Check Gradient or Jacobian in Objective Function" on page 2-74
"How to Check Derivatives" on page 2-74
"Example: Checking Derivatives of Objective and Constraint Functions" on page 2-74

Check Gradient or Jacobian in Objective Function

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 `CheckGradients` 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 `CheckGradients` in the function count; see "Iterations and Function Counts" on page 3-9.

How to Check Derivatives

At the MATLAB command line:

1   Set the `SpecifyObjectiveGradient` or `SpecifyConstraintGradient` options to `true` using `optimoptions`. Make sure your objective or constraint functions supply the appropriate derivatives.

2   Set the `CheckGradients` option to `true`.

Central finite differences are more accurate than the default forward finite differences. To use central finite differences at the MATLAB command line, set `FiniteDifferenceType` option to `'central'` using `optimoptions`.

Example: Checking Derivatives of Objective and Constraint Functions

- "Objective and Constraint Functions" on page 2-74
- "Checking Derivatives at the Command Line" on page 2-75

Objective and Constraint Functions

Consider the problem of minimizing the Rosenbrock function within the unit disk as described in "Solve a Constrained Nonlinear Problem, Solver-Based" on page 1-11. 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 `CheckGradients` option:

```
% For reproducibility--CheckGradients randomly perturbs the initial point
rng(0,'twister');
options = optimoptions(@fmincon,'Algorithm','interior-point',...
    'CheckGradients',true,'SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true);
```

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...
```

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.

See Also

More About

- "Solver Outputs and Iterative Display"

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 3-4
"Exit Message Options" on page 3-7

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 3-9, and also see "Tolerances and Stopping Criteria" on page 2-68).

This table links to the exit flag description of each solver.

Exit Flags by Solver

coneprog exitflag	fgoalattain exitflag	fminbnd exitfl
fmincon exitflag	fminimax exitflag	fminsearch exi
fminunc exitflag	fseminf Output Arguments	fsolve exitfla
fzero exitflag	intlinprog exitflag	linprog exitfl
lsqcurvefit exitflag	lsqlin exitflag	lsqnonlin exit
lsqnonneg exitflag	quadprog exitflag	

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 value of the function tolerance, and
the problem appears regular as measured by the gradient.

xfinal =
    0.5671    0.5671

fval =
  1.0e-06 *
   -0.4059
   -0.4059

exitflag =
     1
```

In the table for the `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, such as "Define and Solve Problem at Command Line" on page 1-16. 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 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 gradient
[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 value of the optimality 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+00, is less
  than options.OptimalityTolerance = 1.000000e-06.
  ```

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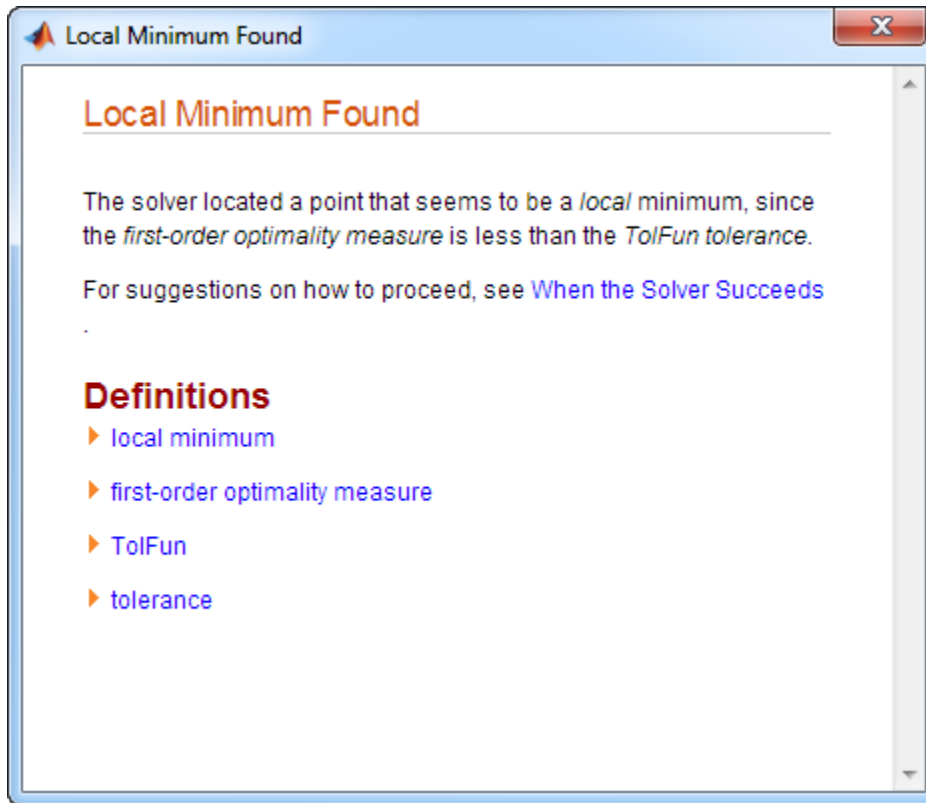
Clicking the `first-order optimality measure` expander link brings up the definition of first-order optimality measure for `fminunc`:
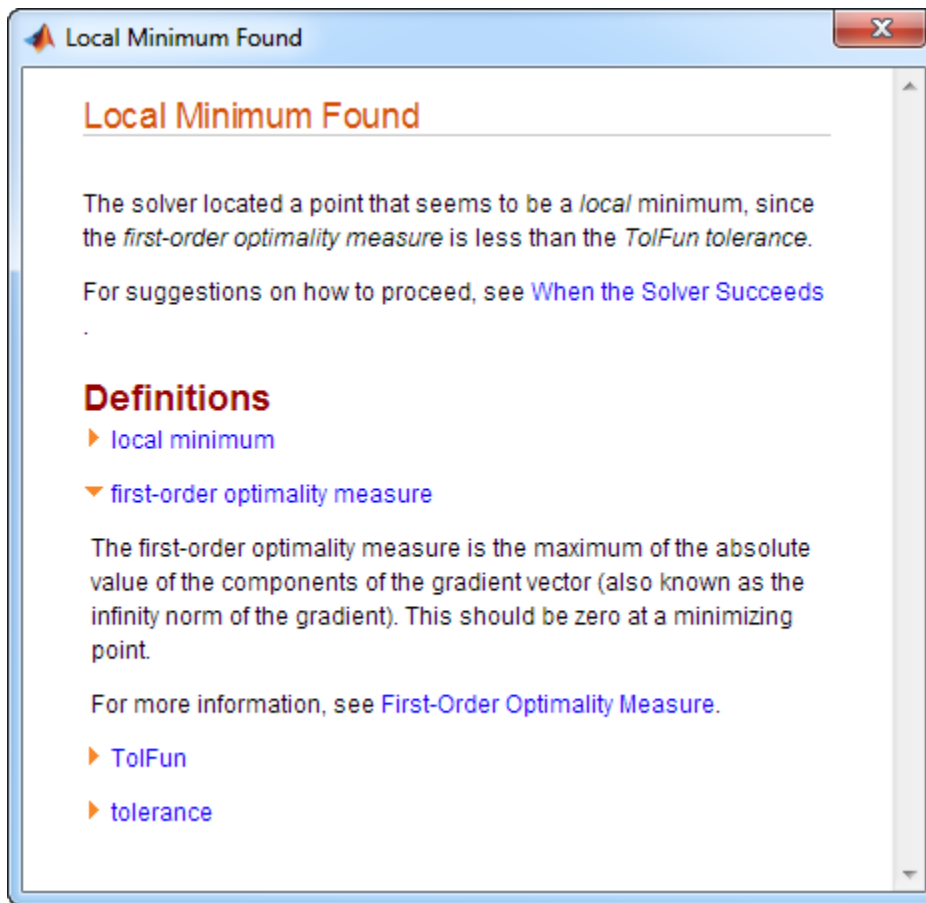
The expander link is a way to obtain more information in the same window. Clicking the `first-order optimality measure` expander link again closes the definition.

- 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 3-14. 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 `exitflag` ≤ 0	None
`'notify-detailed'`	Detailed only if `exitflag` ≤ 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.OptimalityTolerance = 1.000000e-06.
```

See Also

More About

- "Solver Outputs and Iterative Display"

Iterations and Function Counts

In general, Optimization Toolbox solvers iterate to find an optimum. 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.

You can limit the number of iterations or function counts by setting the `MaxIterations` or `MaxFunctionEvaluations` options for a solver using `optimoptions`. Or, if you want a solver to continue after reaching one of these limits, raise the values of these options. See "Set and Change Options" on page 2-62.

At any step, intermediate calculations can involve evaluating the objective function and any constraints at points near the current iterate x_i. For example, the solver might estimate a gradient by finite differences. At each nearby point, the function count (`F-count`) increases by one. The figure "Typical Iteration in 3-D Space" on page 3-9 shows that, in 3-D space with forward finite differences of size delta, one iteration typically corresponds to an increase in function count of four. In the figure, e_i represents the unit vector in the ith coordinate direction.

One iteration

Four F-count:
$f(x_j+\delta e_1)$, $f(x_j+\delta e_2)$,
$f(x_j+\delta e_3)$, $f(x_{j+1})$
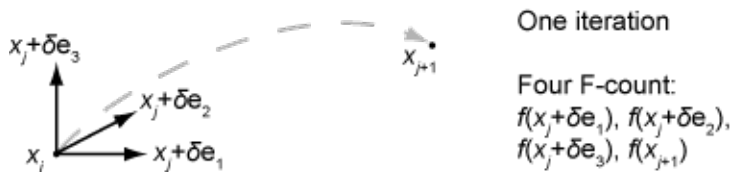
Typical Iteration in 3-D Space

- If the problem has no constraints, the `F-count` reports the total number of objective function evaluations.

- If the problem has constraints, the `F-count` reports only the number of points where function evaluations took place, not the total number of evaluations of constraint functions. So, if the problem has many constraints, the `F-count` can be significantly less than the total number of function evaluations.

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 failed attempts as iterations, and do not report the attempts in the iterative display. All attempted steps increase the `F-count`, regardless of the algorithm.

`F-count` is a header in the iterative display for many solvers. For an example, see "Interpret Result" on page 1-19.

The `F-count` appears in the output structure as `output.funcCount`, enabling you to access the evaluation count programmatically. For more information, see "Output Structures" on page 3-21.

See Also
optimoptions

More About

- "Solver Outputs and Iterative Display"

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 3-11, "Constrained Optimality Theory" on page 3-12, and "Constrained Optimality in Solver Form" on page 3-13.

Stopping Rules Related to First-Order Optimality

The `OptimalityTolerance` tolerance relates to the first-order optimality measure. Typically, if the first-order optimality measure is less than `OptimalityTolerance`, 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 `OptimalityTolerance`, 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 on page 3-4 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:

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

This measure of optimality is based on the familiar condition for a smooth function to achieve a minimum: its gradient must be zero. For unconstrained problems, when the first-order optimality measure is nearly zero, the objective function has gradient nearly zero, so the objective function could be 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 3-13.

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):

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

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:

$$L(x, \lambda) = f(x) + \sum \lambda_{g,i} g_i(x) + \sum \lambda_{h,i} h_i(x). \tag{3-1}$$

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:

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

$$\lambda_{g,i} g_i(x) = 0 \ \forall i, \tag{3-3}$$

$$\begin{cases} g(x) \leq 0, \\ h(x) = 0, \\ \lambda_{g,i} \geq 0. \end{cases} \tag{3-4}$$

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

$$\|\nabla_x L(x, \lambda)\| = \left\| \nabla f(x) + \sum \lambda_{g,i} \nabla g_i(x) + \sum \lambda_{h,i} \nabla h_{h,i}(x) \right\|. \tag{3-5}$$

The optimality measure associated with "Equation 3-3" is

$$\left\|\overrightarrow{\lambda_g g(x)}\right\|, \tag{3-6}$$

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 `ConstraintTolerance` violations. See "Tolerances and Stopping Criteria" on page 2-68.

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":

$$\|\nabla_x L(x, \lambda)\| = \left\| \nabla f(x) + A^T \lambda_{ineqlin} + Aeq^T \lambda_{eqlin} \right.$$
$$\left. + \sum \lambda_{ineqnonlin, i} \nabla c_i(x) + \sum \lambda_{eqnonlin, i} \nabla ceq_i(x) \right\|, \tag{3-7}$$

$$\left\| \overrightarrow{|l_i - x_i| \lambda_{lower, i}}, \overrightarrow{|x_i - u_i| \lambda_{upper, i}}, \overrightarrow{|(Ax - b)_i| \lambda_{ineqlin, i}}, \overrightarrow{|c_i(x)| \lambda_{ineqnonlin, i}} \right\|, \tag{3-8}$$

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 3-22. 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.

See Also

More About

*   "Solver Outputs and Iterative Display"

Iterative Display

In this section...
"Introduction" on page 3-14
"Common Headings" on page 3-14
"Function-Specific Headings" on page 3-15

Introduction

The 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. The table appears in the MATLAB Command Window when you run solvers with appropriate options. For more information about iterations, see "Iterations and Function Counts" on page 3-9.

Obtain the iterative display by using `optimoptions` 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-05
    4         12            -1             1             0

Local minimum found.

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

The iterative display is available for all solvers except:

- `lsqlin 'trust-region-reflective'` algorithm
- `lsqnonneg`
- `quadprog 'trust-region-reflective'` algorithm

Common Headings

This table lists some common headings of iterative display.

Heading	Information Displayed
`f(x)` or `Fval`	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 3-9

Heading	Information Displayed
`Iteration` or `Iter`	Iteration number; see "Iterations and Function Counts" on page 3-9
`Norm of step`	Size of the current step (size is the Euclidean norm, or 2-norm). For the `'trust-region'` and `'trust-region-reflective'` algorithms, when constraints exist, `Norm of step` is the norm of `D*s`. Here, `s` is the step and `D` is a diagonal scaling matrix described in the trust-region subproblem section of the algorithm description.

Function-Specific Headings

The tables in this section describe headings of the iterative display whose meaning is specific to the optimization function you are using.

fgoalattain, fmincon, fminimax, and fseminf

This 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 6-21)
`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 6-43")
`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`

fgoalattain, fmincon, fminimax, or fseminf Heading	Information Displayed
Procedure	Hessian update procedures: • Infeasible start point • Hessian not updated • Hessian modified • Hessian modified twice For more information, see "Updating the Hessian Matrix" on page 6-25. QP subproblem procedures: • dependent — The solver detected and removed dependent (redundant) equality constraints. • 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 poorly conditioned.
Steplength	Multiplicative factor that scales the search direction (see "Equation 6-43")
Trust-region radius	Current trust-region radius

fminbnd and fzero

This 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) 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

This 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 6-9.

fminunc

This 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 6-21)
`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.

3-17

fsolve

This 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 12-6
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

This table describes the headings specific to `intlinprog`.

intlinprog 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 value is an upper bound for the final objective function value
relative gap (%)	$\dfrac{100(b-a)}{\lvert b \rvert + 1}$, where • b is the objective function value of the best integer feasible point. • a is the best lower bound on the objective function value. **Note** Although you specify `RelativeGapTolerance` as a decimal number, the iterative display and `output.relativegap` report the gap as a percentage, meaning 100 times the measured relative gap. If the exit message refers to the relative gap, this value is the measured relative gap, not a percentage.

linprog

This table describes the headings specific to `linprog`. Each algorithm has its own iterative display.

linprog Heading	Information Displayed
Primal Infeas A*x-b or Primal Infeas	Primal infeasibility, a measure of the constraint violations, which should be zero at a solution. For definitions, see "Predictor-Corrector" on page 9-3 (`'interior-point'`) or "Main Algorithm" on page 9-6 (`'interior-point-legacy'`) or "Dual-Simplex Algorithm" on page 9-9.

linprog Heading	Information Displayed
`Dual Infeas A'*y +z-w-f` or `Dual Infeas`	Dual infeasibility, a measure of the derivative of the Lagrangian, which should be zero at a solution. For the definition of the Lagrangian, see "Predictor-Corrector" on page 9-3. For the definition of dual infeasibility, see "Predictor-Corrector" on page 9-3 (`'interior-point'`) or "Main Algorithm" on page 9-6 (`'interior-point-legacy'`) or "Dual-Simplex Algorithm" on page 9-9.
`Upper Bounds {x} +s-ub`	Upper bound feasibility. {x} means those *x* with finite upper bounds. This value is the r_u residual in "Interior-Point-Legacy Linear Programming" on page 9-6.
`Duality Gap x'*z +s'*w`	Duality gap (see "Interior-Point-Legacy Linear Programming" on page 9-6) between the primal objective and the dual objective. `s` and `w` appear in this equation only if the problem has finite upper bounds.
`Total Rel Error`	Total relative error, described at the end of "Main Algorithm" on page 9-6
`Complementarity`	A measure of the Lagrange multipliers times distance from the bounds, which should be zero at a solution. See the r_c variable in "Stopping Conditions" on page 9-6.
`Time`	Time in seconds that `linprog` has been running

lsqlin

The `lsqlin` `'interior-point'` iterative display is inherited from the `quadprog` iterative display. The relationship between these functions is explained in "Linear Least Squares: Interior-Point or Active-Set" on page 12-2. For iterative display details, see "quadprog" on page 3-19.

lsqnonlin and lsqcurvefit

This 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 12-6
`Resnorm`	Value of the squared 2-norm of the residual at `x`
`Residual`	Residual vector of the function

quadprog

This table describes the headings specific to `quadprog`. Only the `'interior-point-convex'` algorithm has the iterative display.

quadprog Heading	Information Displayed
`Primal Infeas`	Primal infeasibility, defined as `max( norm(Aeq*x - beq, inf), abs(min(0, min(A*x-b))) )`
`Dual Infeas`	Dual infeasibility, defined as `norm(H*x + f - A*lambda_ineqlin - Aeq*lambda_eqlin, inf)`

quadprog Heading	Information Displayed
Complementarity	A measure of the maximum absolute value of the Lagrange multipliers of inactive inequalities, which should be zero at a solution. This quantity is g in "Infeasibility Detection" on page 11-7.

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(...)
```

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`.

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

See Also

More About

• "Solver Outputs and Iterative Display"

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` because 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
- `soc`, associated with second-order cone constraints

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."

Examine the Lagrange multiplier structure for the solution of a nonlinear problem with linear and nonlinear inequality constraints and bounds.

```
lb = [-3 -3]; % lower bounds
ub = [3 3]; % upper bounds
A = [1 1]; % linear inequality x(1) + x(2) <= 1
b = 1;
Aeq = [];
beq = [];
x0 = [-1 1];
fun = @(x)100*(x(2) - x(1)^2)^2 + (1 - x(1))^2; % Rosenbrock function
nlcons = @(x)deal(x(1)^2 + x(2)^2 - 1,[]); % nonlinear inequality
options = optimoptions('fmincon','Display','off');
[x,fval,exitflag,output,lambda] = ...
    fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nlcons,options);

disp(lambda)

        eqlin: [0×1 double]
     eqnonlin: [0×1 double]
      ineqlin: 0.3407
        lower: [2×1 double]
        upper: [2×1 double]
   ineqnonlin: 1.7038e-07
```

Here is an interpretation of the Lagrange multiplier structure.

- The `lambda.eqlin` and `lambda.eqnonlin` fields have size 0 because there are no linear equality constraints and no nonlinear equality constraints.

- The `lambda.ineqlin` field has value `0.3407`, indicating that the linear inequality constraint is active. The linear inequality constraint is `x(1) + x(2) <= 1`. Check that the constraint is active at the solution, meaning the solution causes the inequality to be an equality:

```
x(1) + x(2)

ans =

    1.0000
```

- Check the values of the `lambda.lower` and `lambda.upper` fields.

```
lambda.lower

ans =

   1.0e-07 *

   0.2210
   0.2365
```

```
lambda.upper

ans =

   1.0e-07 *

   0.3361
   0.3056
```

These values are effectively zero, indicating that the solution is not near the bounds.

- The value of the `lambda.ineqnonlin` field is `1.7038e-07`, indicating that this constraint is not active. Check the constraint, which is `x(1)^2 + x(2)^2 <= 1`.

```
x(1)^2 + x(2)^2

ans =

    0.5282
```

The nonlinear constraint function value is not near its limit, so the Lagrange multiplier is approximately 0.

See Also

More About

- "Solver Outputs and Iterative Display"

Hessian Output

The `fminunc` and `fmincon` solvers return an approximate Hessian as an optional output.

```
[x,fval,exitflag,output,grad,hessian] = fminunc(fun,x0)
% or
[x,fval,exitflag,output,lambda,grad,hessian] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
```

This topic describes the meaning of the returned Hessian, and the accuracy you can expect.

You can also specify the type of Hessian that the solvers use as input Hessian arguments. For `fminunc`, see "Including Gradients and Hessians" on page 2-19. For `fmincon`, see "Hessian as an Input" on page 16-103.

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. `fminunc` computes the estimate by finite differences, so the estimate is generally accurate.

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

  - If you supply a Hessian in the objective function and set the `HessianFcn` option to `'objective'`, `fminunc` returns this Hessian.

  - If you supply a `HessianMultiplyFcn` function, `fminunc` returns the `Hinfo` matrix from the `HessianMultiplyFcn` function. For more information, see `HessianMultiplyFcn` 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 `trust-region` algorithm returns the Hessian at the next-to-last iterate 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 and, therefore, 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 3-22. 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 several algorithms, with several options for Hessians, as described in "fmincon Trust Region Reflective Algorithm" on page 6-19, "fmincon Active Set Algorithm" on page 6-22, and "fmincon Interior Point Algorithm" on page 6-30.

- **active-set, sqp, or sqp-legacy 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. In general, this approximation does not match the true Hessian in every component, but only in certain subspaces. Therefore, the Hessian returned by `fmincon` can be inaccurate. For more details about the `active-set` calculation, see "SQP Implementation" on page 6-25.

- **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 and set the `HessianFcn` option to `'objective'`, `fmincon` returns this Hessian.

  - If you supply a `HessianMultiplyFcn` function, `fmincon` returns the `Hinfo` matrix from the `HessianMultiplyFcn` 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 `trust-region-reflective` algorithm returns the Hessian at the next-to-last iterate 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 and, therefore, does not compute the Hessian.

- **interior-point Algorithm**

  - If the `HessianApproximation` option is `'lbfgs'` or `'finite-difference'`, or if you supply a `HessianMultiplyFcn` function, `fmincon` returns `[]` for the Hessian.

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

  - If the `HessianFcn` option is a function handle, `fmincon` returns this function as the Hessian at the final point.

See Also

More About

- "Including Gradients and Hessians" on page 2-19
- "Hessian as an Input" on page 16-103

Plot Functions

Plot an Optimization During Execution

You can plot various measures of progress during the execution of a solver. Set the `PlotFcn` 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 `PlotFcn` option description in the solver function reference page.

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 and Plot Function Syntax" on page 15-27.

Use a Plot Function

This example shows how to use plot functions to view the progress of the `fmincon 'interior-point'` algorithm. The problem is taken from "Solve a Constrained Nonlinear Problem, Solver-Based" on page 1-11.

Write the nonlinear objective and constraint functions, including their gradients. The objective function is Rosenbrock's function.

```
type rosenbrockwithgrad
```

```
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 file as `rosenbrockwithgrad.m`.

The constraint function is that the solution satisfies `norm(x)^2 <= 1`.

```
type unitdisk2
```

```
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
```

Save this file as `unitdisk2.m`.

Create an options structure that includes calling three plot functions:

```
options = optimoptions(@fmincon,'Algorithm','interior-point',...
 'SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true,...
 'PlotFcn',{@optimplotx,@optimplotfval,@optimplotfirstorderopt});
```

Create the initial point `x0 = [0,0]`, and set the remaining inputs to empty (`[]`).

```
x0 = [0,0];
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
```

Call `fmincon`, including the options.

```
fun = @rosenbrockwithgrad;
nonlcon = @unitdisk2;
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```
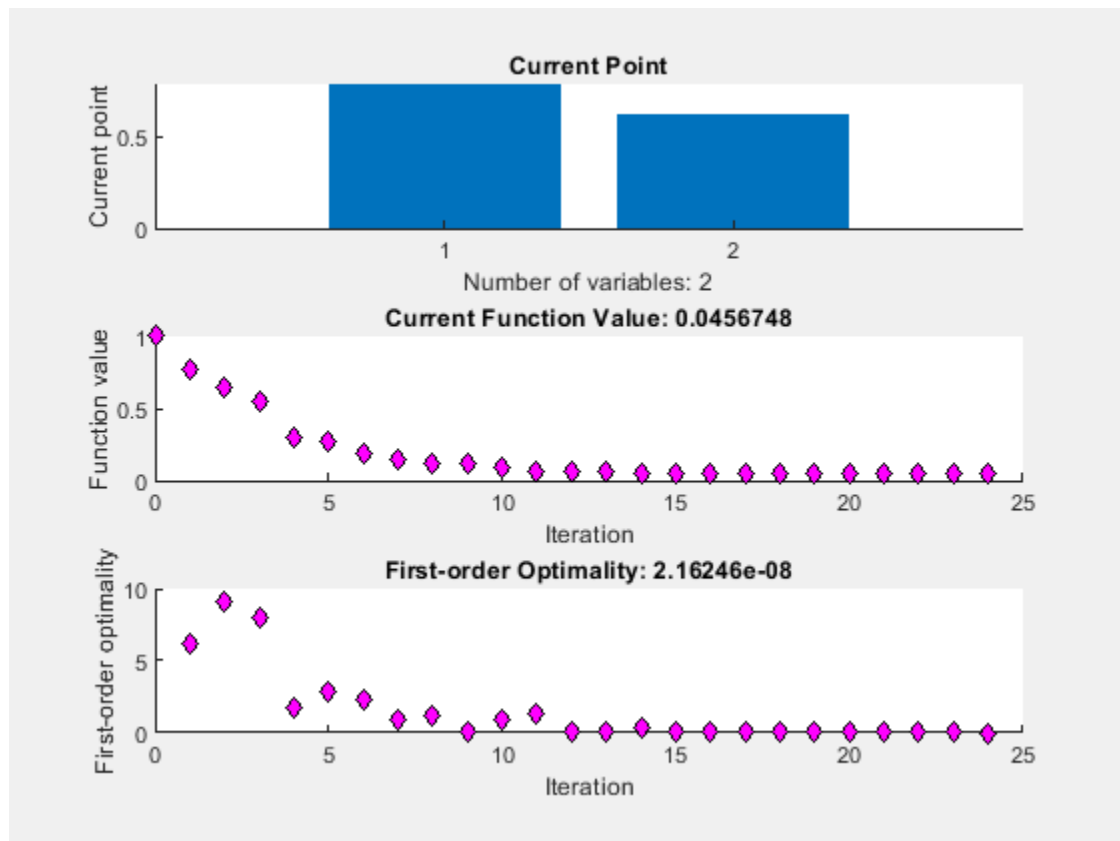

Local minimum found that satisfies the constraints.

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

```
x = 1×2

    0.7864    0.6177
```

See Also

More About

- "Output Function and Plot Function Syntax" on page 15-27
- "Output Function for Problem-Based Optimization" on page 7-38

Output Functions for Optimization Toolbox™

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 and Plot Function Syntax" on page 15-27 for details and syntax.

This section shows the solver-based approach to output functions. For the problem-based approach, see "Output Function for Problem-Based Optimization" on page 7-38.

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.

Example: Use an Output Function

This example shows how to use an output function to monitor the `fmincon` solution process for solving a constrained nonlinear optimization problem. The output function does the following at the end of each `fmincon` iteration:

- Plot the current point.
- Store the current point and its corresponding objective function value in a variable called `history`, and store 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.

Additionally, to make the history available outside of the `fmincon` function, perform the optimization inside a nested function that calls `fmincon` and returns the output function variables. For more information about this method of passing information, see "Passing Extra Parameters" on page 2-57. The `runfmincon` function at the end of this example on page 3-0   contains the nested function call.

Objective and Constraint Functions

The problem is to minimize the function

$$f(x) = \exp(x_1)\big(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\big)$$

subject to the nonlinear inequality constraints

$$x_1 + x_2 - x_1x_2 \geq 3/2$$
$$x_1x_2 \geq -10 \, .$$

The `objfun` function nested in `runfmincon` on page 3-0   implements the objective function. The `confun` function nested in `runfmincon` on page 3-0   implements the constraint function.

Call Solver

To obtain the solution to the problem and see the history of the `fmincon` iterations, call the `runfmincon` function.

```
[xsol,fval,history,searchdir] = runfmincon;

                          Max     Line search  Directional  First-order
  Iter F-count        f(x)  constraint   steplength   derivative   optimality Procedure
```

0	3	1.8394	0.5				Infeasible start p
1	6	1.85127	-0.09197	1	0.109	0.778	
2	9	0.300167	9.33	1	-0.117	0.313	Hessian modified
3	12	0.529835	0.9209	1	0.12	0.232	
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	9.035e-08	1	-0.0146	8.51e-07	

```
Active inequalities (to within options.ConstraintTolerance = 1e-06):
  lower      upper     ineqlin    ineqnonlin
                                      1
                                      2
```
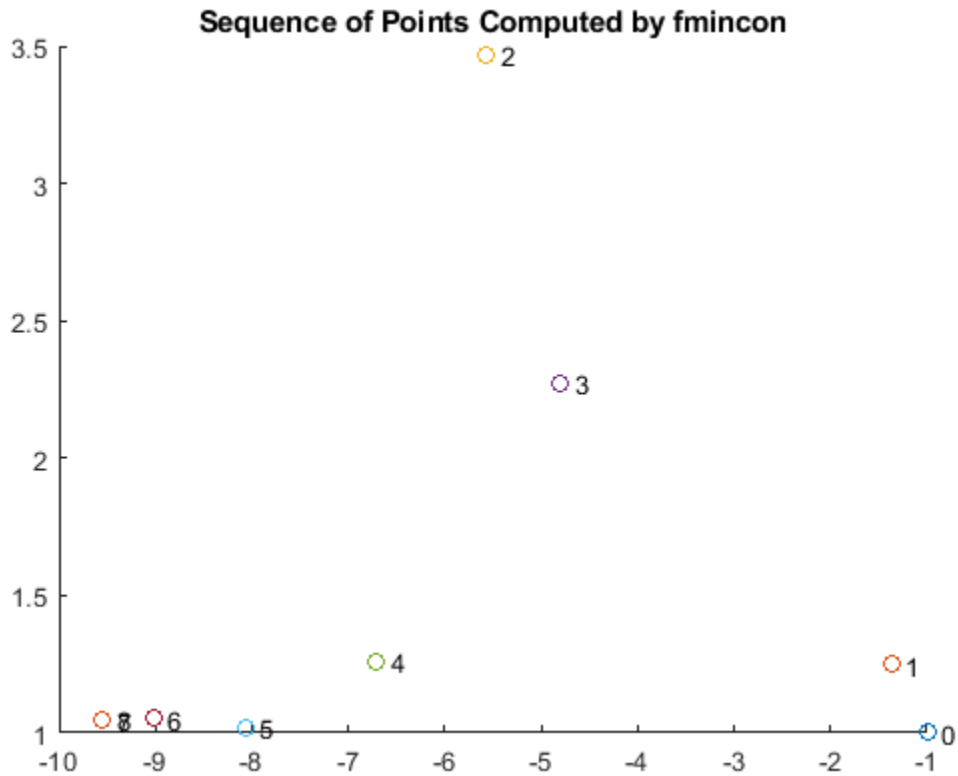

Sequence of Points Computed by fmincon

```
Local minimum found that satisfies the constraints.

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

3-31

The output function creates a plot of the points that `fmincon` evaluates. Each point is labeled by its iteration number. The optimal point occurs at the eighth iteration. The last two points in the sequence are so close that they overlap.

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

```
disp(history)
```

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

The `fval` field in `history` contains the objective function values corresponding to the sequence of points `fmincon` computes.

```
disp(history.fval)
```

```
    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 that `fmincon` computes.

```
disp(history.x)
```

```
   -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
```

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.

```
disp(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
```

Helper Functions

The following code creates the `runfmincon` function, containing the `outfun` output function, `objfun` objective function, and `confun` nonlinear constraint function as nested functions,

```matlab
function [xsol,fval,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,fval] = 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
```

See Also

More About

- "Output Function and Plot Function Syntax" on page 15-27
- "Output Function for Problem-Based Optimization" on page 7-38

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 4-22)
- 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.

See Also

Related Examples
- "When the Solver Fails" on page 4-3
- "Solver Takes Too Long" on page 4-9
- "When the Solver Might Have Succeeded" on page 4-12
- "When the Solver Succeeds" on page 4-18

When the Solver Fails

In this section...
"Too Many Iterations or Function Evaluations" on page 4-3
"Converged to an Infeasible Point" on page 4-6
"Problem Unbounded" on page 4-7
"fsolve Could Not Solve Equation" on page 4-8

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" on page 4-3
"2. Relax Tolerances" on page 4-4
"3. Start the Solver From Different Points" on page 4-4
"4. Check Objective and Constraint Function Definitions" on page 4-4
"5. Center and Scale Your Problem" on page 4-4
"6. Provide Gradient or Jacobian" on page 4-5
"7. Provide Hessian" on page 4-5

1. Enable Iterative Display

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

To enable iterative display 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 "Interpret Result" on page 1-19.

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 `MaxIterations` and/or `MaxFunctionEvaluations` to values larger than the defaults. You can see the default values 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 `StepTolerance` or `OptimalityTolerance`, 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 at the command line, use `optimoptions` as described in "Set and Change Options" on page 2-62.

The `FiniteDifferenceStepSize` option (or `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 on page 4-18.

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 on page 4-20. Check that an infeasible point does not cause an error in your functions; see "Iterations Can Violate Constraints" on page 2-33.

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. The problem-based approach can provide gradients automatically; see "Automatic Differentiation in Optimization Toolbox" on page 10-41.

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 2-17, "Writing Vector and Matrix Objective Functions" on page 2-26, and "Nonlinear Constraints" on page 2-37.

To check that your gradient or Jacobian function is correct, use the `CheckGradients` option, as described in "Checking Validity of Gradients or Jacobians" on page 2-74.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

For examples using gradients and Jacobians, see "Minimization with Gradient and Hessian" on page 6-13, "Nonlinear Constraints with Gradients" on page 6-63, "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97, "Solve Nonlinear System Without and Including Jacobian" on page 13-7, and "Large Sparse System of Nonlinear Equations with Jacobian" on page 13-10. For automatic differentiation in the problem-based approach, see "Effect of Automatic Differentiation in Problem-Based Optimization" on page 7-23.

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 6-66.
- `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 6-93.
- `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 6-13.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97. To provide a Hessian in the problem-based approach, see "Supply Derivatives in Problem-Based Workflow" on page 7-26.

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 `ConstraintTolerance` 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 4-7. If `quadprog` returns this result, see "quadprog Converges to an Infeasible Point" on page 4-7

To proceed when the solver found no feasible point, try one or more of the following.
"1. Check Linear Constraints" on page 4-6
"2. Check Nonlinear Constraints" on page 4-6

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. For details on checking linear constraints, see "Investigate Linear Infeasibilities" on page 9-158.

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.

quadprog Converges to an Infeasible Point

Usually, you get this message because the linear constraints are inconsistent, or are nearly singular. To check whether a feasible point exists, create a linear programming problem with the same constraints and with a zero objective function vector `f`. Solve using the `linprog` `'dual-simplex'` algorithm:

```
options = optimoptions('linprog','Algorithm','dual-simplex');
x = linprog(f,A,b,Aeq,beq,lb,ub,options)
```

If `linprog` finds no feasible point, then your problem is truly infeasible.

If `linprog` finds a feasible point, then try a different `quadprog` algorithm. Alternatively, change some tolerances such as `StepTolerance` or `ConstraintTolerance` and solve the problem again.

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 2-30.

- Try scaling or centering your problem. See Center and Scale Your Problem on page 4-4.

- 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 on page 4-18. `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 `OptimalityTolerance` and `StepTolerance`. 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`.

See Also

More About

- "Investigate Linear Infeasibilities" on page 9-158

Solver Takes Too Long

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

1. "Enable Iterative Display" on page 4-9
2. "Use Appropriate Tolerances" on page 4-9
3. "Use a Plot Function" on page 4-9
4. "Use 'lbfgs' HessianApproximation Option" on page 4-10
5. "Enable CheckGradients" on page 4-10
6. "Use Inf Instead of a Large, Arbitrary Bound" on page 4-10
7. "Use an Output Function" on page 4-10
8. "Use a Sparse Solver or a Multiply Function" on page 4-10
9. "Use Parallel Computing" on page 4-11

Enable Iterative Display

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

To enable iterative display 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 "Interpret Result" on page 1-19. For more information, see "What to Look For in Iterative Display" on page 4-3.

Use Appropriate Tolerances

Solvers can fail to converge if tolerances are too small, especially `OptimalityTolerance` and `StepTolerance`.

To change tolerances at the command line, use `optimoptions` as described in "Set and Change Options" on page 2-62.

Use a Plot Function

You can obtain more visual or detailed information about solver iterations using a plot function. The Options section of your solver's function reference pages lists the plot functions.

To use a plot function at the MATLAB command line, enter

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

Call the solver using the `options` structure.

For an example of using a plot function, see "Use a Plot Function" on page 3-27.

Use 'lbfgs' HessianApproximation Option

For the `fmincon` solver, if you have a problem with many variables (hundreds or more), then oftentimes you can save time and memory by setting the `HessianApproximation` option to `'lbfgs'`. This causes the `fmincon` `'interior-point'` algorithm to use a low-memory Hessian approximation.

Enable CheckGradients

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 `CheckGradients` option, see "Checking Validity of Gradients or Jacobians" on page 2-74.

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 and Plot Function Syntax" on page 15-27.

For an example of using an output function, see "Output Functions for Optimization Toolbox™" on page 3-30.

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 2-10). 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.

  Tip If you use a large-scale algorithm, then 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 12-30, "Quadratic Minimization with

Dense, Structured Hessian" on page 11-24, and "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-93.

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 4-12
"Local Minimum Possible" on page 4-12

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 on page 4-18.
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 on page 4-20. Check that an infeasible point does not cause an error in your functions; see "Iterations Can Violate Constraints" on page 2-33.
3   Changing tolerances, such as `OptimalityTolerance`, `ConstraintTolerance`, and `StepTolerance` — see Use Appropriate Tolerances on page 4-9.
4   Scaling your problem so each coordinate has about the same effect — see Rescale the Problem on page 4-15.
5   Providing gradient and Hessian information — see Provide Analytic Gradients or Jacobian on page 4-16 and Provide a Hessian on page 4-16.

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 `OptimalityTolerance` 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 4-12
"2. Rerun Starting At Final Point" on page 4-13
"3. Try a Different Algorithm" on page 4-13
"4. Change Tolerances" on page 4-15
"5. Rescale the Problem" on page 4-15
"6. Check Nearby Points" on page 4-15
"7. Change Finite Differencing Options" on page 4-16
"8. Provide Analytic Gradients or Jacobian" on page 4-16
"9. Provide a Hessian" on page 4-16

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 on page 4-19.

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 the example "Using Symbolic Mathematics with Optimization Toolbox™ Solvers" on page 6-108:

```
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 on page 4-13 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','SpecifyObjectiveGradient',true);
[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 `OptimalityTolerance` in the Try a Different Algorithm on page 4-13 section:

```
options = optimoptions('fminunc','Algorithm','trust-region',...
    'OptimalityTolerance',1e-8,'SpecifyObjectiveGradient',true); % 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 on page 4-4.

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 on page 4-19 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 `'FiniteDifferenceType'` to `'central'`, instead of the default `'forward'`.

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. The problem-based approach can provide gradients automatically; see "Automatic Differentiation in Optimization Toolbox" on page 10-41.

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 2-17, "Writing Vector and Matrix Objective Functions" on page 2-26, and "Nonlinear Constraints" on page 2-37.

To check that your gradient or Jacobian function is correct, use the `CheckGradients` option, as described in "Checking Validity of Gradients or Jacobians" on page 2-74.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

For examples using gradients and Jacobians, see "Minimization with Gradient and Hessian" on page 6-13, "Nonlinear Constraints with Gradients" on page 6-63, "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97, "Solve Nonlinear System Without and Including Jacobian" on page 13-7, and "Large Sparse System of Nonlinear Equations with Jacobian" on page 13-10. For automatic differentiation in the problem-based approach, see "Effect of Automatic Differentiation in Problem-Based Optimization" on page 7-23.

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 6-66.
- `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 6-93.
- `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 6-13.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97. To provide a Hessian in the problem-based approach, see "Supply Derivatives in Problem-Based Workflow" on page 7-26.

The example in "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97 shows `fmincon` taking 77 iterations without a Hessian, but only 19 iterations with a Hessian.

When the Solver Succeeds

In this section...
"What Can Be Wrong If The Solver Succeeds?" on page 4-18
"1. Change the Initial Point" on page 4-18
"2. Check Nearby Points" on page 4-19
"3. Check your Objective and Constraint Functions" on page 4-20

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 4-22.

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 4-22.

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 4-18, 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.ConstraintTolerance = 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 2-33.

Local vs. Global Optima

In this section...

Why the Solver Does Not Find the Smallest Minimum

In general, solvers return a local minimum (or optimum). The result might be a global minimum (or optimum), but this result is not guaranteed.

*   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.
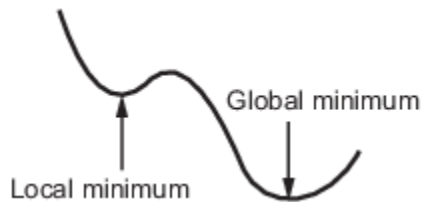
Optimization Toolbox solvers typically find a local minimum. (This local minimum can be a global minimum.) They find the minimum in the basin of attraction of the starting point. For more information about basins of attraction, see "Basins of Attraction" on page 4-23.

The following are exceptions to this general rule.

*   Linear programming problems and positive definite quadratic programming problems are convex, with convex feasible regions, so they have only one basin of attraction. Depending on the specified options, `linprog` ignores any user-supplied starting point, and `quadprog` does not require one, even though you can sometimes speed a minimization by supplying a starting point.
*   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 minimum, you must find an initial value for your solver in the basin of attraction of a global minimum.

You can set initial values to search for a global minimum in these ways:

*   Use a regular grid of initial points.
*   Use random points drawn from a uniform distribution if all of the problem coordinates are 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 minimum, 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 greatly 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 increases. Generally, initial values $x(0)$ that are near each other give steepest descent paths that tend towards the same minimum point. The basin of attraction for steepest descent is the set of initial values that lead to the same local minimum.

This figure shows two one-dimensional minima. The figure shows different basins of attraction with different line styles, and indicates the 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 $x(0)$.
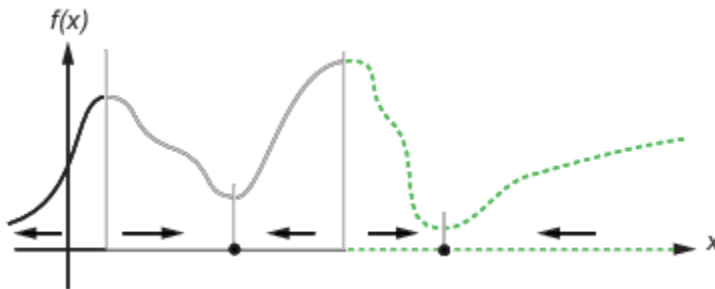
One-dimensional basins

This figure shows how steepest descent paths can be more complicated in more dimensions.
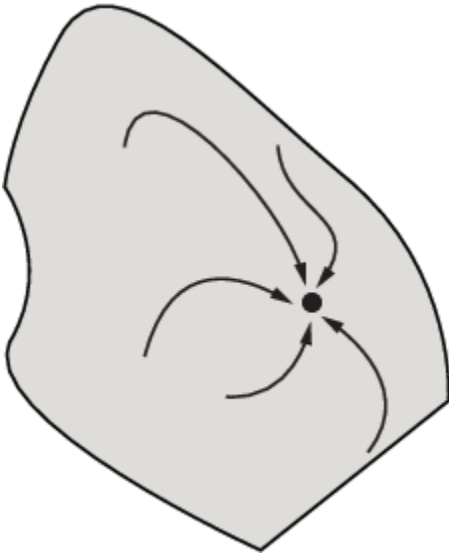
One basin of attraction, showing steepest descent paths from various starting points

This figure shows even more complicated paths and basins of attraction.
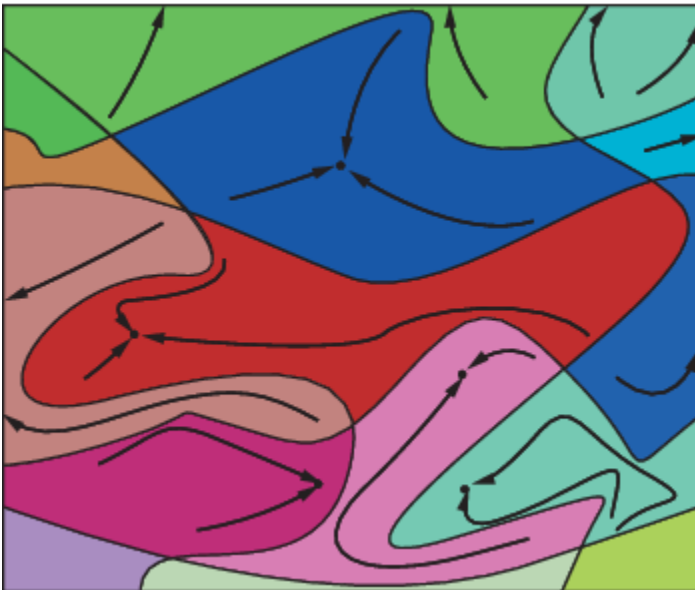
Several basins of attraction

Constraints can break up one basin of attraction into several pieces. For example, consider minimizing y subject to:

$y \geq |x|$

$y \geq 5 - 4(x-2)^2$.

This figure shows the two basins of attraction with the final points.
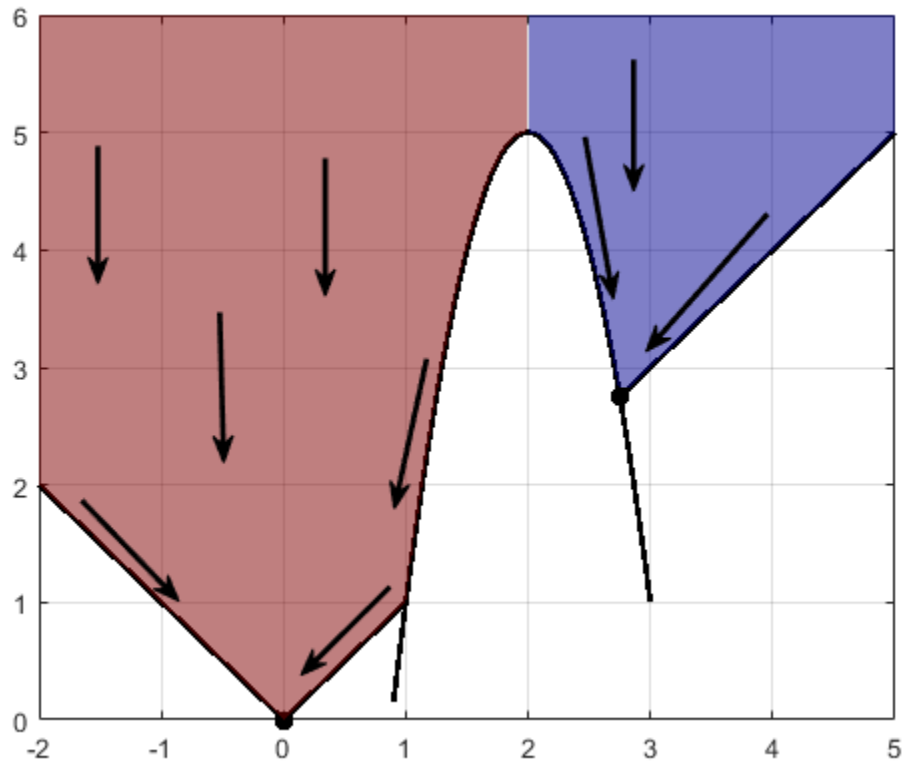
The steepest descent paths are straight lines down to the constraint boundaries. From the constraint boundaries, the steepest descent paths travel down along the boundaries. The final point is either $(0,0)$ or $(11/4,11/4)$, depending on whether the initial x-value is above or below 2.

See Also

More About

- "Improve Results"

Optimizing a Simulation or Ordinary Differential Equation

In this section...
"What Is Optimizing a Simulation or ODE?" on page 4-26
"Potential Problems and Solutions" on page 4-26
"Bibliography" on page 4-30

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 4-26.

For a problem-based example of optimizing an ODE, see "Fit ODE, Problem-Based" on page 12-67. For a solver-based example, see "Fit an Ordinary Differential Equation (ODE)" on page 12-54.

To optimize a Simulink® model easily, try using Simulink Design Optimization™.

Potential Problems and Solutions

- "Problems in Finite Differences" on page 4-26
- "Suggestions for Finite Differences" on page 4-27
- "Problems in Stochastic Functions" on page 4-29
- "Suggestions for Stochastic Functions" on page 4-29
- "Common Calculation of Objective and Constraints" on page 4-29
- "Failure in Objective or Constraint Function Evaluation" on page 4-29
- "Suggestions for Evaluation Failures" on page 4-29

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 4-27
- "Set Larger Finite Differences" on page 4-27
- "Use a Gradient Evaluation Function" on page 4-28
- "Use Tighter ODE Tolerances" on page 4-28

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 4-26.

If you use `patternsearch` for expensive (time-consuming) function evaluations, use the `Cache` option:

```
options = optimoptions('patternsearch','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 4-26 by taking larger finite difference steps than the default.

- If you have MATLAB R2011b or later, set a finite difference step size 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–R2015b and a solver named `'solvername'`:

    ```
    options = optimoptions('solvername','FinDiffRelStep',1e-3);
    ```
  - For R2016a onwards and a solver named `'solvername'`:

    ```
    options = optimoptions('solvername','FiniteDifferenceStepSize',1e-3);
    ```

If you have different scales in different components, set the finite difference step size 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','FiniteDifferenceType','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 `SpecifyObjectiveGradient` option to `true` using `optimoptions`, and, if relevant, also set the `SpecifyConstraintGradient` option to `true`.

- 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 2-48, or, when using the problem-based approach, "Objective and Constraints Having a Common Function in Serial or Parallel, Problem-Based" on page 2-52.

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 2-33, 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 2-33.

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. Use **Optimize** or the problem-based approach instead.

In this section...
"Optimization App Basics" on page 5-2
"Specifying Certain Options" on page 5-6
"Importing and Exporting Your Work" on page 5-8

Optimization App Basics

- "How to Open the Optimization App" on page 5-2
- "Examples that Use the Optimization App" on page 5-3
- "Steps for Using the Optimization App" on page 5-4
- "Pausing and Stopping" on page 5-4
- "Viewing Results" on page 5-4
- "Final Point" on page 5-5
- "Starting a New Problem" on page 5-6

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.
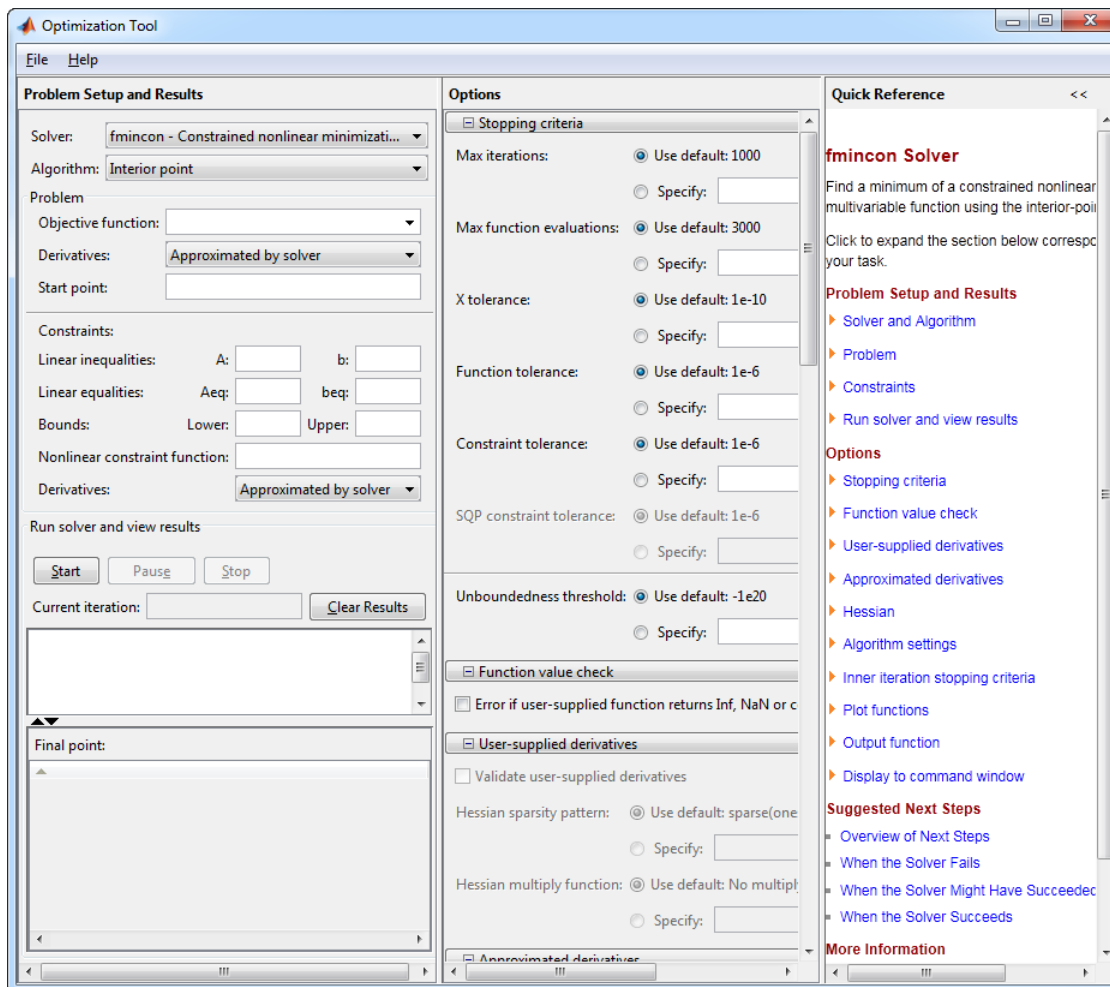
You can also start the Optimization app from the MATLAB **Apps** tab.
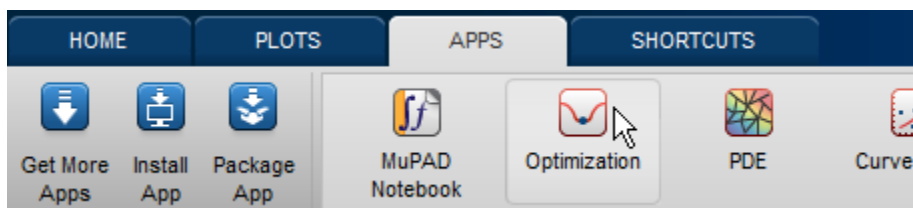
The reference page for the Optimization app provides variations for starting the `optimtool` function.

Examples that Use the Optimization App

The following documentation examples use the optimization app:

- "Solve a Constrained Nonlinear Problem, Solver-Based" on page 1-11
- "Optimize Live Editor Task with fmincon Solver" on page 6-77
- "Optimize Live Editor Task with lsqlin Solver" on page 12-28
- "Plot Functions" on page 3-27

- "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-66

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.

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 5-8.

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 "Optimize Live Editor Task with lsqlin Solver" on page 12-28. The result appears as follows.

Run solver and view results

Start	Pause	Stop

Current iteration: 1   Clear Results

Optimization running.
Objective function value: 2.3333333333333333
Minimum found that satisfies the constraints.

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

Final point:

Index ▲	Value
1	0.333
2	0.667
3	1.333

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 "Pareto Front for Two Objectives" (Global Optimization Toolbox).

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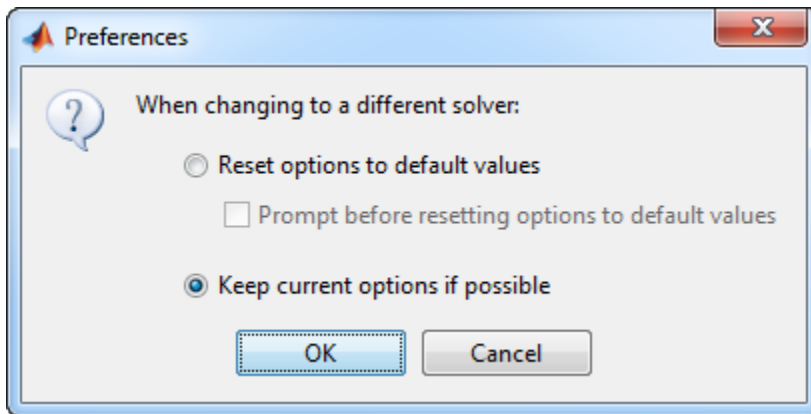
The default value, **Reset options to defaults**, discards any options you specified previously in the `optimtool`. Under this choice, you can select the option **Prompt before resetting options to defaults**.

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 5-6
- "Output function" on page 5-7
- "Display to Command Window" on page 5-7

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. To interrupt a running solver, use the **Stop** button in the figure. 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 predefined 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 and Plot Function Syntax" on page 15-27.

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 and Plot Function Syntax" on page 15-27.

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 3-4 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.
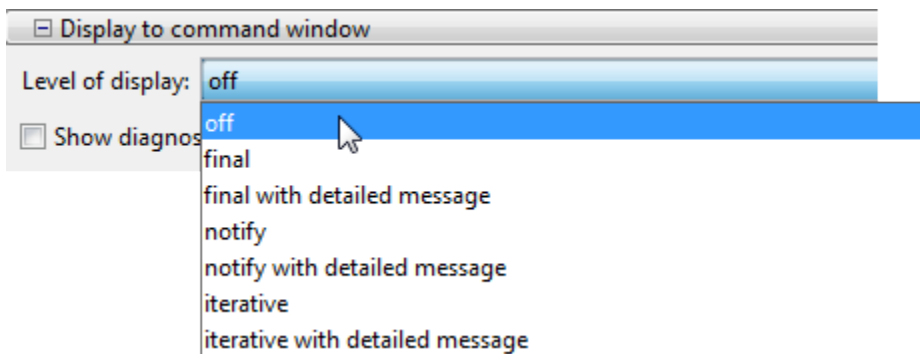
Importing and Exporting Your Work

- "Exporting Your Work" on page 5-8
- "Importing Your Work" on page 5-10
- "Generating a File" on page 5-10

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**.
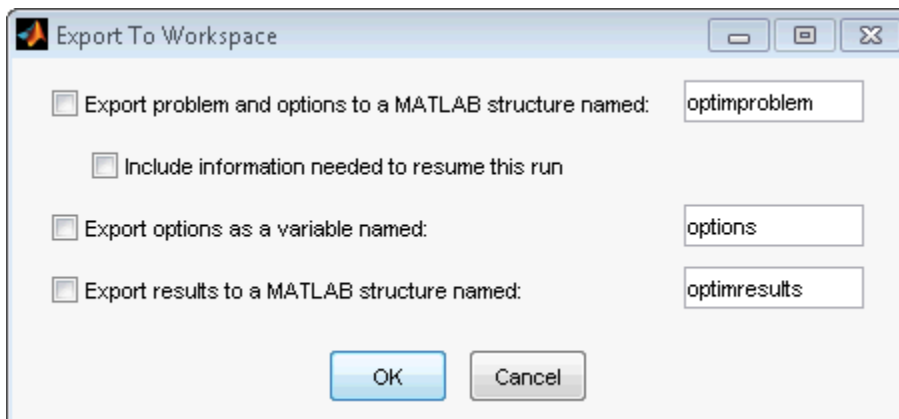
You can specify results that contain:

- 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.

Caution For Optimization Toolbox solvers, the Optimization app imports and exports only one option related to the former `TolFun` tolerance. It displays this option as **Function tolerance**, and uses it as the `OptimalityTolerance` option. You cannot import, export, or change the `FunctionTolerance` option in the Optimization app.

However, Global Optimization Toolbox solvers do not have an `OptimalityTolerance` option. Those solvers can import, export, and set the `FunctionTolerance` option in the Optimization app.

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.

See Also

More About

- "Optimization App Alternatives" on page 5-12
- "Solver-Based Optimization Problem Setup"

Optimization App Alternatives

In this section...
"Optimize Without Using the App" on page 5-12
"Set Options Using Live Scripts" on page 5-12
"Set Options: Command Line or Standard Scripts" on page 5-14
"Choose Plot Functions" on page 5-15
"Pass Solver Arguments" on page 5-16

Optimize Without Using the App

Although the Optimization app affords convenient ways to set options and run problems, it will be removed in a future release. This topic describes other ways to accomplish optimization tasks without using the app.

- Set options easily — "Set Options Using Live Scripts" on page 5-12 or "Set Options: Command Line or Standard Scripts" on page 5-14
- Monitor the optimization — "Choose Plot Functions" on page 5-15
- Pass solver arguments correctly — "Pass Solver Arguments" on page 5-16

Set Options Using Live Scripts

Beginning with R2018a, live scripts show suggestions for `optimoptions` names and values.

1 On the **Home** tab, in the **File** section, click **New Live Script** to create a live script.
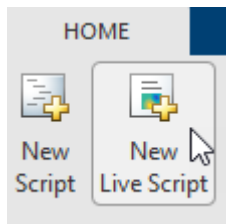
2 In the Live Editor, set options by entering `options = optimoptions(`. MATLAB shows a list of solvers.
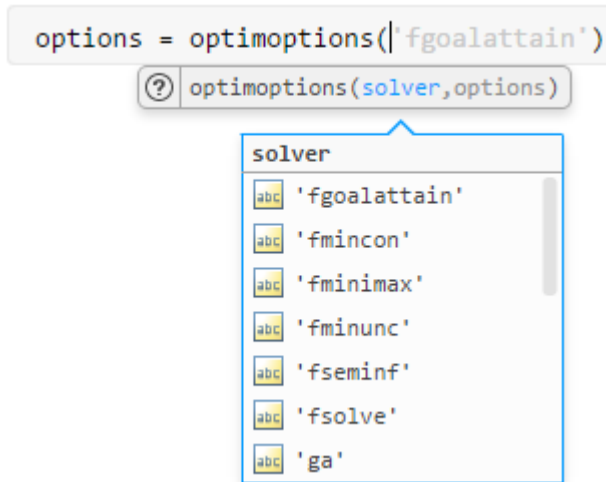
3 Select a solver, then enter a comma. MATLAB displays a list of name-value pairs for the solver.
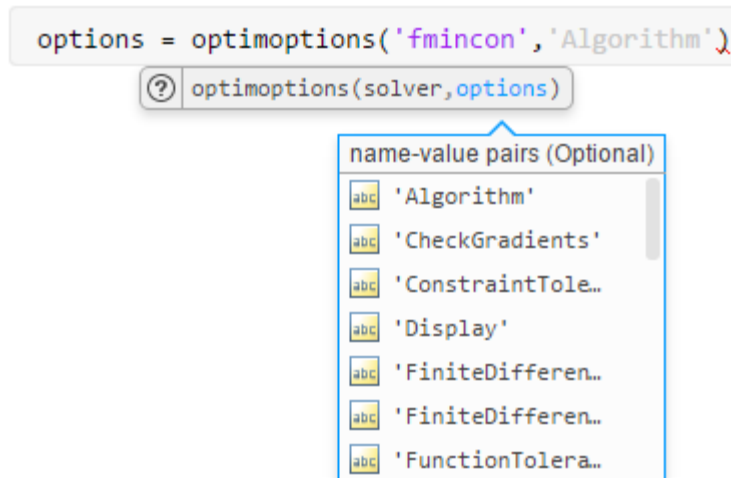
Select a name-value pair in one of these ways:

- Click the name-value pair.
- Use the arrow keys to highlight the name-value pair, and then press **Tab**.
- Type the first letters of the name-value pair, and then press **Tab**.

4 Enter the appropriate value for the selected name. If the value is from a list of choices, you can select it the same way that you selected the name.

5 Continue adding name-value pairs until the options are complete.

6 Be sure to pass the options to your solver.

```
[x,fval,exitflag,output] = ...
    fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nlcon,options)
```

Tip

- For help choosing a solver, see "Optimization Decision Table" on page 2-4.

- For help choosing the solver algorithm, see "Choosing the Algorithm" on page 2-6.
- To understand the meaning of other options, see "Set Options".

Set Options: Command Line or Standard Scripts

Beginning with R2018a, the MATLAB command line and the standard Editor show suggestions for `optimoptions` names and values.

1 Set options by entering `options = optimoptions('` and pressing **Tab**. MATLAB shows a list of solvers.

```
>> options = optimoptions('
```
```
fgoalattain
fmincon
fminimax
fminunc
fseminf
fsolve
ga
gamultiobj
```
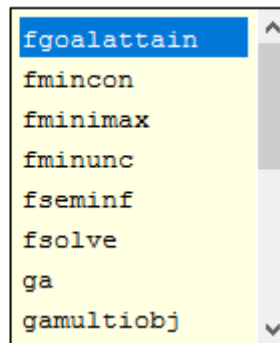
2 Select a solver in one of these ways:

- Double-click the solver.
- Use the arrow keys to highlight the solver, and then press **Tab**.
- Type the first letters of the solver, and then press **Tab**.

3 Enter `','` and then press **Tab**. MATLAB displays a list of name-value pairs for the solver.

```
>> options = optimoptions('fmincon','
```
```
Algorithm
CheckGradients
ConstraintTolerance
Display
FiniteDifferenceStepSize
FiniteDifferenceType
FunctionTolerance
HessianApproximation
```
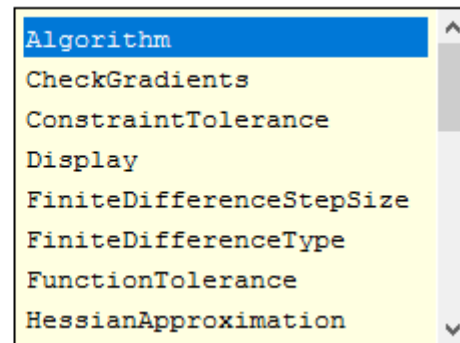
Select a name-value pair using one of the ways described in step 2.

4 Enter the appropriate value for the selected name. If the value is from a list of choices, you can select it the same way that you selected the name.

5 Continue adding name-value pairs until the options are complete.

6 Be sure to pass the options to your solver.

```
[x,fval,exitflag,output] = ...
    fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nlcon,options)
```

Tip

- For help choosing a solver, see "Optimization Decision Table" on page 2-4.
- For help choosing the solver algorithm, see "Choosing the Algorithm" on page 2-6.
- To understand the meaning of other options, see "Set Options".

Choose Plot Functions

To monitor your optimization while it runs, use a plot function. Solvers have a set of built-in plot functions. Use `optimoptions` to set the `'PlotFcn'` name-value pair to a built-in plot function, a cell array of built-in plot functions, or a function handle or cell array of function handles to plot functions.

Choose plot functions using live scripts:
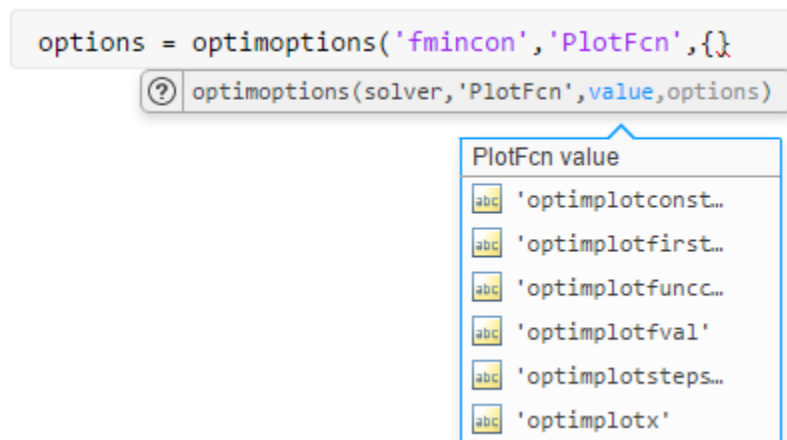
To choose plot functions using the Editor or the command line, enter `options = optimoptions('solvername','PlotFcn',{'` and then press **Tab**:
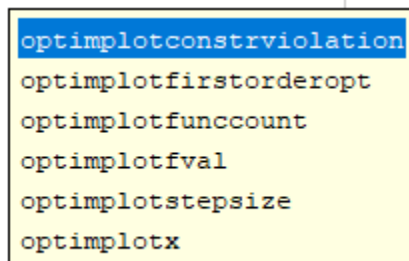
To choose a custom plot function, pass a function handle such as `@myplotfun`. Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27.

`linprog`, `lsqlin`, `quadprog`, and `lsqnonneg` do not support plot functions, because these solvers typically run quickly. To monitor their progress, you can use iterative display for `linprog`, the `lsqlin` `'interior-point'` algorithm, and the `quadprog` `'interior-point-convex'` algorithm. Set the `'Display'` option to `'iter'`.

The `fminbnd`, `fminsearch`, and `fzero` solvers do not use options created by `optimoptions`, only `optimset`. To see which plot functions these solvers use, consult their reference pages:

- `fminbnd` options
- `fminsearch` options
- `fzero` options

Pass Solver Arguments

Solvers use positional function arguments. For example, the syntax for `fmincon` arguments is

`fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)`

If you want to include only `fun`, `x0`, `lb`, and `options` arguments, then the appropriate syntax is

`fmincon(fun,x0,[],[],[],[],lb,[],[],options)`

Sometimes the inexperienced will, instead, write

`fmincon(fun,x0,lb,options)` `% This is incorrect!`

This call throws an error. In this incorrect command, `fmincon` interprets the `lb` argument as representing the A matrix, and the `options` argument as representing the b vector. The third argument always represents the A matrix, and the fourth argument always represents the b vector.

It can be difficult to keep track of positional arguments as you enter a command. The following are suggestions for obtaining the correct syntax.

- Use live scripts. As you enter a command, you see function hints that guide you to enter the correct argument in each position. Enter `[]` for unused arguments.
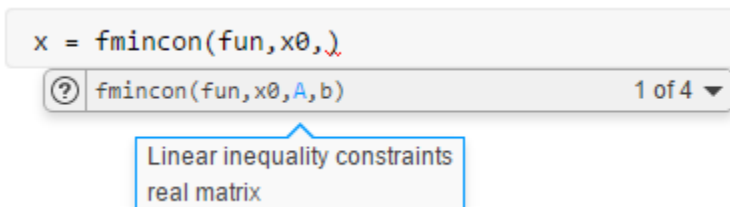
- Use the MATLAB Editor or command line. As you enter commands, you see lists of proper syntax that guide you to enter the correct argument in each position. Enter `[]` for unused arguments.
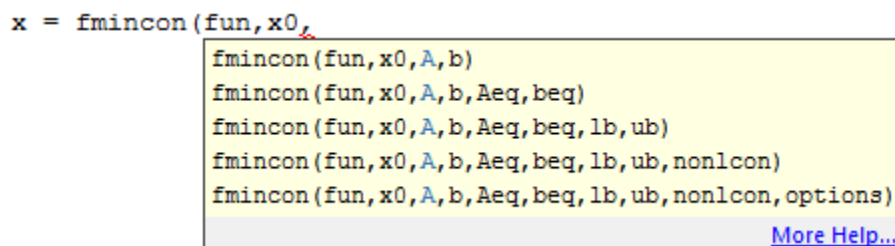
- Create a `problem` structure. This way, you can pass fewer arguments and pass named arguments instead of positional arguments. For `fmincon`, the `problem` structure requires at least the `objective`, `x0`, `solver`, and `options` fields. So, to give only the `fun`, `x0`, `lb`, and `options` arguments, create a `problem` structure as follows:

```
% These commands assume that fun, x0, lb, and opts exist
prob.objective = fun;
prob.x0 = x0;
prob.lb = lb;
prob.options = opts;
prob.solver = 'fmincon';
```

  You can also create a `problem` structure using one `struct` command.

```
% This command assumes that fun, x0, lb, and opts exist
prob = struct('objective',fun,'x0',x0,'lb',lb,...
    'options',opts,'solver','fmincon')
```

- If you have Global Optimization Toolbox, you can create a problem structure for `fmincon`, `fminunc`, `lsqcurvefit`, and `lsqnonlin` by using `createOptimProblem`.

See Also

More About

- "Solver-Based Optimization Problem Setup"
- "Set Options"
- "Run Sections in Live Scripts"
- "Code Sections"

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,

$$\min_s \{q(s),\ s \in N\}. \tag{6-1}$$

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

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

In the standard trust-region method ([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

$$\min\left\{\frac{1}{2}s^T H s + s^T g \ \text{ such that } \ \|Ds\| \le \Delta\right\}, \tag{6-2}$$

where g is the gradient of f at the current point x, H is the Hessian matrix (the symmetric matrix of second derivatives), D is a diagonal scaling matrix, Δ is a positive scalar, and $\| \, . \, \|$ is the 2-norm. Good

algorithms exist for solving "Equation 6-2" (see [48]); such algorithms typically involve the computation of all eigenvalues of H 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

$$H \cdot s_2 = -g, \tag{6-3}$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \tag{6-4}$$

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

A 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 it encounters a direction of negative (or zero) curvature, that is, $d^T H d \leq 0$. The PCG output direction p is either a direction of negative curvature or an approximate solution to the Newton system $Hp = -g$. In either case, p helps to 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

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 . \tag{6-5}$$

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.
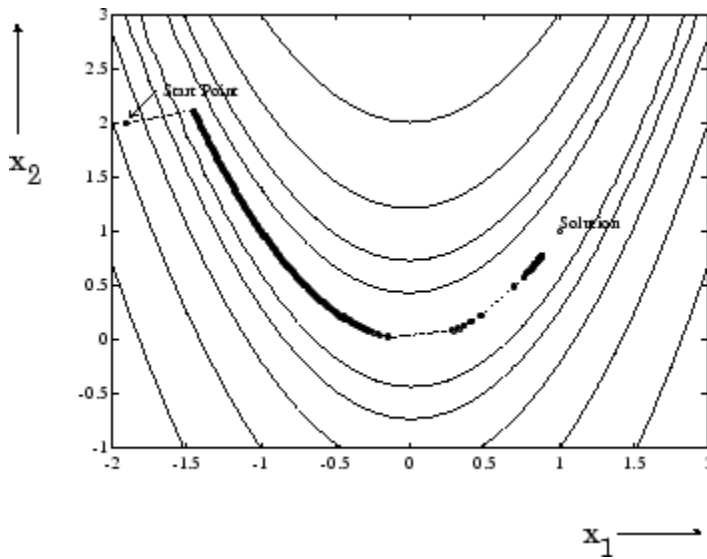
Figure 6-1, Steepest Descent Method on Rosenbrock's Function

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 a more complete description of this figure, including scripts that generate the iterative points, see "Banana Function Minimization" on page 6-49.

Quasi-Newton Methods

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

$$\min_{x} \frac{1}{2} x^T H x + c^T x + b, \tag{6-6}$$

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.,

$$\nabla f(x^*) = H x^* + c = 0. \tag{6-7}$$

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

$$x^* = -H^{-1} c. \tag{6-8}$$

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

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k s_k s_k^T H_k^T}{s_k^T H_k s_k},$$ (6-9)

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

$$d = -H_k^{-1} \cdot \nabla f(x_k).$$ (6-10)

The quasi-Newton method is illustrated by the solution path on Rosenbrock's function in "Figure 6-2, BFGS Method on Rosenbrock's Function" on page 6-6. 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.
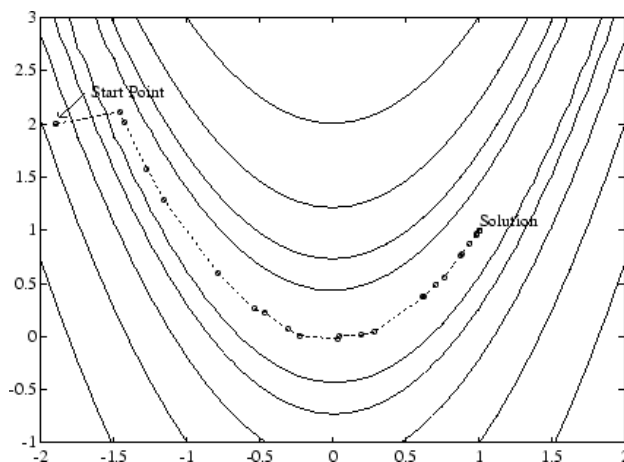
Figure 6-2, BFGS Method on Rosenbrock's Function

For a more complete description of this figure, including scripts that generate the iterative points, see "Banana Function Minimization" on page 6-49.

Line Search

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

$$x_{k+1} = x_k + \alpha^* d_k, \tag{6-11}$$

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

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

- The *bracketing* phase determines the range of points on the line $x_{k+1} = x_k + \alpha^* d_k$ to be searched. The *bracket* corresponds to an interval specifying the range of values of α.

- The *sectioning* step divides the bracket into subintervals, on which the minimum of the objective function is approximated by polynomial interpolation.

The resulting step length α satisfies the Wolfe conditions:

$$f(x_k + \alpha d_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T d_k, \tag{6-12}$$

$$\nabla f(x_k + \alpha d_k)^T d_k \geq c_2 \nabla f_k^T d_k, \tag{6-13}$$

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 6-5 (set `HessUpdate` to `'dfp'` in `options` to select the DFP method). The Hessian, H, is always maintained to be positive definite so that the direction of search, d, is always in a descent direction. This means that for some arbitrarily small step α in the direction d, the objective function decreases in magnitude. You achieve positive definiteness of H by ensuring that H is initialized to be positive definite and thereafter $q_k^T s_k$ (from "Equation 6-14") is always positive. The term $q_k^T s_k$ is a product of the line search step length parameter α_k and a combination of the search direction d with past and present gradient evaluations,

$$q_k^T s_k = \alpha_k \left(\nabla f(x_{k+1})^T d - \nabla f(x_k)^T d \right). \tag{6-14}$$

You always achieve the condition that $q_k^T s_k$ is positive by performing a sufficiently accurate line search. This is because the search direction, d, is a descent direction, so that α_k and the negative

gradient $-\nabla f(x_k)^T d$ are always positive. Thus, the possible negative term $-\nabla f(x_{k+1})^T d$ can be made as small in magnitude as required by increasing the accuracy of the line search.

See Also

`fminunc`

More About

- "fminsearch Algorithm" on page 6-9

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

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

$s = m + 2(m – x(n + 1)),$

and calculate $f(s)$.

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

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

6   If $f(r) \ge f(x(n))$, perform a contraction between m and either $x(n + 1)$ or r, depending on which has the lower objective function value.

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

    $c = m + (r – m)/2$

    and calculate $f(c)$. If $f(c) < f(r)$, accept c and terminate the iteration. **Contract outside**

    Otherwise, continue with Step 7 (Shrink).

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

    $cc = m + (x(n + 1) – m)/2$

    and calculate $f(cc)$. If $f(cc) < f(x(n + 1))$, accept cc and terminate the iteration. **Contract inside**

Otherwise, continue with Step 7 (Shrink).

7 Calculate the *n* points

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

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

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.
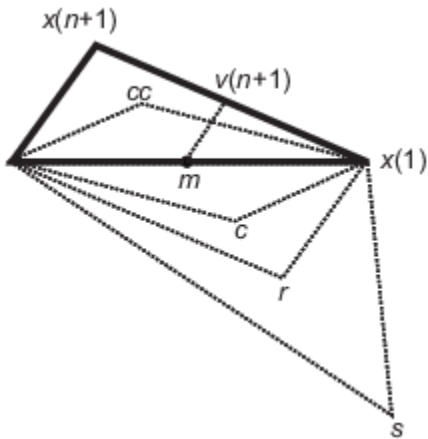
See Also
`fminsearch`

More About
- "Unconstrained Nonlinear Optimization Algorithms" on page 6-2

Unconstrained Minimization Using fminunc

This example shows how to use `fminunc` to solve the nonlinear minimization problem

$$\min_{x} f(x) = e^{x1}\left(4x_1^2 + 2x_2^2 + 4x_1 x_2 + 2x_2 + 1\right).$$

To solve this two-dimensional problem, write a function that returns $f(x)$. Then, invoke the unconstrained minimization routine `fminunc` starting from the initial point `x0 = [-1,1]`.

The helper function `objfun` at the end of this example on page 6-0   calculates $f(x)$.

To find the minimum of $f(x)$, set the initial point and call `fminunc`.

```
x0 = [-1,1];
[x,fval,exitflag,output] = fminunc(@objfun,x0);
```

```
Local minimum found.
```

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

View the results, including the first-order optimality measure in the `output` structure.

```
disp(x)
```

```
    0.5000   -1.0000
```

```
disp(fval)
```

```
   3.6609e-15
```

```
disp(exitflag)
```

```
     1
```

```
disp(output.firstorderopt)
```

```
   1.2284e-07
```

The `exitflag` output indicates whether the algorithm converges. `exitflag = 1` means `fminunc` finds a local minimum.

The `output` structure gives more details about the optimization. For `fminunc`, the structure includes

- `output.iterations`, the number of iterations
- `output.funcCount`, the number of function evaluations
- `output.stepsize`, the final step-size
- `output.firstorderopt`, a measure of first-order optimality (which in this unconstrained case is the infinity norm of the gradient at the solution)
- `output.algorithm`, the type of algorithm used
- `output.message`, the reason the algorithm stopped

Helper Function

This code creates the `objfun` helper 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);
end
```

See Also

Related Examples

*   "Minimization with Gradient and Hessian" on page 6-13

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 $n = 1000$.

The helper function `brownfgh` at the end of this example on page 6-0 calculates $f(x)$, its gradient $g(x)$, and its Hessian $H(x)$. To specify that the `fminunc` solver use the derivative information, set the `SpecifyObjectiveGradient` and `HessianFcn` options using `optimoptions`. To use a Hessian with `fminunc`, you must use the `'trust-region'` algorithm.

```
options = optimoptions(@fminunc,'Algorithm','trust-region',...
    'SpecifyObjectiveGradient',true,'HessianFcn','objective');
```

Set the parameter `n` to 1000, and set the initial point `xstart` to –1 for odd components and +1 for even components.

```
n = 1000;
xstart = -ones(n,1);
xstart(2:2:n) = 1;
```

Find the minimum value of f.

```
[x,fval,exitflag,output] = fminunc(@brownfgh,xstart,options);
```

```
Local minimum found.

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

Examine the solution and solution process.

```
disp(fval)
```

```
    2.8709e-17
```

```
disp(exitflag)
```

```
     1
```

```
disp(output)
```

```
         iterations: 7
          funcCount: 8
           stepsize: 0.0039
       cgiterations: 7
       firstorderopt: 4.7948e-10
          algorithm: 'trust-region'
            message: '...'
      constrviolation: []
```

The function $f(x)$ is a sum of powers of squares, so is nonnegative. The solution `fval` is nearly zero, so is clearly a minimum. The exit flag 1 also indicates that `fminunc` found a solution. The `output` structure shows that `fminunc` took just seven iterations to reach the solution.

Display the largest and smallest elements of the solution.

```
disp(max(x))
```

```
   1.1987e-10
```

```
disp(min(x))
```

```
  -1.1987e-10
```

The solution is very near the point where all elements of $x = 0$.

Helper Function

This code creates the `brownfgh` helper function.

```matlab
function [f,g,H] = brownfgh(x)
%BROWNFGH  Nonlinear minimization problem (function, its gradients
% and Hessian)
% Documentation example

%   Copyright 1990-2008 The MathWorks, Inc.

% 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
     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
%
% Evaluate the (sparse, symmetric) Hessian matrix
  if nargout > 2
     v=zeros(n,1);
     i=1:(n-1);
     v(i)=2*(x(i+1).^2+1).*((x(i).^2).^(x(i+1).^2))+...
             4*(x(i+1).^2+1).*(x(i+1).^2).*(x(i).^2).*((x(i).^2).^((x(i+1).^2)-1))+...
             2*((x(i+1).^2).^(x(i).^2+1)).*(log(x(i+1).^2));
     v(i)=v(i)+4*(x(i).^2).*((x(i+1).^2).^(x(i).^2+1)).*((log(x(i+1).^2)).^2);
     v(i+1)=v(i+1)+...
             2*(x(i).^2).^(x(i+1).^2+1).*(log(x(i).^2))+...
             4*(x(i+1).^2).*((x(i).^2).^(x(i+1).^2+1)).*((log(x(i).^2)).^2)+...
             2*(x(i).^2+1).*((x(i+1).^2).^(x(i).^2));
     v(i+1)=v(i+1)+4*(x(i).^2+1).*(x(i+1).^2).*(x(i).^2).*((x(i+1).^2).^(x(i).^2-1));
     v0=v;
     v=zeros(n-1,1);
     v(i)=4*x(i+1).*x(i).*((x(i).^2).^(x(i+1).^2))+...
             4*x(i+1).*(x(i+1).^2+1).*x(i).*((x(i).^2).^(x(i+1).^2)).*log(x(i).^2);
     v(i)=v(i)+ 4*x(i+1).*x(i).*((x(i+1).^2).^(x(i).^2)).*log(x(i+1).^2);
     v(i)=v(i)+4*x(i).*((x(i+1).^2).^(x(i).^2)).*x(i+1);
     v1=v;
```

```
    i=[(1:n)';(1:(n-1))'];
    j=[(1:n)';(2:n)'];
    s=[v0;2*v1];
    H=sparse(i,j,s,n,n);
    H=(H+H')/2;
  end
end
```

See Also

Related Examples

- "Minimization with Gradient and Hessian Sparsity Pattern" on page 6-16

Minimization with Gradient and Hessian Sparsity Pattern

This example shows how to solve a nonlinear minimization problem with a 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 $n = 1000$.

```
n = 1000;
```

To use the `trust-region` method in `fminunc`, you must compute the gradient in the objective function; it is not optional as in the `quasi-newton` method.

The helper function `brownfg` at the end of this example on page 6-0   computes the objective function and gradient.

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, the structure `Hstr`, a sparse matrix, is available in the file `brownhstr.mat`. Using the `spy` command, you can see that `Hstr` is, indeed, sparse (only 2998 nonzeros).

```
load brownhstr
spy(Hstr)
```
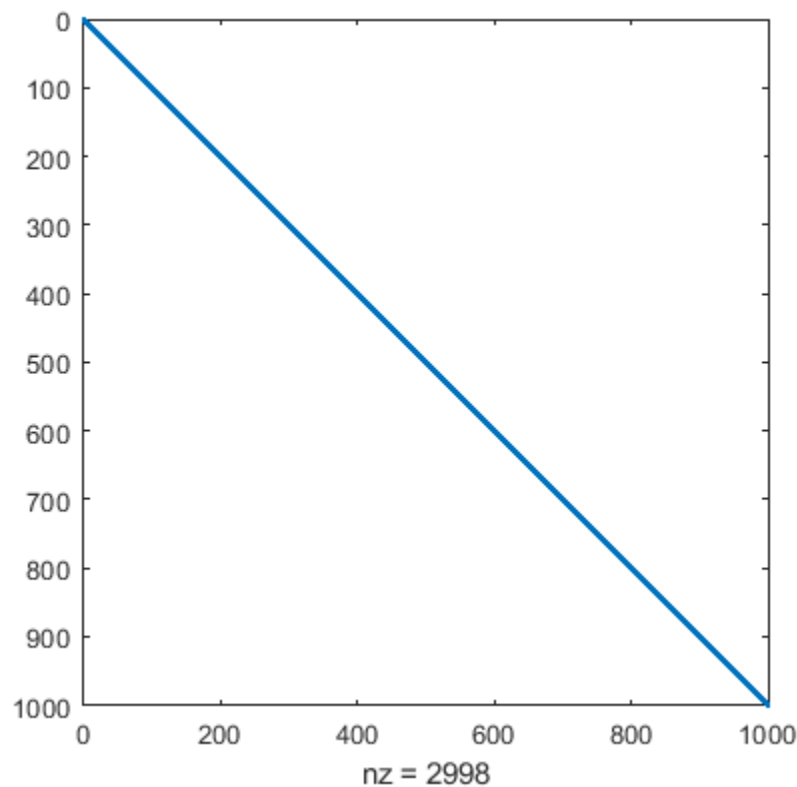
Set the `HessPattern` option to `Hstr` using `optimoptions`. When such a large problem has obvious sparsity structure, not setting the `HessPattern` option uses a great amount of memory and computation unnecessarily, because `fminunc` attempts to use finite differencing on a full Hessian matrix of one million nonzero entries.

To use the Hessian sparsity pattern, you must use the `trust-region` algorithm of `fminunc`. This algorithm also requires you to set the `SpecifyObjectiveGradient` option to `true` using `optimoptions`.

```
options = optimoptions(@fminunc,'Algorithm','trust-region',...
    'SpecifyObjectiveGradient',true,'HessPattern',Hstr);
```

Set the objective function to `@brownfg`. Set the initial point to –1 for odd *x* components and +1 for even *x* components.

```
xstart = -ones(n,1);
xstart(2:2:n,1) = 1;
fun = @brownfg;
```

Solve the problem by calling `fminunc` using the initial point `xstart` and options `options`.

```
[x,fval,exitflag,output] = fminunc(fun,xstart,options);
```

```
Local minimum found.

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

Examine the solution and solution process.

```
disp(fval)

    7.4738e-17
```

```
disp(exitflag)

     1
```

```
disp(output)

         iterations: 7
          funcCount: 8
           stepsize: 0.0046
       cgiterations: 7
       firstorderopt: 7.9822e-10
          algorithm: 'trust-region'
            message: '...'
      constrviolation: []
```

The function $f(x)$ is a sum of powers of squares and, therefore, is nonnegative. The solution `fval` is nearly zero, so it is clearly a minimum. The exit flag `1` also indicates that `fminunc` finds a solution. The `output` structure shows that `fminunc` takes only seven iterations to reach the solution.

Display the largest and smallest elements of the solution.

```
disp(max(x))

    1.9955e-10
```

```
disp(min(x))
```

```
   -1.9955e-10
```

The solution is near the point where all elements of `x = 0`.

Helper Function

This code creates the `brownfg` helper function.

```
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
end
```

See Also

Related Examples

- "Minimization with Gradient and Hessian" on page 6-13

Constrained Nonlinear Optimization Algorithms

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: $c(x) \leq 0$, $ceq(x) = 0$, $A \cdot x \leq b$, $Aeq \cdot x = beq$, $l \leq x \leq u$. There are even more constraints used in semi-infinite programming; see "fseminf Problem Formulation and Algorithm" on page 6-32.

fmincon Trust Region Reflective Algorithm

Trust-Region Methods for Nonlinear Minimization

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

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

$$\min_s \{q(s), \ s \in N\} . \tag{6-15}$$

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

$$\min\left\{\frac{1}{2}s^THs + s^Tg \ \text{ such that } \ \|Ds\| \le \Delta\right\}, \tag{6-16}$$

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 6-16" (see [48]); such algorithms typically involve the computation of all eigenvalues of H and a Newton process applied to the secular equation

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

Such algorithms provide an accurate solution to "Equation 6-16". 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 6-16", 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 6-16" is trivial even if full eigenvalue/eigenvector information is needed (since in the subspace, the problem is only two-dimensional). The dominant work has now shifted to the determination of the subspace.

The two-dimensional subspace S is determined with the aid of a preconditioned conjugate gradient process described below. The solver defines S as the linear space spanned by s_1 and s_2, where s_1 is in the direction of the gradient g, and s_2 is either an approximate Newton direction, i.e., a solution to

$$H \cdot s_2 = -g, \tag{6-17}$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \tag{6-18}$$

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 6-16" 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., $f(x + s) \ge f(x)$. 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 it encounters a direction of negative (or zero) curvature, that is, $d^T H d \leq 0$. The PCG output direction p is either a direction of negative curvature or an approximate solution to the Newton system $Hp = -g$. In either case, p helps to 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

$$\min\{f(x) \text{ such that } Ax = b\}, \tag{6-19}$$

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 6-19" differs from the unconstrained approach in two significant ways. First, an initial feasible point x_0 is computed, using a sparse least-squares step, so that $Ax_0 = b$. Second, Algorithm PCG is replaced with Reduced Preconditioned Conjugate Gradients (RPCG), see [46], in order to compute an approximate reduced Newton step (or a direction of negative curvature in the null space of A). The key linear algebra step involves solving systems of the form

$$\begin{bmatrix} C & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \tag{6-20}$$

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

$$\min\{f(x) \text{ such that } l \leq x \leq u\}, \tag{6-21}$$

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 6-21",

$$(D(x))^{-2}g = 0,$$ (6-22)

where

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

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 6-22" 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 6-22" is defined as the solution to the linear system

$$\widehat{M}Ds^N = -\widehat{g}$$ (6-23)

at the kth iteration, where

$$\widehat{g} = D^{-1}g = \mathrm{diag}\left(|v|^{1/2}\right)g,$$ (6-24)

and

$$\widehat{M} = D^{-1}HD^{-1} + \mathrm{diag}(g)J^v.$$ (6-25)

Here J^v plays the role of the Jacobian of $|v|$. Each diagonal component of the diagonal matrix J^v equals 0, –1, or 1. If all the components of l and u are finite, $J^v = \mathrm{diag}(\mathrm{sign}(g))$. At a point where $g_i = 0$, v_i might not be differentiable. $J^v_{ii} = 0$ is defined at such a point. Nondifferentiability of this type is not a cause for concern because, for such a component, it is not significant which value v_i takes. Further, $|v_i|$ will still be discontinuous at this point, but the function $|v_i| \cdot g_i$ is continuous.

Second, reflections are used to increase the 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 parametrized 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

$$
\nabla f(x^*) + \sum_{i=1}^{m} \lambda_i \cdot \nabla G_i(x^*) = 0
$$

$$
\lambda_i \cdot G_i(x^*) = 0, \quad i = 1, ..., m_e
$$

$$
\lambda_i \geq 0, \quad i = m_e + 1, ..., m,
$$

(6-26)

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 2-10.

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.

$$L(x, \lambda) = f(x) + \sum_{i=1}^{m} \lambda_i \cdot g_i(x) \,. \qquad (6\text{-}27)$$

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

$$\min_{d \in \mathfrak{R}^n} \frac{1}{2} d^T H_k d + \nabla f(x_k)^T d$$

$$\nabla g_i(x_k)^T d + g_i(x_k) = 0, \quad i = 1, \dots, m_e \qquad (6\text{-}28)$$

$$\nabla g_i(x_k)^T d + g_i(x_k) \le 0, \quad i = m_e + 1, \dots, m \,.$$

This subproblem can be solved using any QP algorithm (see, for instance, "Quadratic Programming Solution" on page 6-26). The solution is used to form a new iterate

$$x_{k+1} = x_k + \alpha_k d_k.$$

The step length parameter α_k is determined by an appropriate line search procedure so that a sufficient decrease in a merit function is obtained (see "Updating the Hessian Matrix" on page 6-25). The matrix H_k is a positive definite approximation of the Hessian matrix of the Lagrangian function ("Equation 6-27"). H_k can be updated by any of the quasi-Newton methods, although the BFGS method (see "Updating the Hessian Matrix" on page 6-25) 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, $g(x)$,

$$x_1^2 + x_2^2 - 1.5 \le 0. \qquad (6\text{-}29)$$

This was solved by an SQP implementation in 96 iterations compared to 140 for the unconstrained case. "Figure 6-3, SQP Method on Nonlinearly Constrained Rosenbrock's Function" on page 6-24 shows the path to the solution point $x = [0.9072, 0.8228]$ starting at $x = [-1.9, 2.0]$.
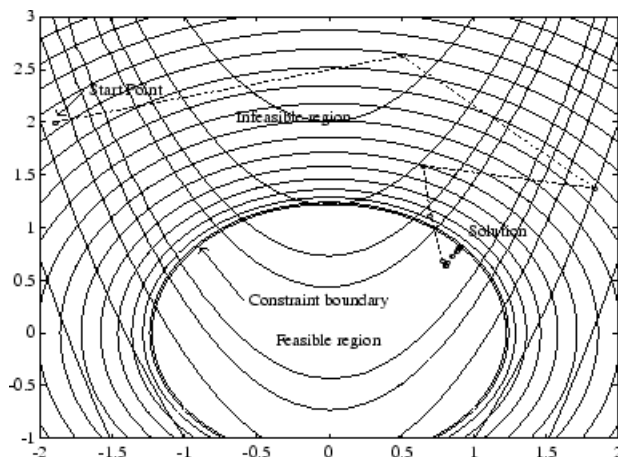
Figure 6-3, SQP Method on Nonlinearly Constrained Rosenbrock's Function

SQP Implementation

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

- "Updating the Hessian Matrix" on page 6-25
- "Quadratic Programming Solution" on page 6-26
- "Initialization" on page 6-28
- "Line Search and Merit Function" on page 6-28

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.

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k s_k s_k^T H_k^T}{s_k^T H_k s_k},\tag{6-30}$$

where

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

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

Powell [33] recommends keeping the Hessian positive definite even though it might be positive indefinite at the solution point. A positive definite Hessian is maintained providing $q_k^T s_k$ is positive at each update and that H is initialized with a positive definite matrix. When $q_k^T s_k$ is not positive, q_k is modified on an element-by-element basis so that $q_k^T s_k > 0$. The general aim of this modification is to distort the elements of q_k, which contribute to a positive definite update, as little as possible. Therefore, in the initial phase of the modification, the most negative element of $q_k * s_k$ is repeatedly halved. This procedure is continued until $q_k^T s_k$ is greater than or equal to a small negative tolerance. If, after this procedure, $q_k^T s_k$ is still not positive, modify q_k by adding a vector v multiplied by a constant scalar w, that is,

$$q_k = q_k + wv,\tag{6-31}$$

where

$$v_i = \nabla g_i(x_{k+1}) \cdot g_i(x_{k+1}) - \nabla g_i(x_k) \cdot g_i(x_k)$$
$$\text{if } (q_k)_i \cdot w < 0 \text{ and } (q_k)_i \cdot (s_k)_i < 0, \ i = 1, ..., m$$
$$v_i = 0 \ \text{ otherwise,}$$

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.

$$\min_{d \in \Re^n} q(d) = \frac{1}{2}d^T H d + c^T d,$$
$$A_i d = b_i, \quad i = 1, ..., m_e$$
$$A_i d \le b_i, \quad i = m_e + 1, ..., m. \tag{6-32}$$

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

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

$$Z_k = Q[:, l+1:m], \tag{6-33}$$

where

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

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

Then if you view the quadratic as a function of p, by substituting for $\widehat{d}_k$, you have

$$q(p) = \frac{1}{2}p^T Z_k^T H Z_k p + c^T Z_k p. \tag{6-34}$$

Differentiating this with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c. \tag{6-35}$$

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

$$Z_k^T H Z_k p = -Z_k^T c. \tag{6-36}$$

A step is then taken of the form

$$x_{k+1} = x_k + \alpha \widehat{d}_k, \quad \text{where } \widehat{d}_k = Z_k p. \tag{6-37}$$

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 $\widehat{d}_k$ 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 6-32"). Otherwise, the step along $\widehat{d}_k$ 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 $\widehat{d}_k$ is given by

$$\alpha = \min_{i \in \{1, \ldots, m\}} \left\{ \frac{-(A_i x_k - b_i)}{A_i \widehat{d}_k} \right\}, \tag{6-38}$$

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

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

$$\bar{A}_k^T \lambda_k = c + H x_k. \tag{6-39}$$

If all elements of λ_k are positive, x_k is the optimal solution of QP ("Equation 6-32"). However, if any component of λ_k is negative, and the component does not correspond to an equality constraint, then the corresponding element is deleted from the active set and a new iterate is sought.

Initialization

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

$$
\begin{array}{ll}
\min\limits_{\gamma \in \Re, \, x \in \Re^n} & \gamma \text{ such that} \\
\end{array}
$$
$$
A_i x = b_i, \quad i = 1, ..., m_e
$$
$$
A_i x - \gamma \le b_i, \quad i = m_e + 1, ..., m.
$$

(6-40)

The notation A_i indicates the ith row of the matrix A. You can find a feasible point (if one exists) to "Equation 6-40" 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).

$$
\widehat{d}_k = -Z_k Z_k^T g_k .
$$

(6-41)

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

$$
H\widehat{d}_1 = -g_k,
$$

(6-42)

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 $\widehat{d}_k$ 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

$$
x_{k+1} = x_k + \alpha d_k .
$$

(6-43)

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.

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

(6-44)

Powell recommends setting the penalty parameter

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

(6-45)

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

$$r_i = \frac{\|\nabla f(x)\|}{\|\nabla g_i(x)\|},$$ (6-46)

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 (and nearly identical `sqp-legacy` algorithm) is similar to the `active-set` algorithm (for a description, see "fmincon Active Set Algorithm" on page 6-22). The basic `sqp` algorithm is described in Chapter 18 of Nocedal and Wright [31].

The `sqp` algorithm is essentially the same as the `sqp-legacy` algorithm, but has a different implementation. Usually, `sqp` has faster execution time and less memory usage than `sqp-legacy`.

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 6-28". 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 6-28" 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 6-28" 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 6-44" 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

$$\min_{x} f(x), \text{ subject to } h(x) = 0 \text{ and } g(x) \leq 0. \tag{6-47}$$

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

$$\min_{x,s} f_\mu(x,s) = \min_{x,s} f(x) - \mu \sum_i \ln(s_i), \text{ subject to } h(x) = 0 \text{ and } g(x) + s = 0. \tag{6-48}$$

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 6-48" is a sequence of equality constrained problems. These are easier to solve than the original inequality-constrained problem "Equation 6-47".

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

$$f_\mu(x,s) + \nu \|(h(x), g(x) + s)\|.$$

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

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:

$$H = \nabla^2 f(x) + \sum_i \lambda_i \nabla^2 g_i(x) + \sum_j \lambda_j \nabla^2 h_j(x).$$
(6-49)

- 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 6-50" defines the direct step $(\Delta x, \Delta s)$:

$$\begin{bmatrix} H & 0 & J_h^T & J_g^T \\ 0 & S\Lambda & 0 & -S \\ J_h & 0 & I & 0 \\ J_g & -S & 0 & I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \\ -\Delta y \\ -\Delta \lambda \end{bmatrix} = - \begin{bmatrix} \nabla f - J_h^T y - J_g^T \lambda \\ S\lambda - \mu e \\ h \\ g + s \end{bmatrix}.$$
(6-50)

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 6-48" 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 6-31. The algorithm obtains Lagrange multipliers by approximately solving the KKT equations

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

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

$$\min_{\Delta x, \Delta s} \nabla f^T \Delta x + \frac{1}{2}\Delta x^T \nabla_{xx}^2 L \Delta x + \mu e^T S^{-1} \Delta s + \frac{1}{2}\Delta s^T S^{-1} \Lambda \Delta s, \qquad (6\text{-}51)$$

subject to the linearized constraints

$$g(x) + J_g \Delta x + \Delta s = 0, \quad h(x) + J_h \Delta x = 0. \qquad (6\text{-}52)$$

To solve "Equation 6-52", the algorithm tries to minimize a norm of the linearized constraints inside a region with radius scaled by R. Then "Equation 6-51" is solved with the constraints being to match the residual from solving "Equation 6-52", 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 6-21.

Interior-Point Algorithm Options

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

- `HonorBounds` — When set to `true`, every iterate satisfies the bound constraints you have set. When set to `false`, the algorithm may violate bounds during intermediate iterations.
- `HessianApproximation` — When set to:
  - `'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.
- `HessianFcn` — fmincon uses the function handle you specify in `HessianFcn` to compute the Hessian. See "Including Hessians" on page 2-21.
- `HessianMultiplyFcn` — Give a separate function for Hessian-times-vector evaluation. For details, see "Including Hessians" on page 2-21 and "Hessian Multiply Function" on page 2-23.
- `SubproblemAlgorithm` — Determines whether or not to attempt the direct Newton step. The default setting `'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 $c(x) \leq 0$, $ceq(x) = 0$, $A \cdot x \leq b$, $Aeq \cdot x = beq$, 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

$$\max_{w_j \in I_j} K_j(x, w_j) \leq 0 \text{ for all } j = 1, ..., |K|, \tag{6-53}$$

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 6-53". 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 $c(x) \leq 0$, $ceq(x) = 0$, $A \cdot x \leq b$, $Aeq \cdot x = beq$, 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.

Tutorial for Optimization Toolbox™

This tutorial includes multiple examples that show how to use two nonlinear optimization solvers, `fminunc` and `fmincon`, and how to set options. The principles outlined in this tutorial apply to the other nonlinear solvers, such as `fgoalattain`, `fminimax`, `lsqnonlin`, `lsqcurvefit`, and `fsolve`.

The tutorial examples cover these tasks:

- Minimizing an objective function
- Minimizing the same function with additional parameters
- Minimizing the objective function with a constraint
- Obtaining a more efficient or accurate solution by providing gradients or a Hessian, or by changing options

Unconstrained Optimization Example

Consider the problem of finding a minimum of the function

$$x \exp(-(x^2 + y^2)) + (x^2 + y^2)/20 \,.$$

Plot the function to see where it is minimized.

```
f = @(x,y) x.*exp(-x.^2-y.^2)+(x.^2+y.^2)/20;
fsurf(f,[-2,2],'ShowContours','on')
```
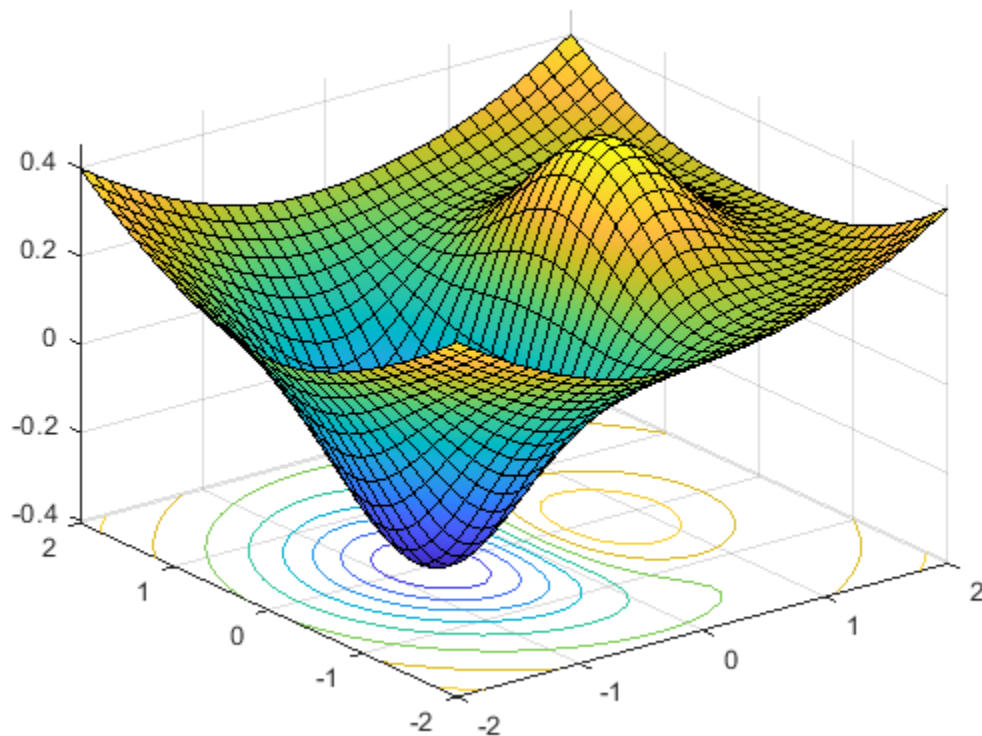
The plot shows that the minimum is near the point (–1/2,0).

Usually you define the objective function as a MATLAB® file. In this case, the function is simple enough to define as an anonymous function.

```
fun = @(x) f(x(1),x(2));
```

Set an initial point for finding the solution.

```
x0 = [-.5; 0];
```

Set optimization options to use the `fminunc` default `'quasi-newton'` algorithm. This step ensures that the tutorial works the same in every MATLAB version.

```
options = optimoptions('fminunc','Algorithm','quasi-newton');
```

View the iterations as the solver performs its calculations.

```
options.Display = 'iter';
```

Call `fminunc`, an unconstrained nonlinear minimizer.

```
[x, fval, exitflag, output] = fminunc(fun,x0,options);
```

```
                                                   First-order
 Iteration  Func-count        f(x)        Step-size    optimality
     0          3           -0.3769                      0.339
     1          6           -0.379694         1          0.286
     2          9           -0.405023         1          0.0284
     3         12           -0.405233         1          0.00386
     4         15           -0.405237         1          3.17e-05
     5         18           -0.405237         1          3.35e-08

Local minimum found.

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

Display the solution found by the solver.

```
uncx = x
```

```
uncx = 2×1

   -0.6691
    0.0000
```

View the function value at the solution.

```
uncf = fval
```

```
uncf = -0.4052
```

The examples use the number of function evaluations as a measure of efficiency. View the total number of function evaluations.

```
output.funcCount
```

```
ans = 18
```

Unconstrained Optimization Example with Additional Parameters

Next, pass extra parameters as additional arguments to the objective function, first by using a MATLAB file, and then by using a nested function.

Consider the objective function from the previous example.

$$f(x, y) = x\exp(-(x^2 + y^2)) + (x^2 + y^2)/20\,.$$

Parameterize the function with (a,b,c) as follows:

$$f(x, y, a, b, c) = (x - a)\exp(-((x - a)^2 + (y - b)^2)) + ((x - a)^2 + (y - b)^2)/c\,.$$

This function is a shifted and scaled version of the original objective function.

MATLAB File Function

Consider a MATLAB file objective function named `bowlpeakfun` defined as follows.

```
type bowlpeakfun
```

```
function y = bowlpeakfun(x, a, b, c)
%BOWLPEAKFUN Objective function for parameter passing in TUTDEMO.

%   Copyright 2008 The MathWorks, Inc.

y = (x(1)-a).*exp(-((x(1)-a).^2+(x(2)-b).^2))+((x(1)-a).^2+(x(2)-b).^2)/c;
```

Define the parameters.

```
a = 2;
b = 3;
c = 10;
```

Create an anonymous function handle to the MATLAB file.

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

```
f = function_handle with value:
    @(x)bowlpeakfun(x,a,b,c)
```

Call `fminunc` to find the minimum.

```
x0 = [-.5; 0];
options = optimoptions('fminunc','Algorithm','quasi-newton');
[x, fval] = fminunc(f,x0,options)
```

```
Local minimum found.

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

x = 2×1

    1.3639
    3.0000
```

```
fval = -0.3840
```

Nested Function

Consider the `nestedbowlpeak` function, which implements the objective as a nested function.

```
type nestedbowlpeak
```

```
function [x,fval] = nestedbowlpeak(a,b,c,x0,options)
%NESTEDBOWLPEAK Nested function for parameter passing in TUTDEMO.

%   Copyright 2008 The MathWorks, Inc.

[x,fval] = fminunc(@nestedfun,x0,options);
    function y = nestedfun(x)
      y = (x(1)-a).*exp(-((x(1)-a).^2+(x(2)-b).^2))+((x(1)-a).^2+(x(2)-b).^2)/c;
    end
end
```

The parameters (a,b,c) are visible to the nested objective function `nestedfun`. The outer function, `nestedbowlpeak`, calls `fminunc` and passes the objective function, `nestedfun`.

Define the parameters, initial guess, and options:

```
a = 2;
b = 3;
c = 10;
x0 = [-.5; 0];
options = optimoptions('fminunc','Algorithm','quasi-newton');
```

Run the optimization:

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

```
Local minimum found.

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

x = 2×1

    1.3639
    3.0000


fval = -0.3840
```

Both approaches produce the same answers, so you can use the one you find most convenient.

Constrained Optimization Example: Inequalities

Consider the previous problem with a constraint:

$$\text{minimize } x\exp(-(x^2 + y^2)) + (x^2 + y^2)/20,$$

$$\text{subject to } xy/2 + (x + 2)^2 + (y - 2)^2/2 \le 2.$$

The constraint set is the interior of a tilted ellipse. View the contours of the objective function plotted together with the tilted ellipse.

```matlab
f = @(x,y) x.*exp(-x.^2-y.^2)+(x.^2+y.^2)/20;
g = @(x,y) x.*y/2+(x+2).^2+(y-2).^2/2-2;
fimplicit(g)
axis([-6 0 -1 7])
hold on
fcontour(f)
plot(-.9727,.4685,'ro');
legend('constraint','f contours','minimum');
hold off
```
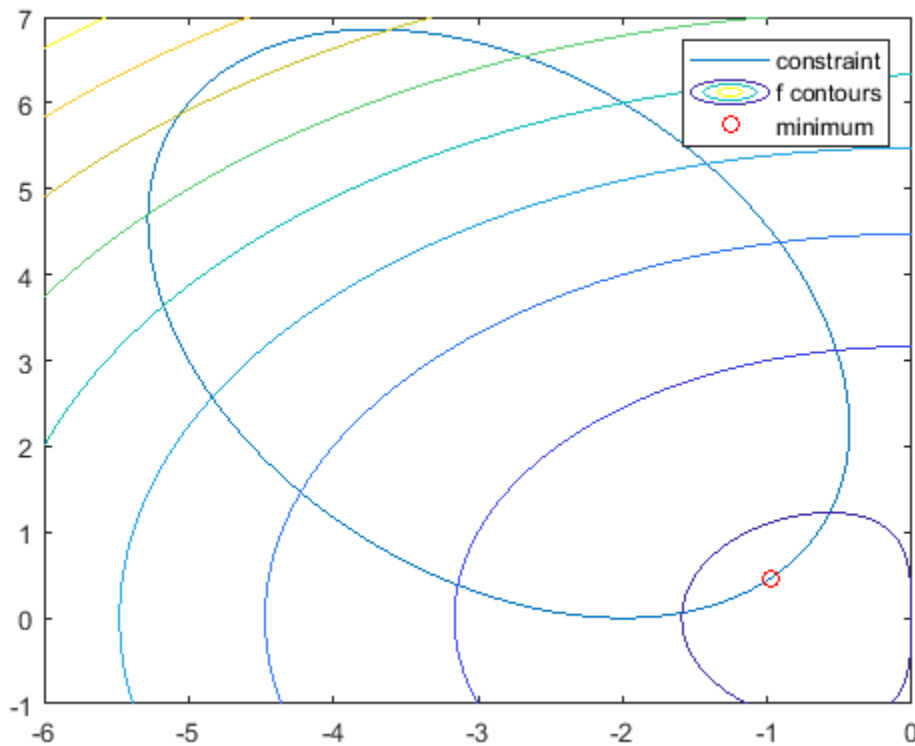

The plot shows that the lowest value of the objective function within the ellipse occurs near the lower-right part of the ellipse. Before calculating the plotted minimum, make a guess at the solution.

```matlab
x0 = [-2 1];
```

Set optimization options to use the interior-point algorithm and display the results at each iteration.

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

Solvers require that nonlinear constraint functions give two outputs, one for nonlinear inequalities and one for nonlinear equalities. To give both outputs, write the constraint using the deal function.

```matlab
gfun = @(x) deal(g(x(1),x(2)),[]);
```

Call the nonlinear constrained solver. The problem has no linear equalities or inequalities or bounds, so pass [ ] for those arguments.

```matlab
[x,fval,exitflag,output] = fmincon(fun,x0,[],[],[],[],[],[],gfun,options);
```

```
                                         First-order       Norm of
    Iter F-count            f(x)   Feasibility    optimality          step
       0        3     2.365241e-01    0.000e+00     1.972e-01
       1        6     1.748504e-01    0.000e+00     1.734e-01     2.260e-01
       2       10    -1.570560e-01    0.000e+00     2.608e-01     9.347e-01
       3       14    -6.629160e-02    0.000e+00     1.241e-01     3.103e-01
       4       17    -1.584082e-01    0.000e+00     7.934e-02     1.826e-01
       5       20    -2.349124e-01    0.000e+00     1.912e-02     1.571e-01
       6       23    -2.255299e-01    0.000e+00     1.955e-02     1.993e-02
       7       26    -2.444225e-01    0.000e+00     4.293e-03     3.821e-02
       8       29    -2.446931e-01    0.000e+00     8.100e-04     4.035e-03
       9       32    -2.446933e-01    0.000e+00     1.999e-04     8.126e-04
      10       35    -2.448531e-01    0.000e+00     4.004e-05     3.289e-04
      11       38    -2.448927e-01    0.000e+00     4.036e-07     8.156e-05
```

Local minimum found that satisfies the constraints.

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

Display the solution found by the solver.

x

x = *1×2*

```
   -0.9727     0.4686
```

View the function value at the solution.

fval

fval = -0.2449

View the total number of function evaluations.

Fevals = output.funcCount

Fevals = 38

The inequality constraint is satisfied at the solution.

[c, ceq] = gfun(x)

c = -2.4608e-06

ceq =

     []

Because c(x) is close to 0, the constraint is *active,* meaning it affects the solution. Recall the
unconstrained solution.

uncx

uncx = *2×1*

```
   -0.6691
```

```
     0.0000
```

Recall the unconstrained objective function.

```
uncf
```

```
uncf = -0.4052
```

See how much the constraint moved the solution and increased the objective.

```
fval-uncf
```

```
ans = 0.1603
```

Constrained Optimization Example: User-Supplied Gradients

You can solve optimization problems more efficiently and accurately by supplying gradients. This example, like the previous one, solves the inequality-constrained problem

$$\text{minimize } x\exp(-(x^2+y^2)) + (x^2+y^2)/20,$$

$$\text{subject to } xy/2 + (x+2)^2 + (y-2)^2/2 \le 2.$$

To provide the gradient of f(x) to `fmincon`, write the objective function in the form of a MATLAB file.

```
type onehump
```

```
function [f,gf] = onehump(x)
% ONEHUMP Helper function for Tutorial for the Optimization Toolbox demo

%   Copyright 2008-2009 The MathWorks, Inc.

r = x(1)^2 + x(2)^2;
s = exp(-r);
f = x(1)*s+r/20;

if nargout > 1
    gf = [(1-2*x(1)^2)*s+x(1)/10;
        -2*x(1)*x(2)*s+x(2)/10];
end
```

The constraint and its gradient are contained in the MATLAB file `tiltellipse`.

```
type tiltellipse
```

```
function [c,ceq,gc,gceq] = tiltellipse(x)
% TILTELLIPSE Helper function for Tutorial for the Optimization Toolbox demo

%   Copyright 2008-2009 The MathWorks, Inc.

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

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

Set an initial point for finding the solution.

```
x0 = [-2; 1];
```

Set optimization options to use the same algorithm as in the previous example for comparison purposes.

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

Set options to use the gradient information in the objective and constraint functions. Note: these options must be turned on or the gradient information will be ignored.

```
options = optimoptions(options,...
    'SpecifyObjectiveGradient',true,...
    'SpecifyConstraintGradient',true);
```

Because `fmincon` does not need to estimate gradients using finite differences, the solver should have fewer function counts. Set options to display the results at each iteration.

```
options.Display = 'iter';
```

Call the solver.

```
[x,fval,exitflag,output] = fmincon(@onehump,x0,[],[],[],[],[],[], ...
                                   @tiltellipse,options);
```

```
                                         First-order      Norm of
 Iter F-count            f(x)  Feasibility   optimality       step
    0       1   2.365241e-01   0.000e+00    1.972e-01
    1       2   1.748504e-01   0.000e+00    1.734e-01   2.260e-01
    2       4  -1.570560e-01   0.000e+00    2.608e-01   9.347e-01
    3       6  -6.629161e-02   0.000e+00    1.241e-01   3.103e-01
    4       7  -1.584082e-01   0.000e+00    7.934e-02   1.826e-01
    5       8  -2.349124e-01   0.000e+00    1.912e-02   1.571e-01
    6       9  -2.255299e-01   0.000e+00    1.955e-02   1.993e-02
    7      10  -2.444225e-01   0.000e+00    4.293e-03   3.821e-02
    8      11  -2.446931e-01   0.000e+00    8.100e-04   4.035e-03
    9      12  -2.446933e-01   0.000e+00    1.999e-04   8.126e-04
   10      13  -2.448531e-01   0.000e+00    4.004e-05   3.289e-04
   11      14  -2.448927e-01   0.000e+00    4.036e-07   8.156e-05

Local minimum found that satisfies the constraints.

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

`fmincon` estimated gradients well in the previous example, so the iterations in this example are similar.

Display the solution found by the solver.

```
xold = x
```

```
xold = 2×1

   -0.9727
    0.4686
```

View the function value at the solution.

```
minfval = fval
```

```
minfval = -0.2449
```

View the total number of function evaluations.

```
Fgradevals = output.funcCount
```

```
Fgradevals = 14
```

Compare this number to the number of function evaluations without gradients.

```
Fevals
```

```
Fevals = 38
```

Constrained Optimization Example: Changing the Default Termination Tolerances

This example continues to use gradients and solves the same constrained problem

$$\text{minimize } x\exp(-(x^2 + y^2)) + (x^2 + y^2)/20,$$

subject to $xy/2 + (x + 2)^2 + (y - 2)^2/2 \le 2$.

In this case, you achieve a more accurate solution by overriding the default termination criteria (`options.StepTolerance` and `options.OptimalityTolerance`). The default values for the `fmincon` interior-point algorithm are `options.StepTolerance = 1e-10` and `options.OptimalityTolerance = 1e-6`.

Override these two default termination criteria.

```
options = optimoptions(options,...
    'StepTolerance',1e-15,...
    'OptimalityTolerance',1e-8);
```

Call the solver.

```
[x,fval,exitflag,output] = fmincon(@onehump,x0,[],[],[],[],[],[], ...
                                   @tiltellipse,options);
```

```
                                         First-order      Norm of
 Iter F-count          f(x)  Feasibility   optimality         step
    0       1   2.365241e-01   0.000e+00    1.972e-01
    1       2   1.748504e-01   0.000e+00    1.734e-01    2.260e-01
    2       4  -1.570560e-01   0.000e+00    2.608e-01    9.347e-01
    3       6  -6.629161e-02   0.000e+00    1.241e-01    3.103e-01
    4       7  -1.584082e-01   0.000e+00    7.934e-02    1.826e-01
    5       8  -2.349124e-01   0.000e+00    1.912e-02    1.571e-01
    6       9  -2.255299e-01   0.000e+00    1.955e-02    1.993e-02
    7      10  -2.444225e-01   0.000e+00    4.293e-03    3.821e-02
    8      11  -2.446931e-01   0.000e+00    8.100e-04    4.035e-03
    9      12  -2.446933e-01   0.000e+00    1.999e-04    8.126e-04
   10      13  -2.448531e-01   0.000e+00    4.004e-05    3.289e-04
   11      14  -2.448927e-01   0.000e+00    4.036e-07    8.156e-05
   12      15  -2.448931e-01   0.000e+00    4.000e-09    8.230e-07
```

```
Local minimum found that satisfies the constraints.

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

To see the difference made by the new tolerances more accurately, display more decimals in the solution.

```
format long
```

Display the solution found by the solver.

```
x
```

x = *2×1*

```
    -0.972742227363546
     0.468569289098342
```

Compare these values to the values in the previous example.

```
xold
```

xold = *2×1*

```
    -0.972742694488360
     0.468569966693330
```

Determine the change in values.

```
x - xold
```

ans = *2×1*
10^{-6} ×

```
     0.467124813385844
    -0.677594988729435
```

View the function value at the solution.

```
fval
```

fval =
  -0.244893137879894

See how much the solution improved.

```
fval - minfval
```

ans =
    -3.996450220755676e-07

The answer is negative because the new solution is smaller.

View the total number of function evaluations.

```
output.funcCount
```

```
ans =
    15
```

Compare this number to the number of function evaluations in the example solved with user-provided gradients and the default tolerances.

```
Fgradevals
```

```
Fgradevals =
    14
```

Constrained Optimization Example: User-Supplied Hessian

If you supply a Hessian in addition to a gradient, solvers are even more accurate and efficient.

The `fmincon` interior-point algorithm takes a Hessian matrix as a separate function (not part of the objective function). The Hessian function H(x,lambda) evaluates the Hessian of the Lagrangian; see "Hessian for fmincon interior-point algorithm" on page 2-21.

Solvers calculate the values `lambda.ineqnonlin` and `lambda.eqlin`; your Hessian function tells solvers how to use these values.

This example has one inequality constraint, so the Hessian is defined as given in the `hessfordemo` function.

```
type hessfordemo
```

```
function H = hessfordemo(x,lambda)
% HESSFORDEMO Helper function for Tutorial for the Optimization Toolbox demo

%   Copyright 2008-2009 The MathWorks, Inc.

s = exp(-(x(1)^2+x(2)^2));
H = [2*x(1)*(2*x(1)^2-3)*s+1/10, 2*x(2)*(2*x(1)^2-1)*s;
        2*x(2)*(2*x(1)^2-1)*s, 2*x(1)*(2*x(2)^2-1)*s+1/10];
hessc = [2,1/2;1/2,1];
H = H + lambda.ineqnonlin(1)*hessc;
```

In order to use the Hessian, you need to set options appropriately.

```
options = optimoptions('fmincon',...
    'Algorithm','interior-point',...
    'SpecifyConstraintGradient',true,...
    'SpecifyObjectiveGradient',true,...
    'HessianFcn',@hessfordemo);
```

The tolerances are set to their defaults, which should result in fewer function counts. Set options to display the results at each iteration.

```
options.Display = 'iter';
```

Call the solver.

```
[x,fval,exitflag,output] = fmincon(@onehump,x0,[],[],[],[],[],[], ...
                           @tiltellipse,options);
```

```
                                       First-order      Norm of
  Iter F-count            f(x)   Feasibility  optimality        step
     0       1    2.365241e-01    0.000e+00    1.972e-01
     1       3    5.821325e-02    0.000e+00    1.443e-01    8.728e-01
     2       5   -1.218829e-01    0.000e+00    1.007e-01    4.927e-01
     3       6   -1.421167e-01    0.000e+00    8.486e-02    5.165e-02
     4       7   -2.261916e-01    0.000e+00    1.989e-02    1.667e-01
     5       8   -2.433609e-01    0.000e+00    1.537e-03    3.486e-02
     6       9   -2.446875e-01    0.000e+00    2.057e-04    2.727e-03
     7      10   -2.448911e-01    0.000e+00    2.068e-06    4.191e-04
     8      11   -2.448931e-01    0.000e+00    2.001e-08    4.218e-06
```

```
Local minimum found that satisfies the constraints.

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

The results show fewer and different iterations.

Display the solution found by the solver.

```
x
```

```
x = 2×1

  -0.972742246093537
   0.468569316215571
```

View the function value at the solution.

```
fval
```

```
fval =
  -0.244893121872758
```

View the total number of function evaluations.

```
output.funcCount
```

```
ans =
    11
```

Compare this number to the number of function evaluations in the example solved using only gradient evaluations, with the same default tolerances.

```
Fgradevals
```

```
Fgradevals =
    14
```

See Also

More About

- "Passing Extra Parameters" on page 2-57
- "Solver-Based Optimization Problem Setup"

Banana Function Minimization

This example shows how to minimize Rosenbrock's "banana function":

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

$f(x)$ is called the banana function because of its curvature around the origin. It is notorious in optimization examples because of the slow convergence most methods exhibit when trying to solve this problem.

$f(x)$ has a unique minimum at the point $x = [1, 1]$ where $f(x) = 0$. This example shows a number of ways to minimize $f(x)$ starting at the point $x0 = [-1.9, 2]$.

Optimization Without Derivatives

The `fminsearch` function finds a minimum for a problem without constraints. It uses an algorithm that does not estimate any derivatives of the objective function. Rather, it uses a geometric search method described in "fminsearch Algorithm" on page 6-9.

Minimize the banana function using `fminsearch`. Include an output function to report the sequence of iterations.

```
fun = @(x)(100*(x(2) - x(1)^2)^2 + (1 - x(1))^2);
options = optimset('OutputFcn',@bananaout,'Display','off');
x0 = [-1.9,2];
[x,fval,eflag,output] = fminsearch(fun,x0,options);
title 'Rosenbrock solution via fminsearch'
```


Rosenbrock solution via fminsearch
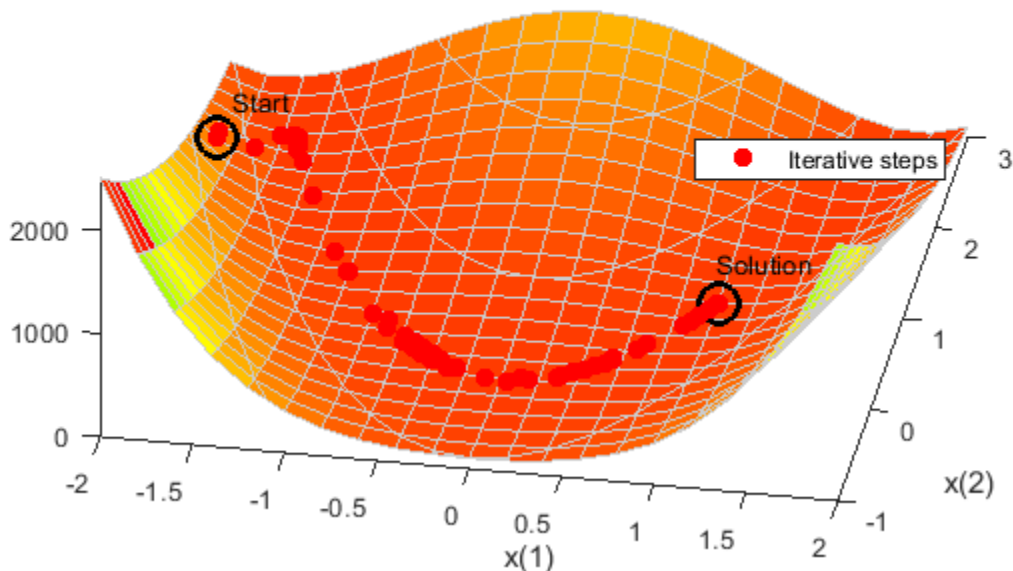
```
Fcount = output.funcCount;
disp(['Number of function evaluations for fminsearch was ',num2str(Fcount)])
```

```
Number of function evaluations for fminsearch was 210
```

```
disp(['Number of solver iterations for fminsearch was ',num2str(output.iterations)])
```

```
Number of solver iterations for fminsearch was 114
```

Optimization with Estimated Derivatives

The `fminunc` function finds a minimum for a problem without constraints. It uses a derivative-based algorithm. The algorithm attempts to estimate not only the first derivative of the objective function, but also the matrix of second derivatives. `fminunc` is usually more efficient than `fminsearch`.

Minimize the banana function using `fminunc`.

```
options = optimoptions('fminunc','Display','off',...
    'OutputFcn',@bananaout,'Algorithm','quasi-newton');
[x,fval,eflag,output] = fminunc(fun,x0,options);
title 'Rosenbrock solution via fminunc'
```


Rosenbrock solution via fminunc
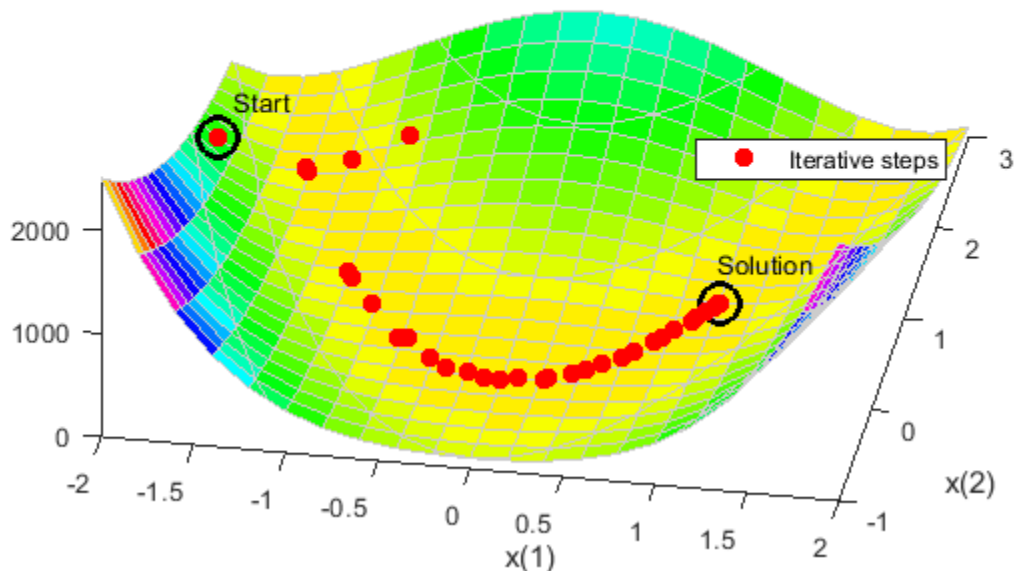
```
Fcount = output.funcCount;
disp(['Number of function evaluations for fminunc was ',num2str(Fcount)])
```

```
Number of function evaluations for fminunc was 150
```

```
disp(['Number of solver iterations for fminunc was ',num2str(output.iterations)])
```

Number of solver iterations for fminunc was 34

Optimization with Steepest Descent

If you attempt to minimize the banana function using a steepest descent algorithm, the high curvature of the problem makes the solution process very slow.

You can run `fminunc` with the steepest descent algorithm by setting the hidden `HessUpdate` option to the value `'steepdesc'` for the `'quasi-newton'` algorithm. Set a larger-than-default maximum number of function evaluations, because the solver does not find the solution quickly. In this case, the solver does not find the solution even after 600 function evaluations.

```
options = optimoptions(options,'HessUpdate','steepdesc',...
    'MaxFunctionEvaluations',600);
[x,fval,eflag,output] = fminunc(fun,x0,options);
title 'Rosenbrock solution via steepest descent'
```


Rosenbrock solution via steepest descent
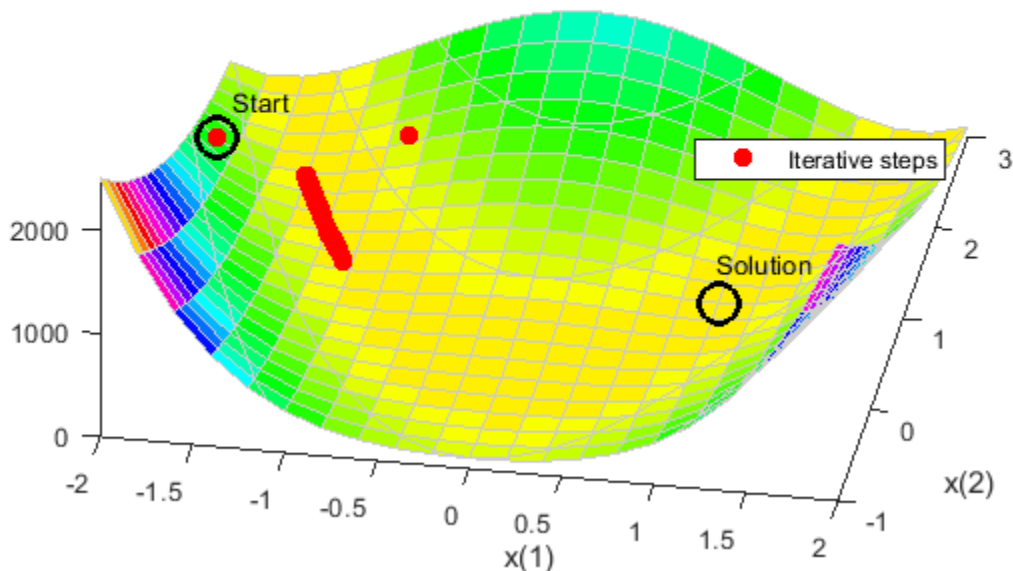
```
Fcount = output.funcCount;
disp(['Number of function evaluations for steepest descent was ',...
    num2str(Fcount)])
```

Number of function evaluations for steepest descent was 600

```
disp(['Number of solver iterations for steepest descent was ',...
    num2str(output.iterations)])
```

Number of solver iterations for steepest descent was 45

Optimization with Analytic Gradient

If you provide a gradient, `fminunc` solves the optimization using fewer function evaluations. When you provide a gradient, you can use the `'trust-region'` algorithm, which is often faster and uses less memory than the `'quasi-newton'` algorithm. Reset the `HessUpdate` and `MaxFunctionEvaluations` options to their default values.

```
grad = @(x)[-400*(x(2) - x(1)^2)*x(1) - 2*(1 - x(1));
            200*(x(2) - x(1)^2)];
fungrad = @(x)deal(fun(x),grad(x));
options = resetoptions(options,{'HessUpdate','MaxFunctionEvaluations'});
options = optimoptions(options,'SpecifyObjectiveGradient',true,...
    'Algorithm','trust-region');
[x,fval,eflag,output] = fminunc(fungrad,x0,options);
title 'Rosenbrock solution via fminunc with gradient'
```


Rosenbrock solution via fminunc with gradient
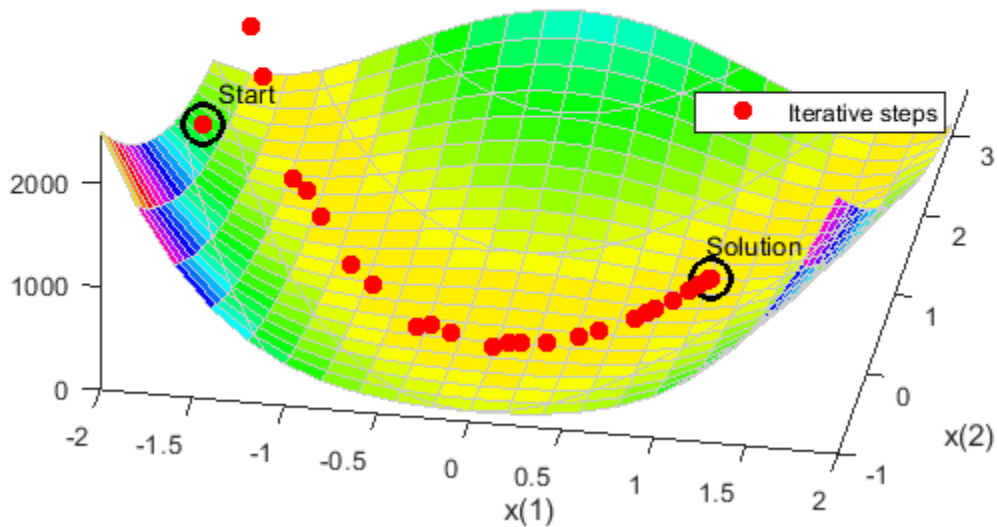
```
Fcount = output.funcCount;
disp(['Number of function evaluations for fminunc with gradient was ',...
    num2str(Fcount)])
```

Number of function evaluations for fminunc with gradient was 32

```
disp(['Number of solver iterations for fminunc with gradient was ',...
    num2str(output.iterations)])
```

Number of solver iterations for fminunc with gradient was 31

Optimization with Analytic Hessian

If you provide a Hessian (matrix of second derivatives), `fminunc` can solve the optimization using even fewer function evaluations. For this problem the results are the same with or without the Hessian.

```
hess = @(x)[1200*x(1)^2 - 400*x(2) + 2, -400*x(1);
            -400*x(1), 200];
fungradhess = @(x)deal(fun(x),grad(x),hess(x));
options.HessianFcn = 'objective';
[x,fval,eflag,output] = fminunc(fungradhess,x0,options);
title 'Rosenbrock solution via fminunc with Hessian'
```


Rosenbrock solution via fminunc with Hessian
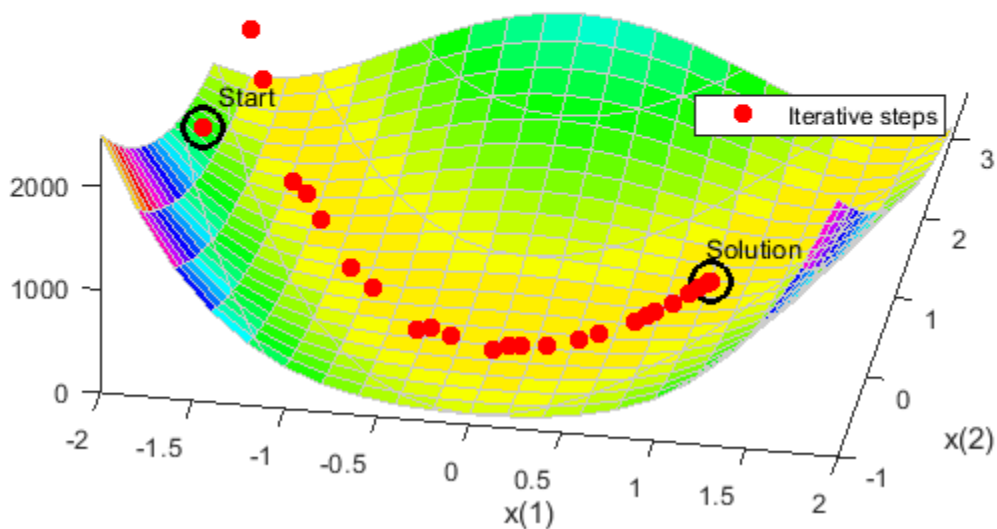
```
Fcount = output.funcCount;
disp(['Number of function evaluations for fminunc with gradient and Hessian was ',...
    num2str(Fcount)])
```

Number of function evaluations for fminunc with gradient and Hessian was 32

```
disp(['Number of solver iterations for fminunc with gradient and Hessian was ',num2str(output.ite
```

Number of solver iterations for fminunc with gradient and Hessian was 31

Optimization with a Least Squares Solver

The recommended solver for a nonlinear sum of squares is `lsqnonlin`. This solver is even more efficient than `fminunc` without a gradient for this special class of problems. To use `lsqnonlin`, do

not write your objective as a sum of squares. Instead, write the underlying vector that `lsqnonlin` internally squares and sums.

```
options = optimoptions('lsqnonlin','Display','off','OutputFcn',@bananaout);
vfun = @(x)[10*(x(2) - x(1)^2),1 - x(1)];
[x,resnorm,residual,eflag,output] = lsqnonlin(vfun,x0,[],[],options);
title 'Rosenbrock solution via lsqnonlin'
```


Rosenbrock solution via lsqnonlin
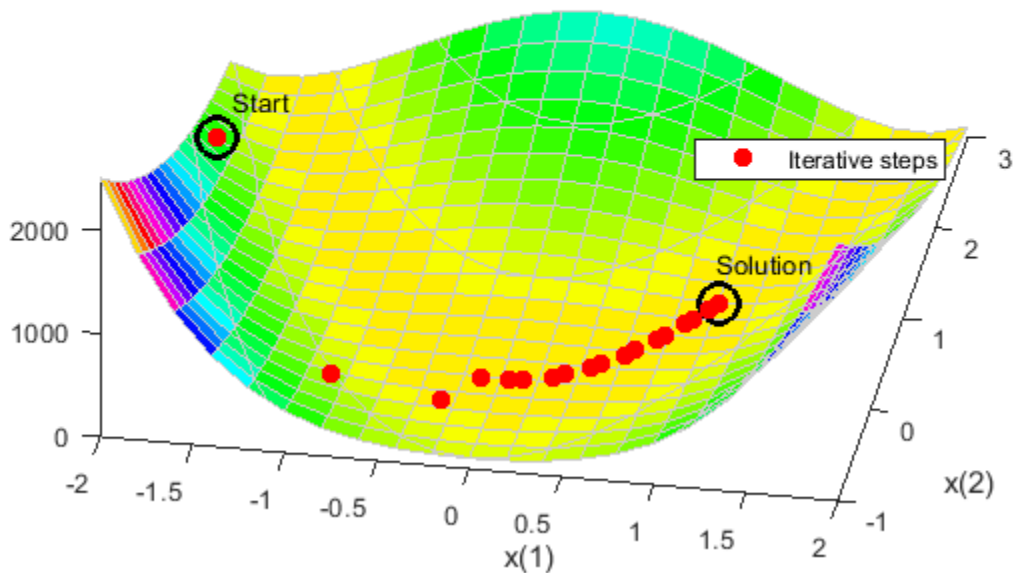
```
Fcount = output.funcCount;
disp(['Number of function evaluations for lsqnonlin was ',...
    num2str(Fcount)])
```

Number of function evaluations for lsqnonlin was 87

```
disp(['Number of solver iterations for lsqnonlin was ',num2str(output.iterations)])
```

Number of solver iterations for lsqnonlin was 28

Optimization with a Least Squares Solver and Jacobian

As in the minimization using a gradient for `fminunc`, `lsqnonlin` can use derivative information to lower the number of function evaluations. Provide the Jacobian of the nonlinear objective function vector and run the optimization again.

```
jac = @(x)[-20*x(1),10;
           -1,0];
vfunjac = @(x)deal(vfun(x),jac(x));
options.SpecifyObjectiveGradient = true;
```

```
[x,resnorm,residual,eflag,output] = lsqnonlin(vfunjac,x0,[],[],options);
title 'Rosenbrock solution via lsqnonlin with Jacobian'
```

Rosenbrock solution via lsqnonlin with Jacobian
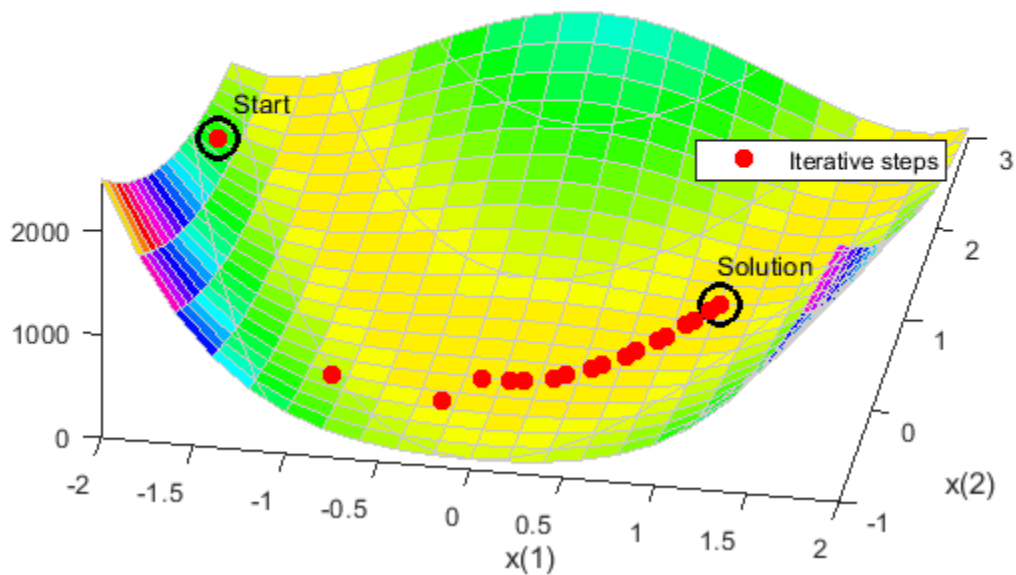

```
Fcount = output.funcCount;
disp(['Number of function evaluations for lsqnonlin with Jacobian was ',...
    num2str(Fcount)])
```

Number of function evaluations for lsqnonlin with Jacobian was 29

```
disp(['Number of solver iterations for lsqnonlin with Jacobian was ',...
    num2str(output.iterations)])
```

Number of solver iterations for lsqnonlin with Jacobian was 28

See Also

More About

- "Solve a Constrained Nonlinear Problem, Solver-Based" on page 1-11
- "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97

Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™

This example shows 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.

```
function f = expensive_objfun(x)
%EXPENSIVE_OBJFUN An expensive objective function used in optimparfor example.

%   Copyright 2007-2013 The MathWorks, Inc.

% 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 2007-2013 The MathWorks, Inc.

% 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 time.

```
startPoint = [-1 1 1 -1];
options = optimoptions('fmincon','Display','iter','Algorithm','interior-point');
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);
```

```
                                          First-order      Norm of
 Iter F-count            f(x)   Feasibility  optimality       step
    0       5    1.839397e+00    1.500e+00    3.211e+00
```

```
 1      11    -9.760099e-01    3.708e+00    7.902e-01    2.362e+00
 2      16    -1.480976e+00    0.000e+00    8.344e-01    1.069e+00
 3      21    -2.601599e+00    0.000e+00    8.390e-01    1.218e+00
 4      29    -2.823630e+00    0.000e+00    2.598e+00    1.118e+00
 5      34    -3.905339e+00    0.000e+00    1.210e+00    7.302e-01
 6      39    -6.212992e+00    3.934e-01    7.372e-01    2.405e+00
 7      44    -5.948762e+00    0.000e+00    1.784e+00    1.905e+00
 8      49    -6.940062e+00    1.233e-02    7.668e-01    7.553e-01
 9      54    -6.973887e+00    0.000e+00    2.549e-01    3.920e-01
10      59    -7.142993e+00    0.000e+00    1.903e-01    4.735e-01
11      64    -7.155325e+00    0.000e+00    1.365e-01    2.626e-01
12      69    -7.179122e+00    0.000e+00    6.336e-02    9.115e-02
13      74    -7.180116e+00    0.000e+00    1.069e-03    4.670e-02
14      79    -7.180409e+00    0.000e+00    7.799e-04    2.815e-03
15      84    -7.180410e+00    0.000e+00    6.189e-06    3.122e-04

Local minimum found that satisfies the constraints.

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

Serial FMINCON optimization takes 17.0722 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. Pass empty matrices `[]` 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 = optimoptions('ga','MaxGenerations',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    Func-count       f(x)           f(x)     Generations
    1            100        -5.546e+05      1.483e+15        0
    2            150        -5.581e+17     -1.116e+16        0
    3            200        -7.556e+17      6.679e+22        0
    4            250        -7.556e+17     -7.195e+16        1
    5            300        -9.381e+27     -1.876e+26        0
    6            350        -9.673e+27     -7.497e+26        0
    7            400        -4.511e+36     -9.403e+34        0
    8            450        -5.111e+36     -3.011e+35        0
```

```
    9             500     -7.671e+36     9.346e+37        0
   10             550      -1.52e+43     -3.113e+41       0
   11             600     -2.273e+45     -4.67e+43        0
   12             650     -2.589e+47     -6.281e+45       0
   13             700     -2.589e+47     -1.015e+46       1
   14             750     -8.149e+47     -5.855e+46       0
   15             800     -9.503e+47      -1.29e+47       0
Optimization terminated: maximum number of generations exceeded.
Serial GA optimization takes 80.2351 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
```

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);
```

```
                                            First-order      Norm of
 Iter F-count            f(x)   Feasibility   optimality        step
    0        5    1.839397e+00     1.500e+00    3.211e+00
    1       11   -9.760099e-01     3.708e+00    7.902e-01    2.362e+00
    2       16   -1.480976e+00     0.000e+00    8.344e-01    1.069e+00
    3       21   -2.601599e+00     0.000e+00    8.390e-01    1.218e+00
    4       29   -2.823630e+00     0.000e+00    2.598e+00    1.118e+00
    5       34   -3.905339e+00     0.000e+00    1.210e+00    7.302e-01
    6       39   -6.212992e+00     3.934e-01    7.372e-01    2.405e+00
    7       44   -5.948762e+00     0.000e+00    1.784e+00    1.905e+00
    8       49   -6.940062e+00     1.233e-02    7.668e-01    7.553e-01
    9       54   -6.973887e+00     0.000e+00    2.549e-01    3.920e-01
   10       59   -7.142993e+00     0.000e+00    1.903e-01    4.735e-01
   11       64   -7.155325e+00     0.000e+00    1.365e-01    2.626e-01
   12       69   -7.179122e+00     0.000e+00    6.336e-02    9.115e-02
   13       74   -7.180116e+00     0.000e+00    1.069e-03    4.670e-02
   14       79   -7.180409e+00     0.000e+00    7.799e-04    2.815e-03
   15       84   -7.180410e+00     0.000e+00    6.189e-06    3.122e-04
```

```
Local minimum found that satisfies the constraints.

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

Parallel FMINCON optimization takes 8.11945 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 = optimoptions(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
```

```
                            Best           Mean         Stall
Generation    Func-count      f(x)           f(x)      Generations
    1            100       -5.546e+05      1.483e+15         0
    2            150       -5.581e+17     -1.116e+16         0
    3            200       -7.556e+17      6.679e+22         0
    4            250       -7.556e+17     -7.195e+16         1
    5            300       -9.381e+27     -1.876e+26         0
    6            350       -9.673e+27     -7.497e+26         0
    7            400       -4.511e+36     -9.403e+34         0
    8            450       -5.111e+36     -3.011e+35         0
    9            500       -7.671e+36      9.346e+37         0
   10            550        -1.52e+43     -3.113e+41         0
   11            600       -2.273e+45      -4.67e+43         0
   12            650       -2.589e+47     -6.281e+45         0
   13            700       -2.589e+47     -1.015e+46         1
   14            750       -8.149e+47     -5.855e+46         0
   15            800       -9.503e+47      -1.29e+47         0
Optimization terminated: maximum number of generations exceeded.
Parallel GA optimization takes 15.6984 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')
```
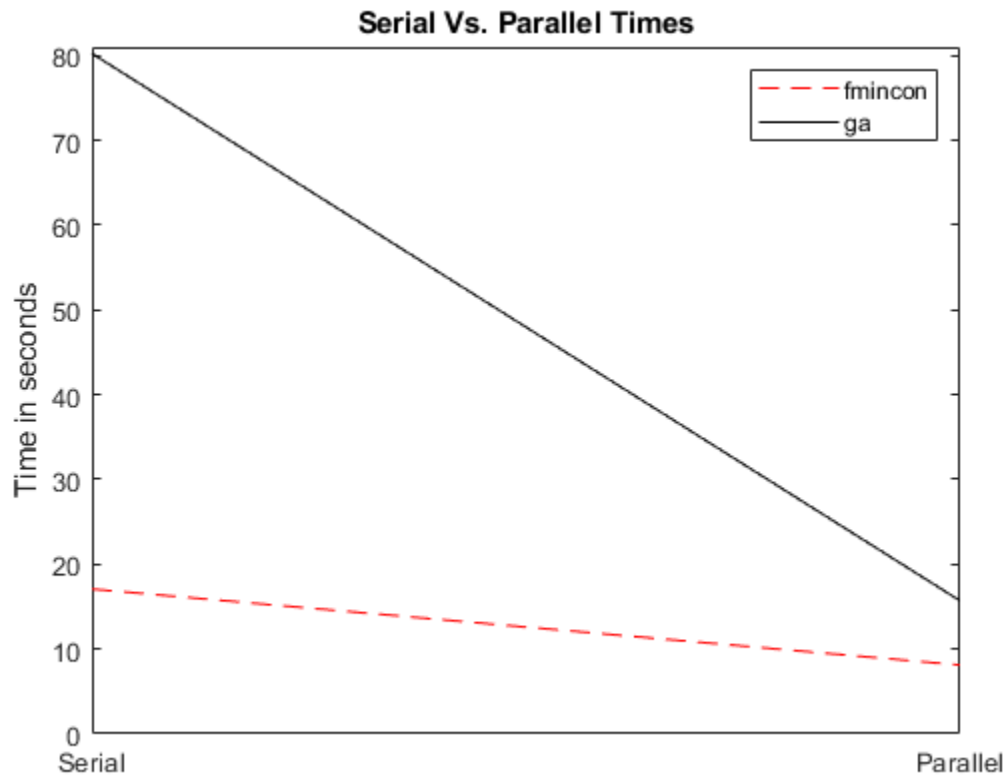

Utilizing parallel function evaluation via `parfor` improved the efficiency of both `fmincon` and `ga`. The improvement is typically better for expensive objective and constraint functions.

See Also

More About

- "Parallel Computing"

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

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

subject to the constraints

$$x_1x_2 - x_1 - x_2 \leq -1.5$$
$$x_1x_2 \geq -10.$$

Because neither of the constraints is linear, create a function, `confun.m`, that returns the value of both constraints in a vector `c`. Because the `fmincon` solver expects the constraints to be written in the form $c(x) \leq 0$, write your constraint function to return the following value:

$$c(x) = \begin{bmatrix} x_1x_2 - x_1 - x_2 + 1.5 \\ -10 - x_1x_2 \end{bmatrix}.$$

Create Objective Function

The helper function `objfun` is the objective function; it appears at the end of this example on page 6-0 . Set the `fun` argument as a function handle to the `objfun` function.

```
fun = @objfun;
```

Create Nonlinear Constraint Function

Nonlinear constraint functions must return two arguments: `c`, the inequality constraint, and `ceq`, the equality constraint. Because this problem has no equality constraint, the helper function `confun` at the end of this example on page 6-0 returns `[]` as the equality constraint.

Solve Problem

Set the initial point to `[-1,1]`.

```
x0 = [-1,1];
```

The problem has no bounds or linear constraints. Set those arguments to `[]`.

```
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
```

Solve the problem using `fmincon`.

```
[x,fval] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,@confun)
```

```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
```

```
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.
```

x = *1×2*

```
   -9.5473    1.0474
```

fval = 0.0236

Examine Solution

The exit message indicates that the solution is feasible with respect to the constraints. To double-check, evaluate the nonlinear constraint function at the solution. Negative values indicate satisfied constraints.

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

c = *2×1*
10^{-4} ×

```
   -0.3179
   -0.3062
```

ceq =

```
     []
```

Both nonlinear constraints are negative and close to zero, indicating that the solution is feasible and that both constraints are active at the solution.

Helper Functions

This code creates the objfun helper 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);
end
```

This code creates the confun helper function.

```
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
```

See Also

Related Examples

Nonlinear Constraints with Gradients

This example shows how to solve a nonlinear problem with nonlinear constraints using derivative information.

Ordinarily, minimization routines use numerical gradients calculated by finite-difference approximation. This procedure systematically perturbs each variable in order to calculate function and constraint partial derivatives. Alternatively, you can provide a function to compute partial derivatives analytically. Typically, when you provide derivative information, solvers work more accurately and efficiently.

Objective Function and Nonlinear Constraint

The problem is to solve

$$\min_{x} f(x) = e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right),$$

subject to the constraints

$$x_1x_2 - x_1 - x_2 \leq -1.5$$
$$x_1x_2 \geq -10.$$

Because the `fmincon` solver expects the constraints to be written in the form $c(x) \leq 0$, write your constraint function to return the following value:

$$c(x) = \begin{bmatrix} x_1x_2 - x_1 - x_2 + 1.5 \\ -10 - x_1x_2 \end{bmatrix}.$$

Objective Function with Gradient

The objective function is

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

Compute the gradient of $f(x)$ with respect to the variables x_1 and x_2.

$$\nabla f(x) = \begin{bmatrix} f(x) + \exp(x_1)(8x_1 + 4x_2) \\ \exp(x_1)(4x_1 + 4x_2 + 2) \end{bmatrix}.$$

The `objfungrad` helper function at the end of this example on page 6-0

```
fun = @objfungrad;
```

Constraint Function with Gradient

The helper function `confungrad` is the nonlinear constraint function; it appears at the end of this

The derivative information for the inequality constraint has each column correspond to one constraint. In other words, the gradient of the constraints is in the following format:

$$\begin{bmatrix} \dfrac{\partial c_1}{\partial x_1} & \dfrac{\partial c_2}{\partial x_1} \\[2mm] \dfrac{\partial c_1}{\partial x_2} & \dfrac{\partial c_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 - 1 & -x_2 \\ x_1 - 1 & -x_1 \end{bmatrix}.$$

Set `@confungrad` as the nonlinear constraint function.

```
nonlcon = @confungrad;
```

Set Options to Use Derivative Information

Indicate to the `fmincon` solver that the objective and constraint functions provide derivative information. To do so, use `optimoptions` to set the `SpecifyObjectiveGradient` and `SpecifyConstraintGradient` option values to `true`.

```
options = optimoptions('fmincon',...
    'SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true);
```

Solve Problem

Set the initial point to `[-1,1]`.

```
x0 = [-1,1];
```

The problem has no bounds or linear constraints, so set those argument values to `[]`.

```
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
```

Call `fmincon` to solve the problem.

```
[x,fval] = 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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x = 1×2

   -9.5473    1.0474


fval = 0.0236
```

The solution is the same as in the example "Nonlinear Inequality Constraints" on page 6-61, which solves the problem without using derivative information. The advantage of using derivatives is that solving the problem takes fewer function evaluations while gaining robustness, although this advantage is not obvious in this example. Using even more derivative information, as in "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-66, gives even more benefit, such as fewer solver iterations.

Helper Functions

This code creates the `objfungrad` helper function.

```
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
end
```

This code creates the `confungrad` helper function.

```
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
end
```

See Also

Related Examples

- "Nonlinear Inequality Constraints" on page 6-61
- "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-66

fmincon Interior-Point Algorithm with Analytic Hessian

This example shows how to use derivative information to make the solution process faster and more robust. The `fmincon` interior-point algorithm can accept a Hessian function as an input. When you supply a Hessian, you can obtain a faster, more accurate solution to a constrained minimization problem.

The helper function `bigtoleft` is an objective function that grows rapidly negative as the `x(1)` coordinate becomes negative. Its gradient is a three-element vector. The code for the `bigtoleft` helper function appears at the end of this example on page 6-0 .

The constraint set for this example is the intersection of the interiors of two cones—one pointing up, and one pointing down. The constraint function is a two-component vector containing one component for each cone. Because this example is three-dimensional, the gradient of the constraint is a 3-by-2 matrix. The code for the `twocone` helper function appears at the end of this example on page 6-0 .

Create a figure of the constraints, colored using the objective function.

```
% Create figure
figure1 = figure;

% Create axes
axes1 = axes('Parent',figure1);
view([-63.5 18]);
grid('on');
hold('all');

% Set up polar coordinates and two cones
r=0:.1:6.5;
th=2*pi*(0:.01:1);
x=r'*cos(th);
y=r'*sin(th);
z=-10+sqrt(x.^2+y.^2);
zz=3-sqrt(x.^2+y.^2);

% Evaluate objective function on cone surfaces
newxf=reshape(bigtoleft([x(:),y(:),z(:)]),66,101)/3000;
newxg=reshape(bigtoleft([x(:),y(:),z(:)]),66,101)/3000;

% Create lower surf with color set by objective
surf(x,y,z,newxf,'Parent',axes1,'EdgeAlpha',0.25);

% Create upper surf with color set by objective
surf(x,y,zz,newxg,'Parent',axes1,'EdgeAlpha',0.25);
axis equal
```
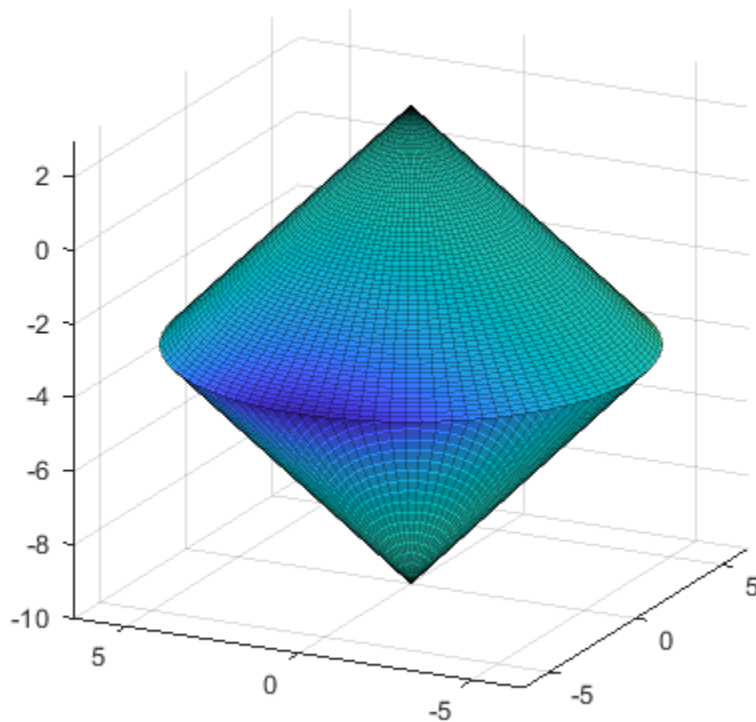
Create Hessian Function

To use second-order derivative information in the `fmincon` solver, you must create a Hessian that is the Hessian of the Lagrangian. The Hessian of the Lagrangian is given by the equation

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

Here, $f(x)$ is the `bigtoleft` function, and the $c_i(x)$ are the two cone constraint functions. The `hessinterior` helper function at the end of this example on page 6-0   computes the Hessian of the Lagrangian at a point x with the Lagrange multiplier structure `lambda`. The function first computes $\nabla^2 f(x)$. It then computes the two constraint Hessians $\nabla^2 c_1(x)$ and $\nabla^2 c_2(x)$, multiplies them by their corresponding Lagrange multipliers `lambda.ineqnonlin(1)` and `lambda.ineqnonlin(2)`, and adds them. You can see from the definition of the `twocone` constraint function that $\nabla^2 c_1(x) = \nabla^2 c_2(x)$, which simplifies the calculation.

Create Options to Use Derivatives

To enable `fmincon` to use the objective gradient, constraint gradients, and Hessian, you must set appropriate options. The `HessianFcn` option using the Hessian of the Lagrangian is available only for the interior-point algorithm.

```
options = optimoptions('fmincon','Algorithm','interior-point',...
    "SpecifyConstraintGradient",true,"SpecifyObjectiveGradient",true,...
    'HessianFcn',@hessinterior);
```

Minimize Using All Derivative Information

Set the initial point `x0 = [-1,-1,-1]`.

```
x0 = [-1,-1,-1];
```

The problem has no linear constraints or bounds. Set those arguments to `[]`.

```
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
```

Solve the problem.

```
[x,fval,eflag,output] = fmincon(@bigtoleft,x0,...
            A,b,Aeq,beq,lb,ub,@twocone,options);

Local minimum found that satisfies the constraints.

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

Examine Solution and Solution Process

Examine the solution, objective function value, exit flag, and number of function evaluations and iterations.

```
disp(x)

   -6.5000   -0.0000   -3.5000

disp(fval)

  -2.8941e+03

disp(eflag)

     1

disp([output.funcCount,output.iterations])

     7     6
```

If you do not use a Hessian function, `fmincon` takes more iterations to converge and requires more function evaluations.

```
options.HessianFcn = [];
[x2,fval2,eflag2,output2] = fmincon(@bigtoleft,x0,...
            A,b,Aeq,beq,lb,ub,@twocone,options);

Local minimum found that satisfies the constraints.

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

```
disp([output2.funcCount,output2.iterations])

     13      9
```

If you also do not include the gradient information, `fmincon` takes the same number of iterations, but requires many more function evaluations.

```
options.SpecifyConstraintGradient = false;
options.SpecifyObjectiveGradient = false;
[x3,fval3,eflag3,output3] = fmincon(@bigtoleft,x0,...
            A,b,Aeq,beq,lb,ub,@twocone,options);
```

```
Local minimum found that satisfies the constraints.

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

```
disp([output3.funcCount,output3.iterations])

     43      9
```

Helper Functions

This code creates the `bigtoleft` helper function.

```
function [f gradf] = bigtoleft(x)
% This is a simple function that grows rapidly negative
% as x(1) becomes 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
end
```

This code creates the `twocone` helper function.

```
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
end
```

This code creates the `hessinterior` helper function.

```
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;
end
```

See Also

Related Examples

- "Linear or Quadratic Objective with Quadratic Constraints" on page 6-71
- "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97

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. The example generates and uses the gradient and Hessian of the objective and constraint functions.

Quadratic Constrained Problem

Suppose that your problem has the form

$$\min_{x}\left(\frac{1}{2}x^{T}Qx + f^{T}x + c\right)$$

subject to

$$\frac{1}{2}x^{T}H_{i}x + k_{i}^{T}x + d_{i} \le 0,$$

where $1 \le i \le m$. 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, which means `fmincon` is the appropriate solver according to the "Optimization Decision Table" on page 2-4.

The example assumes that the quadratic matrices are symmetric without loss of generality. You can replace a nonsymmetric H (or Q) matrix with an equivalent symmetrized version $(H + H^T)/2$.

If x has N components, then Q and H_i are N-by-N matrices, f and k_i are N-by-1 vectors, and c and 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 a column vector.)

```
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

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

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

`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',...
    'SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true,...
    'HessianFcn',@(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. View 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` takes only 10 function evaluations to solve the problem.

Compare this result to the solution without the Hessian.

```
options.HessianFcn = [];
[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...'
```

In this case, `fmincon` takes 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.

See Also

Related Examples

- "fmincon Interior-Point Algorithm with Analytic Hessian" on page 6-66
- "Including Gradients and Hessians" on page 2-19
- "Including Gradients in Constraint Functions" on page 2-38

Nonlinear Equality and Inequality Constraints

This example shows how to solve an optimization problem containing nonlinear constraints. 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.

Nonlinear Constraints

Suppose that your nonlinear equality constraint is

$$x_1^2 + x_2 = 1$$

and your nonlinear inequality constraint is

$$x_1 x_2 \geq -10.$$

Rewrite these constraints as

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

The `confuneq` function at the end of this example on page 6-0   implements these inequalities in the correct syntax.

Objective Function

Solve the problem

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

subject to the constraints. The `objfun` function at the end of this example on page 6-0   implements this objective function.

Solve Problem

Solve the problem by calling the `fmincon` solver. This solver requires an initial point; use the point `x0 = [-1,-1]`.

```
x0 = [-1,-1];
```

There are no bounds or linear constraints in the problem, so set those inputs to `[]`.

```
A = [];
b = [];
Aeq = [];
```

```
beq = [];
lb = [];
ub = [];
```

Call the solver.

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

```
Local minimum found that satisfies the constraints.

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

x = 1×2

   -0.7529    0.4332


fval = 1.5093
```

The solver reports that the constraints are satisfied at the solution. Check the nonlinear constraints at the solution.

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

```
c = -9.6739
```

```
ceq = 2.0666e-12
```

c is less than 0, as required. `ceq` is equal to 0 within the default constraint tolerance of `1e-6`.

Helper Functions

The following code creates the `confuneq` function.

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

The following code creates the `objfun` 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);
end
```

See Also

Related Examples

- "Nonlinear Inequality Constraints" on page 6-61
- "Optimize Live Editor Task with fmincon Solver" on page 6-77

Optimize Live Editor Task with fmincon Solver

This example shows how to use the **Optimize** Live Editor task with the `fmincon` solver to minimize a quadratic subject to linear and nonlinear constraints and bounds.

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

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

subject to the constraints

$$
\begin{aligned}
0.5 &\leq x_1 & \text{(bound)} \\
-x_1 - x_2 + 1 &\leq 0 & \text{(linear inequality)}
\end{aligned}
$$

$$
\left.
\begin{aligned}
-x_1^2 - x_2^2 + 1 &\leq 0 \\
-9x_1^2 - x_2^2 + 9 &\leq 0 \\
-x_1^2 + x_2 &\leq 0 \\
-x_2^2 + x_1 &\leq 0
\end{aligned}
\right\} \text{(nonlinear inequality)}
$$

The starting point `x0` for this problem is $x_1 = 3$ and $x_2 = 1$.

Start Optimize Live Editor Task

Create a new live Script by clicking the **New Live Script** button in the **File** section on the **Home** tab.

Insert an **Optimize** Live Editor task. Click the **Insert** tab and then, in the **Code** section, select **Task > Optimize**.

For later use in entering problem data, select **Insert > Section Break**. New sections appear above and below the task.

Enter Problem Data

1   Starting from the top of the task, enter the problem type and constraint types. Click the **Objective > Quadratic** button and the **Constraints > Lower bounds**, **Linear inequality**, and **Nonlinear** buttons. The task shows that the recommended solver is `fmincon`.

2   **Objective Function**

The objective function is simple enough to represent as an anonymous function. Position the cursor in the section above the task and enter this code.

```
fun = @(x)sum(x.^2);
```

3 **Lower Bound**

The problem contains the lower bound $x_1 \geq 0.5$. Express this bound as a variable lb. With the cursor at the end of the line defining the objective function, press **Enter**, and enter the following code to specify the lower bound.

```
lb = [0.5 -Inf];
```

4 **Initial Point**

With the cursor at the end of the line defining the lower bound, press **Enter**, and enter the following code to set the initial point.

```
x0 = [3,1];
```

5 **Linear Constraint**

With the cursor at the end of the line defining the initial point, press **Enter**, and enter the following code to set the linear constraint.

```
A = [-1,-1];
b = -1;
```

6 **Run Section**

The top section now includes five parameters.

```
1   fun = @(x)sum(x.^2);
2   lb = [0.5 -Inf];
3   x0 = [3,1];
4   A = [-1,-1];
5   b = -1;
```

Next, you need to run the section to place the parameters in the workspace as variables. To do so, click the left-most area of the section, which contains a bar of diagonal stripes. After you click this area, the bar becomes a solid bar, indicating the variables are now in the workspace. (Note: You can also press **Ctrl+Enter** to run the section.)

7 **Set Problem Data**

Enter the variables in the **Select problem data** section of the task. To specify the objective function, select **Objective function > Function handle** and choose **fun**.

8 Set the initial point **x0**.

9 Select **Lower bounds > From workspace** and select **lb**.

10 Set the linear inequality constraint variables A and b in the **Linear inequality** area.

11 Now specify the nonlinear inequality constraints. In the **Select problem data** section, select **Nonlinear > Local function**, and then click the **New** button. The function appears in a new section below the task. Edit the resulting code to contain the following uncommented lines.

```
function [c,ceq] = constraintFcn(x)
% You can include commented code lines or not.
% Be sure that just these uncommented lines remain:
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 = [];
end
```

12 In the **Select problem data** section, select the **constraintFcn** function.

13 Monitor Progress

In the **Display progress** section of the task, select **Text display > Each iteration** so you can monitor the solver progress. Select **Objective value** for the plot.

Your setup looks like this:

Run Solver and Examine Results

To run the solver, click the options button ⁝ at the top right of the task window, and select **Run Section**.

The plot appears in a separate figure window and in the task output area.

To see where the solution variables are returned, look at the top of the task.

Optimize

solution , objectiveValue = Minimize **fun** using **fmincon** solver

The final point and its associated objective function value appear in the `solution` and `objectiveValue` variables in the workspace. View these values by entering this code in the live editor section below the task.

```
solution, objectiveValue
```

Press **Ctrl+Enter** to run the section.

```
solution, objectiveValue

solution = 1×2
      1.0000    1.0000

objectiveValue = 2.0000
```

See Also
Optimize | `fmincon`

Related Examples

- "Solve a Constrained Nonlinear Problem, Solver-Based" on page 1-11
- "Use Optimize Live Editor Task Effectively" on page 1-39
- "Add Interactive Tasks to a Live Script"

Minimization with Bound Constraints and Banded Preconditioner

This example show how to solve a nonlinear problem with bounds using the `fmincon trust-region-reflective` algorithm. This algorithm has provisions for added efficiency when the problem is sparse, has an analytic gradient, and has known structure such as its Hessian pattern.

Objective Function With Gradient

For a given *n* that is a positive multiple of 4, the objective function is

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

where $p = 7/3$, $x_0 = 0$, and $x_{n+1} = 0$. The `tbroyfg` function at the end of this example on page 6-0 implements the objective function, including its gradient.

The problem has bounds: $-10 \le x_i \le 10$ for all *i*. Use $n = 800$.

```
n = 800;
lb = -10*ones(n,1);
ub = -lb;
```

Hessian Pattern

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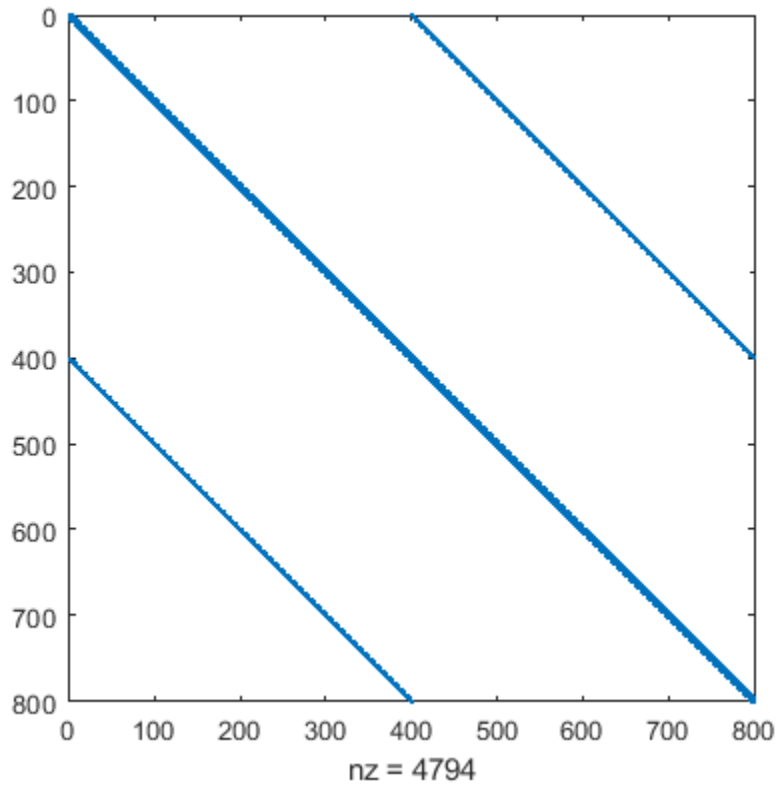
In this plot, the center stripe is itself a five-banded matrix. The following plot shows the matrix more clearly.

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

nz = 94
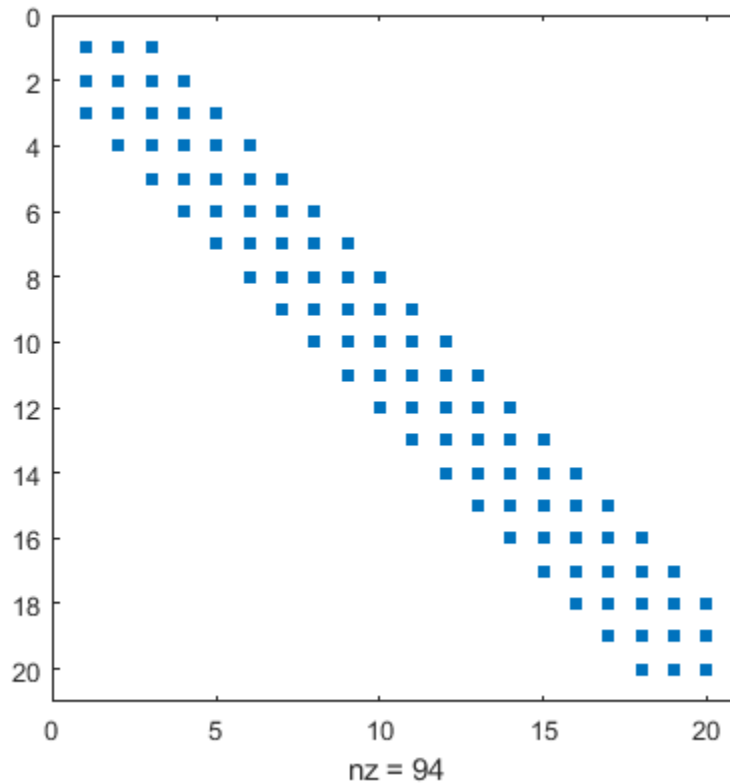
Problem Options

Set options to use the `trust-region-reflective` algorithm. This algorithm requires you to set the `SpecifyObjectiveGradient` option to `true`.

Also, 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. This is because `fmincon` attempts to use finite differencing on a full Hessian matrix of 640,000 nonzero entries.

```
options = optimoptions('fmincon','SpecifyObjectiveGradient',true,'HessPattern',Hstr,...
    'Algorithm','trust-region-reflective');
```

Solve Problem

Set the initial point to –1 for odd indices and +1 for even indices.

```
x0 = -ones(n,1);
x0(2:2:n) = 1;
```

There are no linear or nonlinear constraints, so set those parameters to `[]`.

```
A = [];
b = [];
Aeq = [];
beq = [];
nonlcon = [];
```

Call `fmincon` to solve the problem.

```
[x,fval,exitflag,output] = ...
    fmincon(@tbroyfg,x0,A,b,Aeq,beq,lb,ub,nonlcon,options);
```

Local minimum possible.

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

Examine Solution and Solution Process

Examine the exit flag, objective function value, first-order optimality measure, and number of solver iterations.

```
disp(exitflag);

     3
```

```
disp(fval)

  270.4790
```

```
disp(output.firstorderopt)

    0.0163
```

```
disp(output.iterations)

     7
```

`fmincon` did not take very many iterations to reach a solution. However, the solution has a relatively high first-order optimality measure, which is the reason that the exit flag is not the more preferable value of 1.

Improve Solution

Try using a five-banded preconditioner instead of the default diagonal preconditioner. Using `optimoptions`, set the `PrecondBandWidth` option to 2 and solve the problem again. (The bandwidth is the number of upper (or lower) diagonals, not counting the main diagonal.)

```
options.PrecondBandWidth = 2;
[x2,fval2,exitflag2,output2] = ...
    fmincon(@tbroyfg,x0,A,b,Aeq,beq,lb,ub,nonlcon,options);
```

Local minimum possible.

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

```
disp(exitflag2);

     3
```

```
disp(fval2)

  270.4790
```

```
disp(output2.firstorderopt)

  7.5340e-05
```

```
disp(output2.iterations)
```

```
     9
```

The exit flag and objective function value do not appear to change. However, the number of iterations increased, and the first-order optimality measure decreased considerably. Compute the difference in objective function value.

```
disp(fval2 - fval)
```

```
  -2.9005e-07
```

The objective function value decreased by a tiny amount. The improvement in the solution is mainly an improvement in the first-order optimality measure, without much improvement in the objective function.

Helper Function

This code creates the `tbroyfg` function.

```matlab
function [f,grad] = tbroyfg(x,dummy)
%TBROYFG Objective function and its gradients for nonlinear minimization
% with bound constraints and banded preconditioner.
% Documentation example.

%   Copyright 1990-2008 The MathWorks, Inc.

n = length(x);  % n should be a multiple of 4
p = 7/3;
y=zeros(n,1);
i = 2:(n-1);
y(i) = abs((3-2*x(i)).*x(i) - x(i-1) - x(i+1)+1).^p;
y(n) = abs((3-2*x(n)).*x(n) - x(n-1)+1).^p;
y(1) = abs((3-2*x(1)).*x(1) - x(2)+1).^p;
j = 1:(n/2);
z = zeros(length(j),1);
z(j) = abs(x(j) + x(j+n/2)).^p;
f = 1 + sum(y) + sum(z);
%
% Evaluate the gradient.
if nargout > 1
   g = zeros(n,1);
   t = zeros(n,1);
   i = 2:(n-1);
   t(i) = (3-2*x(i)).*x(i) - x(i-1) - x(i+1) + 1;
   g(i) = p*abs(t(i)).^(p-1).*sign(t(i)).*(3-4*x(i));
   g(i-1) = g(i-1) - p*abs(t(i)).^(p-1).*sign(t(i));
   g(i+1) = g(i+1) - p*abs(t(i)).^(p-1).*sign(t(i));
   tt = (3-2*x(n)).*x(n) - x(n-1) + 1;
   g(n) = g(n) + p*abs(tt).^(p-1).*sign(tt).*(3-4*x(n));
   g(n-1) = g(n-1) - p*abs(tt).^(p-1).*sign(tt);
   tt = (3-2*x(1)).*x(1)-x(2)+1;
   g(1) = g(1) + p*abs(tt).^(p-1).*sign(tt).*(3-4*x(1));
   g(2) = g(2) - p*abs(tt).^(p-1).*sign(tt);
   j = 1:(n/2);
   t(j) = x(j) + x(j+n/2);
   g(j) = g(j) + p*abs(t(j)).^(p-1).*sign(t(j));
   jj = j + (n/2);
```

```
    g(jj) = g(jj) + p*abs(t(j)).^(p-1).*sign(t(j));
    grad = g;
end
end
```

Minimization with Linear Equality Constraints, Trust-Region Reflective Algorithm

The `fmincon` `trust-region-reflective` algorithm can minimize a nonlinear objective function subject to linear equality constraints only (no bounds or any other constraints). For example, 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. This example takes $n = 1000$.

Create Problem

The `browneq.mat` file contains matrices `Aeq` and `beq` that represent the linear constraints `Aeq*x = beq`. The `Aeq` matrix has 100 rows representing 100 linear constraints (so `Aeq` is a 100-by-1000 matrix). Load the `browneq.mat` data.

```
load browneq.mat
```

The `brownfgh` function at the end of this example on page 6-0   implements the objective function, including its gradient and Hessian.

Set Options

The `trust-region-reflective` algorithm requires the objective function to include a gradient. The algorithm accepts a Hessian in the objective function. Set the options to include all of the derivative information.

```
options = optimoptions('fmincon','Algorithm','trust-region-reflective',...
    'SpecifyObjectiveGradient',true,'HessianFcn','objective');
```

Solve Problem

Set the initial point to –1 for odd indices and +1 for even indices.

```
n = 1000;
x0 = -ones(n,1);
x0(2:2:n) = 1;
```

There are no bounds, linear inequality or nonlinear constraints, so set those parameters to `[]`.

```
A = [];
b = [];
lb = [];
ub = [];
nonlcon = [];
```

Call `fmincon` to solve the problem.

```
[x,fval,exitflag,output] = ...
    fmincon(@brownfgh,x0,A,b,Aeq,beq,lb,ub,nonlcon,options);

Local minimum possible.
```

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

Examine Solution and Solution Process

Examine the exit flag, objective function value, and constraint violation.

disp(exitflag)

     3

disp(fval)

   205.9313

disp(output.constrviolation)

    2.2249e-13

The exitflag value of 3 also indicates that `fmincon` stopped because the change in the objective function value was less than the tolerance `FunctionTolerance`. The final objective function value is given by `fval`. Constraints are satisfied, as you see in `output.constrviolation`, which shows a very small number.

To calculate the constraint violation yourself, execute the following code.

norm(Aeq*x-beq,Inf)

ans = 2.2249e-13

Helper Function

The following code creates the `brownfgh` function.

```
function [f,g,H] = brownfgh(x)
%BROWNFGH  Nonlinear minimization problem (function, its gradients
% and Hessian).
% Documentation example.

%   Copyright 1990-2019 The MathWorks, Inc.

% 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
     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
```

```matlab
% Evaluate the (sparse, symmetric) Hessian matrix
if nargout > 2
   v = zeros(n,1);
   i = 1:(n-1);
   v(i) = 2*(x(i+1).^2+1).*((x(i).^2).^(x(i+1).^2))+...
          4*(x(i+1).^2+1).*(x(i+1).^2).*(x(i).^2).*((x(i).^2).^((x(i+1).^2)-1))+...
          2*((x(i+1).^2).^(x(i).^2+1)).*(log(x(i+1).^2));
   v(i) = v(i)+4*(x(i).^2).*((x(i+1).^2).^(x(i).^2+1)).*((log(x(i+1).^2)).^2);
   v(i+1) = v(i+1)+...
            2*(x(i).^2).^(x(i+1).^2+1).*(log(x(i).^2))+...
            4*(x(i+1).^2).*((x(i).^2).^(x(i+1).^2+1)).*((log(x(i).^2)).^2)+...
            2*(x(i).^2+1).*((x(i+1).^2).^(x(i).^2));
   v(i+1) = v(i+1)+4*(x(i).^2+1).*(x(i+1).^2).*(x(i).^2).*((x(i+1).^2).^(x(i).^2-1));
   v0 = v;
   v = zeros(n-1,1);
   v(i) = 4*x(i+1).*x(i).*((x(i).^2).^(x(i+1).^2))+...
          4*x(i+1).*(x(i+1).^2+1).*x(i).*((x(i).^2).^(x(i+1).^2)).*log(x(i).^2);
   v(i) = v(i)+ 4*x(i+1).*x(i).*((x(i+1).^2).^(x(i).^2)).*log(x(i+1).^2);
   v(i) = v(i)+4*x(i).*((x(i+1).^2).^(x(i).^2)).*x(i+1);
   v1 = v;
   i = [(1:n)';(1:(n-1))'];
   j = [(1:n)';(2:n)'];
   s = [v0;2*v1];
   H = sparse(i,j,s,n,n);
   H = (H+H')/2;
end
end
```

See Also

More About

- "Problem-Based Nonlinear Minimization with Linear Constraints" on page 7-19

Minimization with Dense Structured Hessian, Linear Equalities

In this section...

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 `HessianMultiplyFcn` option in "Hessian Multiply Function" on page 16-104. The objective function has the structure

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

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

$$H = \widehat{H} - VV^T.$$

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

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

In this example, the Hessian multiply function needs $\widehat{H}$ 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, $\widehat{H}$ is not a constant and must be computed at the current `x`. You can do this by computing $\widehat{H}$ in the objective function and returning $\widehat{H}$ 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 as well as the objective function, the example (Step 3 on page 6-94) uses `optimoptions` to set the `SpecifyObjectiveGradient` option to `true`.

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 `SpecifyObjectiveGradient` option to `true` and to set the `HessianMultiplyFcn` 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',...
    'SpecifyObjectiveGradient',true, ...
    'HessianMultiplyFcn',@(Hinfo,Y)hmfleq1(Hinfo,Y,V),...
    'Display','iter',...
    'OptimalityTolerance',1e-9,...
    'FunctionTolerance',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 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 =
    1.8874e-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.

Calculate Gradients and Hessians Using Symbolic Math Toolbox™

This example shows how to obtain faster and more robust solutions to nonlinear optimization problems using `fmincon` along with Symbolic Math Toolbox functions. If you have a Symbolic Math Toolbox license, you can easily calculate analytic gradients and Hessians for objective and constraint functions using these Symbolic Math Toolbox functions:

- `jacobian` (Symbolic Math Toolbox) 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. This example shows how to use `jacobian` to generate symbolic gradients and Hessians of objective and constraint functions.
- `matlabFunction` (Symbolic Math Toolbox) 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 functions and their derivatives at arbitrary points.

Symbolic Math Toolbox functions have different syntaxes and structures compared to Optimization Toolbox functions. In particular, symbolic variables are real or complex scalars, whereas Optimization Toolbox™ functions pass vector arguments. So, you need to take several steps to symbolically generate the objective function, constraints, and all their requisite derivatives, in a form suitable for the interior-point algorithm of `fmincon`.

Problem-based optimization can calculate and use gradients automatically; see "Automatic Differentiation in Optimization Toolbox" on page 10-41. For a problem-based approach to this problem using automatic differentiation, see "Constrained Electrostatic Nonlinear Optimization, Problem-Based" on page 7-14.

Problem Description

Consider the electrostatics problem of placing 10 electrons in a conducting body. The electrons arrange themselves in a way that minimizes their total potential energy, subject to the constraint of lying inside the body. All the electrons are on the boundary of the body at a minimum. The electrons are indistinguishable, so no unique minimum exists for this problem (permuting the electrons in one solution gives another valid solution). This example is inspired by Dolan, Moré, and Munson [58].

This example is a conducting body defined by the inequalities

$$z \leq -|x| - |y|$$

$$x^2 + y^2 + (z + 1)^2 \leq 1.$$

This body looks like a pyramid on a sphere.

```
[X,Y] = meshgrid(-1:.01:1);
Z1 = -abs(X) - abs(Y);
Z2 = -1 - sqrt(1 - X.^2 - Y.^2);
Z2 = real(Z2);
W1 = Z1; W2 = Z2;
W1(Z1 < Z2) = nan; % Only plot points where Z1 > Z2
W2(Z1 < Z2) = nan; % Only plot points where Z1 > Z2
hand = figure; % Handle to the figure, for more plotting later
```

```
set(gcf,'Color','w') % White background
surf(X,Y,W1,'LineStyle','none');
hold on
surf(X,Y,W2,'LineStyle','none');
view(-44,18)
```
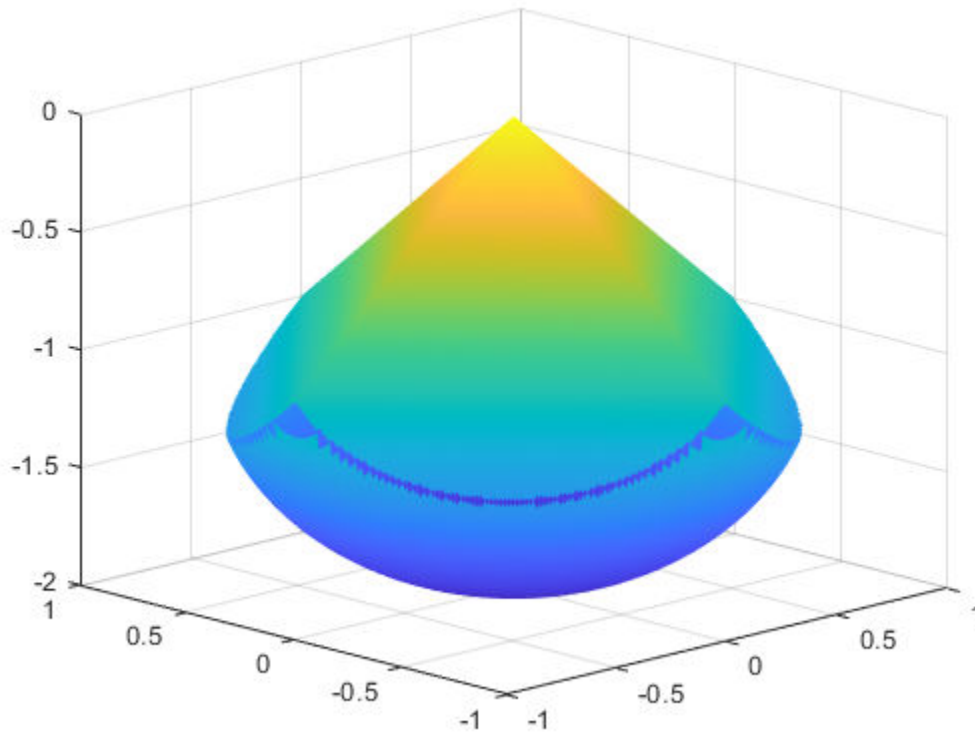

A slight gap exists between the upper and lower surfaces of the figure. This gap is an artifact of the general plotting routine used to create the figure. The routine erases any rectangular patch on one surface that touches the other surface.

Create Variables

Generate a symbolic vector x as a 30-by-1 vector composed of real symbolic variables xij, with 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');
```

Display the vector x.

x

$$x = \begin{pmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{21} \\ x_{22} \\ x_{23} \\ x_{31} \\ x_{32} \\ x_{33} \\ x_{41} \\ x_{42} \\ x_{43} \\ x_{51} \\ x_{52} \\ x_{53} \\ x_{61} \\ x_{62} \\ x_{63} \\ x_{71} \\ x_{72} \\ x_{73} \\ x_{81} \\ x_{82} \\ x_{83} \\ x_{91} \\ x_{92} \\ x_{93} \\ x_{101} \\ x_{102} \\ x_{103} \end{pmatrix}$$

Include Linear Constraints

Write the linear constraints

$$z \leq -|x| - |y|$$

as a set of four linear inequalities for each electron:

```
xi3 - xi1 - xi2 ≤ 0
xi3 - xi1 + xi2 ≤ 0
xi3 + xi1 - xi2 ≤ 0
xi3 + xi1 + xi2 ≤ 0
```

So, this problem has 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 $A*x \leq b$ represents the inequalities.

```
disp(A*x)
```

$$\begin{pmatrix}
x_{11} + x_{12} + x_{13} \\
x_{12} - x_{11} + x_{13} \\
x_{11} - x_{12} + x_{13} \\
x_{13} - x_{12} - x_{11} \\
x_{21} + x_{22} + x_{23} \\
x_{22} - x_{21} + x_{23} \\
x_{21} - x_{22} + x_{23} \\
x_{23} - x_{22} - x_{21} \\
x_{31} + x_{32} + x_{33} \\
x_{32} - x_{31} + x_{33} \\
x_{31} - x_{32} + x_{33} \\
x_{33} - x_{32} - x_{31} \\
x_{41} + x_{42} + x_{43} \\
x_{42} - x_{41} + x_{43} \\
x_{41} - x_{42} + x_{43} \\
x_{43} - x_{42} - x_{41} \\
x_{51} + x_{52} + x_{53} \\
x_{52} - x_{51} + x_{53} \\
x_{51} - x_{52} + x_{53} \\
x_{53} - x_{52} - x_{51} \\
x_{61} + x_{62} + x_{63} \\
x_{62} - x_{61} + x_{63} \\
x_{61} - x_{62} + x_{63} \\
x_{63} - x_{62} - x_{61} \\
x_{71} + x_{72} + x_{73} \\
x_{72} - x_{71} + x_{73} \\
x_{71} - x_{72} + x_{73} \\
x_{73} - x_{72} - x_{71} \\
x_{81} + x_{82} + x_{83} \\
x_{82} - x_{81} + x_{83} \\
x_{81} - x_{82} + x_{83} \\
x_{83} - x_{82} - x_{81} \\
x_{91} + x_{92} + x_{93} \\
x_{92} - x_{91} + x_{93} \\
x_{91} - x_{92} + x_{93} \\
x_{93} - x_{92} - x_{91} \\
x_{101} + x_{102} + x_{103} \\
x_{102} - x_{101} + x_{103} \\
x_{101} - x_{102} + x_{103} \\
x_{103} - x_{102} - x_{101}
\end{pmatrix}$$

Create the Nonlinear Constraints and Their Gradients and Hessians

The nonlinear constraints are also structured.

$$x^2 + y^2 + (z + 1)^2 \le 1.$$

Generate the constraints and their gradients and Hessians.

```
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 form is correct, as described in "Nonlinear Constraints" on page 2-37.

The Hessian matrices, `hessc{1}`, ..., `hessc{10}`, are square and symmetric. This example stores them in a cell array, which is better than storing them in separate variables such as `hessc1`, ..., `hessc10`.

Use the `.'` syntax to transpose. The `'` syntax means conjugate transpose, which has different symbolic derivatives.

Create the Objective Function and Its Gradient and Hessian

The objective function, potential energy, is the sum of the inverses of the distances between each electron pair.

$$\text{energy} = \sum_{i < j} \frac{1}{|x_i - x_j|}.$$

The distance is the square root of the sum of the squares of the differences in the components of the vectors.

Calculate the energy (objective function) and its gradient and Hessian.

```
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 has 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. You can place the resulting files in any folder on the MATLAB path. In this case, place them in your current folder.

```
currdir = [pwd filesep]; % You might need to use currdir = pwd
filename = [currdir,'demoenergy.m'];
matlabFunction(energy,gradenergy,'file',filename,'vars',{x});
```

`matlabFunction` returns `energy` as the first output and `gradenergy` as the second. The function also takes a single input vector `{x}` instead of a list of inputs `x11`, ..., `x103`.

The resulting file `demoenergy.m` contains 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 2-17.

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 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 2-37.

Generate the Hessian Files

To generate the Hessian of the Lagrangian for the problem, first generate files for the energy Hessian and the constraint Hessians.

The Hessian of the objective function, `hessenergy`, is a large symbolic expression containing over 150,000 symbols, as shown by evaluating `size(char(hessenergy))`. So, running `matlabFunction(hessenergy)` takes a substantial amount of time.

Generate the file `hessenergy.m`.

```
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 the helper function `hessfinal.m`, whose code appears at the end of this example on page 6-0   .

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 options to use the `interior-point` algorithm, and to use gradients and the Hessian.

```
options = optimoptions(@fmincon,'Algorithm','interior-point','SpecifyObjectiveGradient',true,...
    'SpecifyConstraintGradient',true,'HessianFcn',@hessfinal,'Display','final');
```

Call `fmincon`.

```
[xfinal fval exitflag output] = fmincon(@demoenergy,Xinitial,...
    A,b,[],[],[],[],@democonstr,options);
```

```
Local minimum found that satisfies the constraints.

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

<stopping criteria details>
```

View the objective function value, exit flag, number of iterations, and number of function evaluations.

```
disp(fval)
```

    34.1365

```
disp(exitflag)
```

     1

```
disp(output.iterations)
```

    19

```
disp(output.funcCount)
```

    28

Even though the initial positions of the electrons are random, the final positions are nearly symmetric.

```
for i = 1:10
    plot3(xfinal(3*i-2),xfinal(3*i-1),xfinal(3*i),'r.','MarkerSize',25);
end
```

To examine the entire 3-D geometry, rotate the figure.

```
rotate3d
```
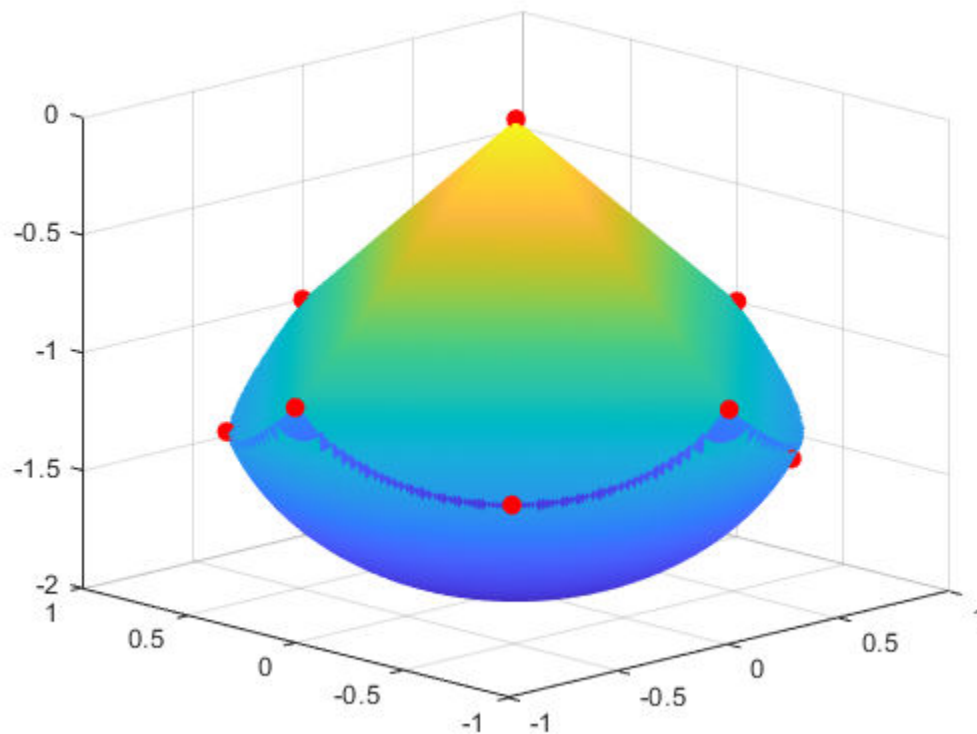


```
figure(hand)
```

Compare to Optimization Without Gradients and Hessians

The use of gradients and Hessians makes the optimization run faster and more accurately. To compare the same optimization using no gradient or Hessian information, set options to not use gradients and Hessians.

```
options = optimoptions(@fmincon,'Algorithm','interior-point',...
    'Display','final');
[xfinal2 fval2 exitflag2 output2] = fmincon(@demoenergy,Xinitial,...
    A,b,[],[],[],[],@democonstr,options);
```

```
Feasible point with lower objective function value found.


Local minimum found that satisfies the constraints.

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

<stopping criteria details>
```

View the objective function value, exit flag, number of iterations, and number of function evaluations.

```
disp(fval2)

    34.1365

disp(exitflag2)

     1

disp(output2.iterations)

    80

disp(output2.funcCount)

       2525
```

Compare the number of iterations and number of function evaluations with and without derivative information.

```
tbl = table([output.iterations;output2.iterations],[output.funcCount;output2.funcCount],...
    'RowNames',{'With Derivatives','Without Derivatives'},'VariableNames',{'Iterations','Fevals'}

tbl=2×2 table
                          Iterations    Fevals
                          _____    _____

    With Derivatives          19          28
    Without Derivatives       80         2525
```

Clear the Symbolic Variable Assumptions

The symbolic variables in this example have the assumption that they are real in the symbolic engine workspace. Deleting the variables does not clear this assumption from the symbolic engine workspace. Clear the variable assumptions by using `syms`.

```
syms x
```

Verify that the assumptions are empty.

```
assumptions(x)
```

```
ans =

Empty sym: 1-by-0
```

Helper Function

The following code creates the `hessfinal` helper function.

```
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);
end
```

See Also

Related Examples

- "Using Symbolic Mathematics with Optimization Toolbox™ Solvers" on page 6-108
- "Constrained Electrostatic Nonlinear Optimization, Problem-Based" on page 7-14

Using Symbolic Mathematics with Optimization Toolbox™ Solvers

This example shows how to use the Symbolic Math Toolbox™ functions `jacobian` and `matlabFunction` to provide analytical derivatives to optimization solvers. Optimization Toolbox™ solvers are usually more accurate and efficient when you supply gradients and Hessians of the objective and constraint functions.

Problem-based optimization can calculate and use gradients automatically; see "Automatic Differentiation in Optimization Toolbox" on page 10-41. For a problem-based example using automatic differentiation, see "Constrained Electrostatic Nonlinear Optimization, Problem-Based" on page 7-14.

There are several considerations in using symbolic calculations with optimization functions:

1   Optimization objective and constraint functions should be defined in terms of a vector, say `x`. However, symbolic variables are scalar or complex-valued, not vector-valued. This requires you to translate between vectors and scalars.

2   Optimization gradients, and sometimes Hessians, are supposed to be calculated within the body of the objective or constraint functions. This means that a symbolic gradient or Hessian has to be placed in the appropriate place in the objective or constraint function file or function handle.

3   Calculating gradients and Hessians symbolically can be time-consuming. Therefore you should perform this calculation only once, and generate code, via `matlabFunction`, to call during execution of the solver.

4   Evaluating symbolic expressions with the `subs` function is time-consuming. It is much more efficient to use `matlabFunction`.

5   `matlabFunction` generates code that depends on the orientation of input vectors. Since `fmincon` calls the objective function with column vectors, you must be careful to call `matlabFunction` with column vectors of symbolic variables.

First Example: Unconstrained Minimization with Hessian

The objective function to minimize is:

$$f(x_1, x_2) = \log\left(1 + 3\left(x_2 - (x_1^3 - x_1)\right)^2 + (x_1 - 4/3)^2\right).$$

This function is positive, with a unique minimum value of zero attained at `x1 = 4/3`, `x2 =(4/3)^3 - 4/3 = 1.0370...`

We write the independent variables as `x1` and `x2` because in this form they can be used as symbolic variables. As components of a vector `x` they would be written `x(1)` and `x(2)`. The function has a twisty valley as depicted in the plot below.

```
syms x1 x2 real
x = [x1;x2]; % column vector of symbolic variables
f = log(1 + 3*(x2 - (x1^3 - x1))^2 + (x1 - 4/3)^2)

f =
```

$$\log\left(\left(x_1 - \frac{4}{3}\right)^2 + 3\left(-x_1^3 + x_1 + x_2\right)^2 + 1\right)$$

```
fsurf(f,[-2 2],'ShowContours','on')
view(127,38)
```
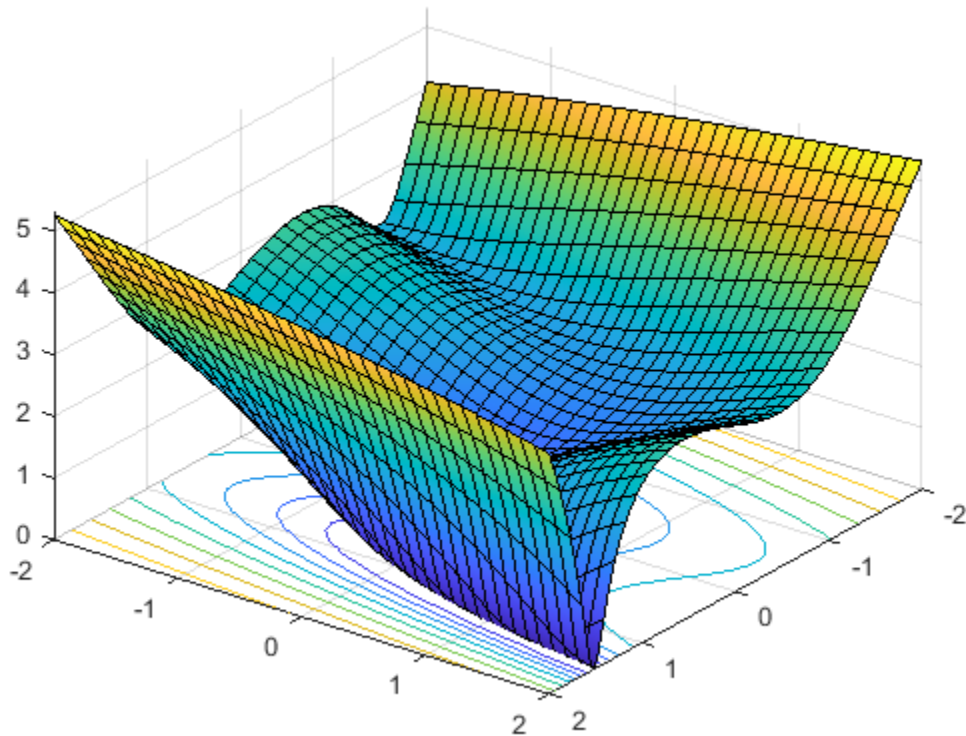

Compute the gradient and Hessian of f:

```
gradf = jacobian(f,x).' % column gradf
```

gradf =

$$\begin{pmatrix} -\dfrac{6\left(3\,x_1^{\,2}-1\right)\left(-x_1^{\,3}+x_1+x_2\right)-2\,x_1+\frac{8}{3}}{\sigma_1} \\[4mm] \dfrac{-6\,x_1^{\,3}+6\,x_1+6\,x_2}{\sigma_1} \end{pmatrix}$$

where

$$\sigma_1 = \left(x_1-\frac{4}{3}\right)^2 + 3\left(-x_1^{\,3}+x_1+x_2\right)^2 + 1$$

```
hessf = jacobian(gradf,x)
```

hessf =

6-109

$$
\begin{pmatrix}
\dfrac{6\left(3x_1{}^2-1\right)^2-36x_1\left(-x_1{}^3+x_1+x_2\right)+2}{\sigma_2}-\dfrac{\sigma_3{}^2}{\sigma_2{}^2} & \sigma_1 \\
\\
\sigma_1 & \dfrac{6}{\sigma_2}-\dfrac{\left(-6x_1{}^3+6x_1+6x_2\right)^2}{\sigma_2{}^2}
\end{pmatrix}
$$

where

$$
\sigma_1 = \frac{\left(-6x_1{}^3+6x_1+6x_2\right)\sigma_3}{\sigma_2{}^2} - \frac{18x_1{}^2-6}{\sigma_2}
$$

$$
\sigma_2 = \left(x_1-\frac{4}{3}\right)^2 + 3\left(-x_1{}^3+x_1+x_2\right)^2 + 1
$$

$$
\sigma_3 = 6\left(3x_1{}^2-1\right)\left(-x_1{}^3+x_1+x_2\right) - 2x_1 + \frac{8}{3}
$$

The `fminunc` solver expects to pass in a vector x, and, with the `SpecifyObjectiveGradient` option set to `true` and `HessianFcn` option set to `'objective'`, expects a list of three outputs: `[f(x),gradf(x),hessf(x)]`.

`matlabFunction` generates exactly this list of three outputs from a list of three inputs. Furthermore, using the `vars` option, `matlabFunction` accepts vector inputs.

```
fh = matlabFunction(f,gradf,hessf,'vars',{x});
```

Now solve the minimization problem starting at the point [-1,2]:

```
options = optimoptions('fminunc', ...
    'SpecifyObjectiveGradient', true, ...
    'HessianFcn', 'objective', ...
    'Algorithm','trust-region', ...
    'Display','final');
[xfinal,fval,exitflag,output] = fminunc(fh,[-1;2],options)
```

```
Local minimum possible.

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

xfinal = 2×1

    1.3333
    1.0370


fval = 7.6623e-12

exitflag = 3

output = struct with fields:
        iterations: 14
         funcCount: 15
          stepsize: 0.0027
      cgiterations: 11
     firstorderopt: 3.4391e-05
         algorithm: 'trust-region'
           message: '...'
```

```
    constrviolation: []
```

Compare this with the number of iterations using no gradient or Hessian information. This requires the `'quasi-newton'` algorithm.

```
options = optimoptions('fminunc','Display','final','Algorithm','quasi-newton');
fh2 = matlabFunction(f,'vars',{x});
% fh2 = objective with no gradient or Hessian
[xfinal,fval,exitflag,output2] = fminunc(fh2,[-1;2],options)
```

```
Local minimum found.

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

xfinal = 2×1

    1.3333
    1.0371


fval = 2.1985e-11

exitflag = 1

output2 = struct with fields:
      iterations: 18
       funcCount: 81
        stepsize: 2.1164e-04
    lssteplength: 1
    firstorderopt: 2.4587e-06
       algorithm: 'quasi-newton'
         message: '...'
```

The number of iterations is lower when using gradients and Hessians, and there are dramatically fewer function evaluations:

```
sprintf(['There were %d iterations using gradient' ...
    ' and Hessian, but %d without them.'], ...
    output.iterations,output2.iterations)
```

```
ans =
'There were 14 iterations using gradient and Hessian, but 18 without them.'
```

```
sprintf(['There were %d function evaluations using gradient' ...
    ' and Hessian, but %d without them.'], ...
    output.funcCount,output2.funcCount)
```

```
ans =
'There were 15 function evaluations using gradient and Hessian, but 81 without them.'
```

Second Example: Constrained Minimization Using the fmincon Interior-Point Algorithm

We consider the same objective function and starting point, but now have two nonlinear constraints:

$$5\sinh(x_2/5) \geq x_1^4$$

$$5\tanh(x_1/5) \geq x_2^2 - 1.$$

The constraints keep the optimization away from the global minimum point [1.333,1.037]. Visualize the two constraints:

```
[X,Y] = meshgrid(-2:.01:3);
Z = (5*sinh(Y./5) >= X.^4);
% Z=1 where the first constraint is satisfied, Z=0 otherwise
Z = Z+ 2*(5*tanh(X./5) >= Y.^2 - 1);
% Z=2 where the second is satisfied, Z=3 where both are
surf(X,Y,Z,'LineStyle','none');
fig = gcf;
fig.Color = 'w'; % white background
view(0,90)
hold on
plot3(.4396, .0373, 4,'o','MarkerEdgeColor','r','MarkerSize',8);
% best point
xlabel('x')
ylabel('y')
hold off
```
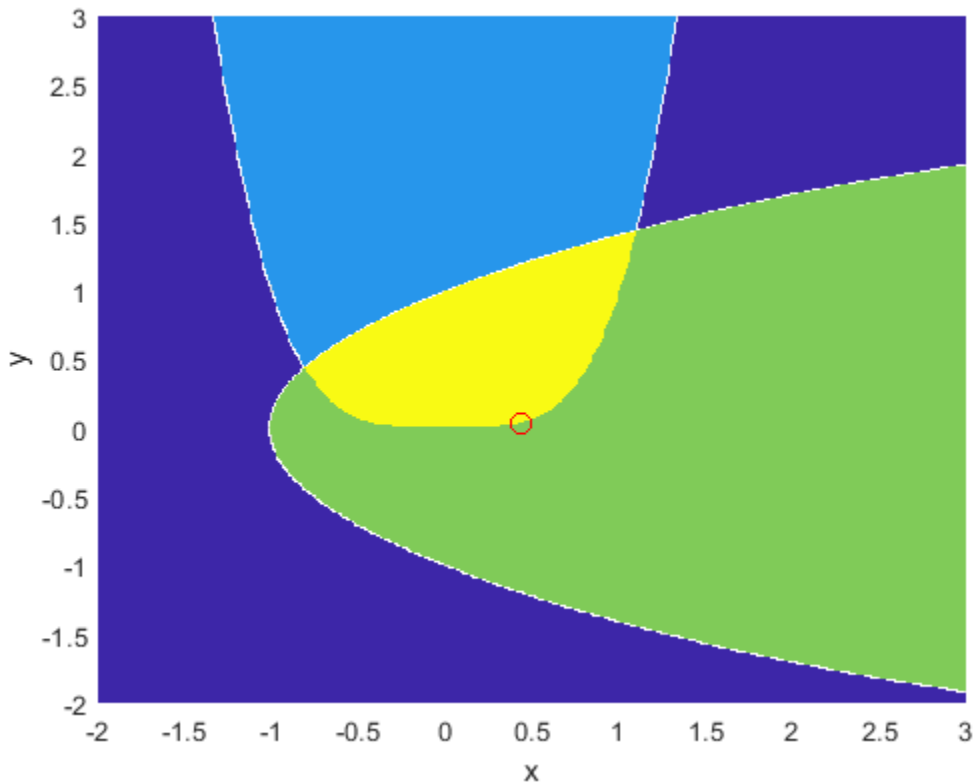

We plotted a small red circle around the optimal point.

Here is a plot of the objective function over the feasible region, the region that satisfies both constraints, pictured above in dark red, along with a small red circle around the optimal point:

```
W = log(1 + 3*(Y - (X.^3 - X)).^2 + (X - 4/3).^2);
% W = the objective function
```

```
W(Z < 3) = nan; % plot only where the constraints are satisfied
surf(X,Y,W,'LineStyle','none');
view(68,20)
hold on
plot3(.4396, .0373, .8152,'o','MarkerEdgeColor','r', ...
    'MarkerSize',8); % best point
xlabel('x')
ylabel('y')
zlabel('z')
hold off
```
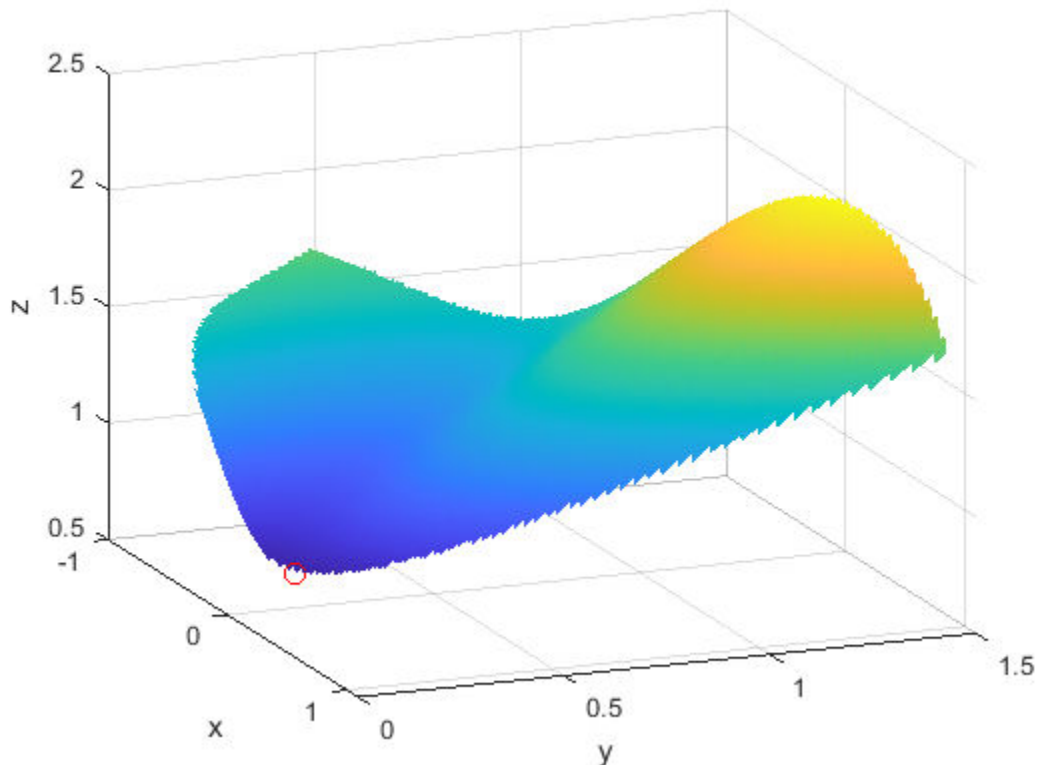

The nonlinear constraints must be written in the form `c(x) <= 0`. We compute all the symbolic constraints and their derivatives, and place them in a function handle using `matlabFunction`.

The gradients of the constraints should be column vectors; they must be placed in the objective function as a matrix, with each column of the matrix representing the gradient of one constraint function. This is the transpose of the form generated by `jacobian`, so we take the transpose below.

We place the nonlinear constraints into a function handle. `fmincon` expects the nonlinear constraints and gradients to be output in the order `[c ceq gradc gradceq]`. Since there are no nonlinear equality constraints, we output `[]` for `ceq` and `gradceq`.

```
c1 = x1^4 - 5*sinh(x2/5);
c2 = x2^2 - 5*tanh(x1/5) - 1;
c = [c1 c2];
gradc = jacobian(c,x).'; % transpose to put in correct form
constraint = matlabFunction(c,[],gradc,[],'vars',{x});
```

The interior-point algorithm requires its Hessian function to be written as a separate function, instead of being part of the objective function. This is because a nonlinearly constrained function needs to include those constraints in its Hessian. Its Hessian is the Hessian of the Lagrangian; see the User's Guide for more information.

The Hessian function takes two input arguments: the position vector x, and the Lagrange multiplier structure lambda. The parts of the lambda structure that you use for nonlinear constraints are lambda.ineqnonlin and lambda.eqnonlin. For the current constraint, there are no linear equalities, so we use the two multipliers lambda.ineqnonlin(1) and lambda.ineqnonlin(2).

We calculated the Hessian of the objective function in the first example. Now we calculate the Hessians of the two constraint functions, and make function handle versions with matlabFunction.

```
hessc1 = jacobian(gradc(:,1),x); % constraint = first c column
hessc2 = jacobian(gradc(:,2),x);

hessfh = matlabFunction(hessf,'vars',{x});
hessc1h = matlabFunction(hessc1,'vars',{x});
hessc2h = matlabFunction(hessc2,'vars',{x});
```

To make the final Hessian, we put the three Hessians together, adding the appropriate Lagrange multipliers to the constraint functions.

```
myhess = @(x,lambda)(hessfh(x) + ...
    lambda.ineqnonlin(1)*hessc1h(x) + ...
    lambda.ineqnonlin(2)*hessc2h(x));
```

Set the options to use the interior-point algorithm, the gradient, and the Hessian, have the objective function return both the objective and the gradient, and run the solver:

```
options = optimoptions('fmincon', ...
    'Algorithm','interior-point', ...
    'SpecifyObjectiveGradient',true, ...
    'SpecifyConstraintGradient',true, ...
    'HessianFcn',myhess, ...
    'Display','final');
% fh2 = objective without Hessian
fh2 = matlabFunction(f,gradf,'vars',{x});
[xfinal,fval,exitflag,output] = fmincon(fh2,[-1;2],...
    [],[],[],[],[],[],constraint,options)

Local minimum found that satisfies the constraints.

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

xfinal = 2×1

    0.4396
    0.0373


fval = 0.8152

exitflag = 1

output = struct with fields:
        iterations: 10
```

```
            funcCount: 13
    constrviolation: 0
            stepsize: 1.9160e-06
           algorithm: 'interior-point'
        firstorderopt: 1.9217e-08
         cgiterations: 0
              message: '...'
         bestfeasible: [1x1 struct]
```

Again, the solver makes many fewer iterations and function evaluations with gradient and Hessian supplied than when they are not:

```
options = optimoptions('fmincon','Algorithm','interior-point',...
    'Display','final');
% fh3 = objective without gradient or Hessian
fh3 = matlabFunction(f,'vars',{x});
% constraint without gradient:
constraint = matlabFunction(c,[],'vars',{x});
[xfinal,fval,exitflag,output2] = fmincon(fh3,[-1;2],...
    [],[],[],[],[],[],constraint,options)
```

```
Local minimum found that satisfies the constraints.

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

xfinal = 2×1

    0.4396
    0.0373


fval = 0.8152

exitflag = 1

output2 = struct with fields:
          iterations: 17
           funcCount: 54
      constrviolation: 0
            stepsize: 8.6490e-07
           algorithm: 'interior-point'
        firstorderopt: 3.8841e-07
         cgiterations: 0
              message: '...'
         bestfeasible: [1x1 struct]
```

```
sprintf(['There were %d iterations using gradient' ...
    ' and Hessian, but %d without them.'],...
    output.iterations,output2.iterations)
```

```
ans =
'There were 10 iterations using gradient and Hessian, but 17 without them.'
```

```
sprintf(['There were %d function evaluations using gradient' ...
    ' and Hessian, but %d without them.'], ...
    output.funcCount,output2.funcCount)

ans =
'There were 13 function evaluations using gradient and Hessian, but 54 without them.'
```

Cleaning Up Symbolic Variables

The symbolic variables used in this example were assumed to be real. To clear this assumption from the symbolic engine workspace, it is not sufficient to delete the variables. You must clear the assumptions of variables using the syntax

```
assume([x1,x2],'clear')
```

All assumptions are cleared when the output of the following command is empty

```
assumptions([x1,x2])

ans =

Empty sym: 1-by-0
```

See Also

More About

- "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97

Obtain Best Feasible Point

This example shows how to obtain the best feasible point encountered by `fmincon`.

The helper function `bigtoleft` is a cubic polynomial objective function in a three-dimensional variable `x` that grows rapidly negative as the `x(1)` coordinate becomes negative. Its gradient is a three-element vector. The code for the `bigtoleft` helper function appears at the end of this example on page 6-0 .

The constraint set for this example is the intersection of the interiors of two cones—one pointing up, and one pointing down. The constraint function is a two-component vector containing one component for each cone. Because this example is three-dimensional, the gradient of the constraint is a 3-by-2 matrix. The code for the `twocone` helper function appears at the end of this example on page 6-0 .

Create a figure of the constraints colored using the objective function by calling the `plottwoconecons` function, whose code appears at the end of this example on page 6-0 .

```
figure1 = plottwoconecons;
```
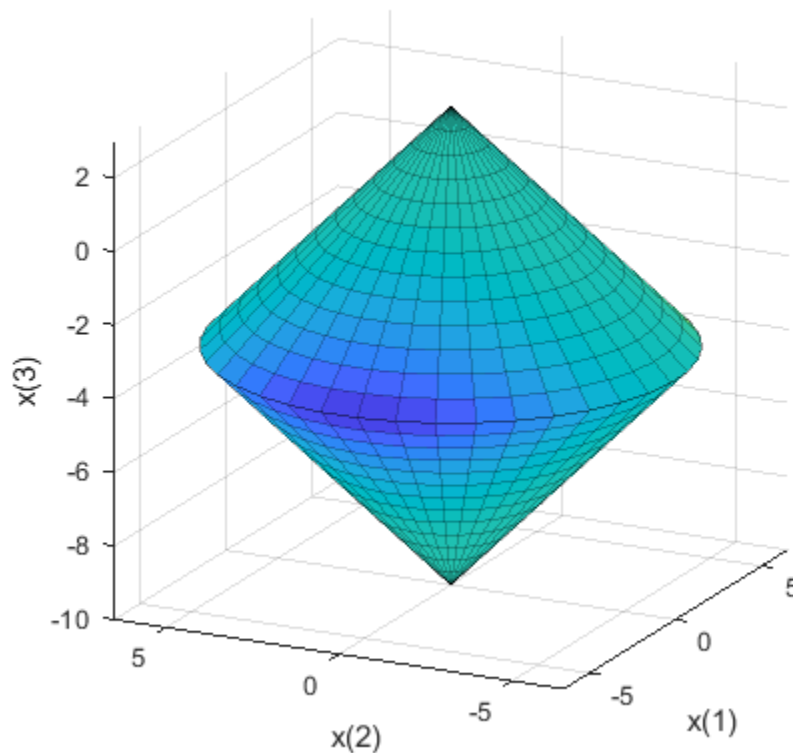

Create Options To Use Derivatives

To enable `fmincon` to use the objective gradient and constraint gradients, set appropriate options. Choose the `'sqp'` algorithm.

```
options = optimoptions('fmincon','Algorithm','sqp',...
    "SpecifyConstraintGradient",true,"SpecifyObjectiveGradient",true);
```

To examine the solution process, set options to return iterative display.

```
options.Display = 'iter';
```

Minimize Using Derivative Information

Set the initial point x0 = [-1/20,-1/20,-1/20].

```
x0 = -1/20*ones(1,3);
```

The problem has no linear constraints or bounds. Set those arguments to [].

```
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
```

Solve the problem.

```
[x,fval,eflag,output] = fmincon(@bigtoleft,x0,...
        A,b,Aeq,beq,lb,ub,@twocone,options);
```

Iter	Func-count	Fval	Feasibility	Step Length	Norm of step	First-order optimality
0	1	-1.625000e-03	0.000e+00	1.000e+00	0.000e+00	8.250e-02
1	3	-2.490263e-02	0.000e+00	1.000e+00	8.325e-02	5.449e-01
2	5	-2.529919e+02	0.000e+00	1.000e+00	2.802e+00	2.585e+02
3	7	-6.408576e+03	9.472e+00	1.000e+00	1.538e+01	1.771e+03
4	9	-1.743599e+06	5.301e+01	1.000e+00	5.991e+01	9.216e+04
5	11	-5.552305e+09	1.893e+03	1.000e+00	1.900e+03	1.761e+07
6	13	-1.462524e+15	5.632e+04	1.000e+00	5.636e+04	8.284e+10
7	15	-2.573346e+24	1.471e+08	1.000e+00	1.471e+08	1.058e+17
8	17	-1.467510e+41	2.617e+13	1.000e+00	2.617e+13	1.789e+28
9	19	-8.716877e+72	2.210e+24	1.000e+00	2.210e+24	2.387e+49
10	21	-5.598248e+134	4.090e+44	1.000e+00	4.090e+44	4.368e+90
11	23	-5.691634e+258	1.355e+86	1.000e+00	1.136e+86	1.967e+173
12	25	-4.518887e+258	8.636e+85	1.000e+00	7.539e+85	1.766e+173
13	70	-4.518887e+258	8.636e+85	1.070e-07	3.756e+78	1.766e+173

```
Feasible point with lower objective function value found.


Converged to an infeasible point.

fmincon stopped because the size of the current step is less than
the value of the step size tolerance but constraints are not
satisfied to within the value of the constraint tolerance.
```

Examine Solution and Solution Process

Examine the solution, objective function value, exit flag, and number of function evaluations and iterations.

```
disp(x)
```

```
   1.0e+85 *

  -7.6403    0.4014   -0.9857
```

```
disp(fval)
```

```
 -4.5189e+258
```

```
disp(eflag)
```

```
    -2
```

```
disp(output.constrviolation)
```

```
   8.6365e+85
```

The objective function value is smaller than –1e250, a very negative value. The constraint violation is larger than 1e85, a large amount that is still much smaller in magnitude than the objective function value. The exit flag also shows that the returned solution is infeasible.

To recover the best feasible point that `fmincon` encounters, along with its objective function value, display the `output.bestfeasible` structure.

```
disp(output.bestfeasible)
```

```
               x: [-2.9297 -0.1813 -0.1652]
            fval: -252.9919
   constrviolation: 0
     firstorderopt: 258.5032
```

The `bestfeasible` point is not a good solution, as you see in the next section. It is simply the best feasible point that `fmincon` encountered in its iterations. In this case, even though `bestfeasible` is not a solution, it is a better point than the returned infeasible solution.

```
table([fval;output.bestfeasible.fval],...
    [output.constrviolation;output.bestfeasible.constrviolation],...
    'VariableNames',["Fval" "Constraint Violation"],'RowNames',["Final Point" "Best Feasible"])
```

ans=2×2 table

	Fval	Constraint Violation
Final Point	-4.5189e+258	8.6365e+85
Best Feasible	-252.99	0

Improve Solution: Set Bounds

There are several ways to obtain a feasible solution. One way is to set bounds on the variables.

```
lb = -10*ones(3,1);
ub = -lb;
[xb,fvalb,eflagb,outputb] = fmincon(@bigtoleft,x0,...
          A,b,Aeq,beq,lb,ub,@twocone,options);
```

Iter	Func-count	Fval	Feasibility	Step Length	Norm of step	First-order optimality
0	1	-1.625000e-03	0.000e+00	1.000e+00	0.000e+00	8.250e-02
1	3	-2.490263e-02	0.000e+00	1.000e+00	8.325e-02	5.449e-01

```
2          5    -2.529919e+02    0.000e+00    1.000e+00    2.802e+00    2.585e+02
3          7    -4.867942e+03    5.782e+00    1.000e+00    1.151e+01    1.525e+03
4          9    -1.035980e+04    3.536e+00    1.000e+00    9.587e+00    1.387e+03
5         12    -5.270039e+03    2.143e+00    7.000e-01    4.865e+00    2.804e+02
6         14    -2.538947e+03    2.855e-02    1.000e+00    2.229e+00    1.715e+03
7         16    -2.703320e+03    2.330e-02    1.000e+00    5.517e-01    2.521e+02
8         19    -2.845099e+03    0.000e+00    1.000e+00    1.752e+00    8.873e+01
9         21    -2.896934e+03    2.150e-03    1.000e+00    1.709e-01    1.608e+01
10        23    -2.894135e+03    7.954e-06    1.000e+00    1.039e-02    2.028e+00
11        25    -2.894126e+03    4.113e-07    1.000e+00    2.312e-03    2.100e-01
12        27    -2.894125e+03    4.619e-09    1.000e+00    2.450e-04    1.471e-04

Feasible point with lower objective function value found.


Local minimum found that satisfies the constraints.

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

The iterative display shows that `fmincon` starts at a feasible point (feasibility 0), spends a few iterations infeasible, again reaches 0 feasibility, then has small but nonzero infeasibility for the remaining iterations. The solver again reports that it found a lower feasible value at a point other than the final point `xb`. View the final point and objective function value, and the reported feasible point with lower objective function value.

```
disp(xb)

   -6.5000   -0.0000   -3.5000

disp(fvalb)

  -2.8941e+03

disp(outputb.bestfeasible)

                x: [-6.5000 2.4500e-04 -3.5000]
             fval: -2.8941e+03
    constrviolation: 4.1127e-07
      firstorderopt: 0.2100
```

The constraint violation at the `bestfeasible` point is about `4.113e-7`. In the iterative display, this infeasibiliity occurs at iteration 11. The reported objective function value at that iteration is `-2.894126e3`, which is slightly less than the final value of `-2.894125e3`. The final point has lower infeasibility and first-order optimality measure. Which point is better? They are nearly the same, but each point has some claim to being better. To see the solution details, set the display format to `long`.

```
format long
table([fvalb;outputb.bestfeasible.fval],...
    [outputb.constrviolation;outputb.bestfeasible.constrviolation],...
    [outputb.firstorderopt;outputb.bestfeasible.firstorderopt],...
    'VariableNames',["Function Value" "Constraint Violation" "First-Order Optimality"],...
    'RowNames',["Final Point" "Best Feasible"])

ans=2×3 table
                    Function Value      Constraint Violation      First-Order Optimality
                    _____      _____      _____
```

```
      Final Point        -2894.12500606462     4.61890525826902e-09     0.000147112019817541
      Best Feasible      -2894.1255345325      4.11267924604886e-07     0.21002297099706
```

Improve Solution: Use Another Algorithm

Another way to obtain a feasible solution is to use the `'interior-point'` algorithm, even without bounds,

```
lb = [];
ub = [];
options.Algorithm = "interior-point";
[xip,fvalip,eflagip,outputip] = fmincon(@bigtoleft,x0,...
          A,b,Aeq,beq,lb,ub,@twocone,options);
```

```
                                              First-order      Norm of
 Iter F-count            f(x)    Feasibility    optimality        step
    0       1   -1.625000e-03    0.000e+00      7.807e-02
    1       2   -2.374253e-02    0.000e+00      5.222e-01    8.101e-02
    2       3   -2.232989e+02    0.000e+00      2.379e+02    2.684e+00
    3       4   -3.838433e+02    1.768e-01      3.198e+02    5.573e-01
    4       5   -3.115565e+03    1.810e-01      1.028e+03    4.660e+00
    5       6   -3.143463e+03    2.013e-01      8.965e+01    5.734e-01
    6       7   -2.917730e+03    1.795e-02      6.140e+01    5.231e-01
    7       8   -2.894095e+03    0.000e+00      9.206e+00    1.821e-02
    8       9   -2.894107e+03    0.000e+00      2.500e+00    3.899e-03
    9      10   -2.894142e+03    1.299e-05      2.136e-03    1.371e-02
   10      11   -2.894125e+03    3.614e-08      4.070e-03    1.739e-05
   11      12   -2.894125e+03    0.000e+00      5.994e-06    5.832e-08
```

```
Feasible point with lower objective function value found.


Local minimum found that satisfies the constraints.

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

The iterative display again shows `fmincon` reaching points that are infeasible in its search for a solution, and `fmincon` again issues a message that it encountered a feasible point with lower objective function value.

```
       disp(xip)
```

```
  -6.499999996950366   -0.000000032933161   -3.500000000098132
```

```
disp(fvalip)
```

```
   -2.89412499599976e+03
```

```
disp(outputip.bestfeasible)
```

```
               x: [1x3 double]
            fval: -2.894125047137580e+03
   constrviolation: 3.613823373882497e-08
     firstorderopt: 0.004069724065403
```

Again, the two solutions are nearly the same, and the `bestfeasible` solution comes from the iteration before the end. The final solution `xip` has better first-order optimality measure and feasibility, but the `bestfeasible` solution has slightly lower objective function value and an infeasibility that it not too large.

```
table([fvalip;outputip.bestfeasible.fval],...
    [outputip.constrviolation;outputip.bestfeasible.constrviolation],...
    [outputip.firstorderopt;outputip.bestfeasible.firstorderopt],...
    'VariableNames',["Function Value" "Constraint Violation" "First-Order Optimality"],...
    'RowNames',["Final Point" "Best Feasible"])
```

ans=2×3 table

	Function Value	Constraint Violation	First-Order Optimality
Final Point	-2894.12499599998	0	5.99383541448297e-06
Best Feasible	-2894.12504713758	3.6138233738825e-08	0.00406972406540262

Finally, reset the format to the default `short`.

```
format short
```

Helper Functions

This code creates the `bigtoleft` helper function.

```
function [f gradf] = bigtoleft(x)
% This is a simple function that grows rapidly negative
% as x(1) becomes 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
end
```

This code creates the `twocone` helper function.

```
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
end
```

This code creates the function that plots the nonlinear constraints.

```
function figure1 = plottwoconecons
% Create figure
figure1 = figure;

% Create axes
axes1 = axes('Parent',figure1);
view([-63.5 18]);
grid('on');
hold('all');

% Set up polar coordinates and two cones
r = linspace(0,6.5,14);
th = 2*pi*linspace(0,1,40);
x = r'*cos(th);
y = r'*sin(th);
z = -10+sqrt(x.^2+y.^2);
zz = 3-sqrt(x.^2+y.^2);

% Evaluate objective function on cone surfaces
newxf = reshape(bigtoleft([x(:),y(:),z(:)]),14,40)/3000;
newxg = reshape(bigtoleft([x(:),y(:),z(:)]),14,40)/3000;

% Create lower surf with color set by objective
surf(x,y,z,newxf,'Parent',axes1,'EdgeAlpha',0.25);

% Create upper surf with color set by objective
surf(x,y,zz,newxg,'Parent',axes1,'EdgeAlpha',0.25);
axis equal
xlabel 'x(1)'
ylabel 'x(2)'
zlabel 'x(3)'
end
```

See Also

`fmincon`

More About

- "When the Solver Might Have Succeeded" on page 4-12

Code Generation in fmincon Background

In this section...
"What Is Code Generation?" on page 6-124
"Code Generation Requirements" on page 6-124
"Generated Code Not Multithreaded" on page 6-125

What Is Code Generation?

Code generation is the conversion of MATLAB code to C code using MATLAB Coder™. Code generation requires a MATLAB Coder license.

Typically, you use code generation to deploy code on hardware that is not running MATLAB. For example, you can deploy code on a robot, using `fmincon` for optimizing movement or planning.

For an example, see "Generate Code for fmincon" on page 6-127. For code generation in other optimization solvers, see "Generate Code for fsolve" on page 13-38, "Generate Code for quadprog" on page 11-60, or "Generate Code for lsqcurvefit or lsqnonlin" on page 12-89.

Code Generation Requirements

- `fmincon` supports code generation using either the `codegen` function or the MATLAB Coder app. You must have a MATLAB Coder license to generate code.
- The target hardware must support standard double-precision floating-point computations. You cannot generate code for single-precision or fixed-point computations.
- Code generation targets do not use the same math kernel libraries as MATLAB solvers. Therefore, code generation solutions can vary from solver solutions, especially for poorly conditioned problems.
- All code for generation must be MATLAB code. In particular, you cannot use a custom black-box function as an objective function for `fmincon`. You can use `coder.ceval` to evaluate a custom function coded in C or C++. However, the custom function must be called in a MATLAB function.
- `fmincon` does not support the `problem` argument for code generation.

  ```
  [x,fval] = fmincon(problem) % Not supported
  ```
- You must specify the objective function and any nonlinear constraint function by using function handles, not strings or character names.

  ```
  x = fmincon(@fun,x0,A,b,Aeq,beq,lb,ub,@nonlcon) % Supported
  % Not supported: fmincon('fun',...) or fmincon("fun",...)
  ```
- All `fmincon` input matrices such as `A`, `Aeq`, `lb`, and `ub` must be full, not sparse. You can convert sparse matrices to full by using the `full` function.
- The `lb` and `ub` arguments must have the same number of entries as the `x0` argument or must be empty `[]`.
- For advanced code optimization involving embedded processors, you also need an Embedded Coder® license.
- You must include options for `fmincon` and specify them using `optimoptions`. The options must include the `Algorithm` option, set to `'sqp'` or `'sqp-legacy'`.

```matlab
options = optimoptions('fmincon','Algorithm','sqp');
[x,fval,exitflag] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options);
```

- Code generation supports these options:

  - `Algorithm` — Must be `'sqp'` or `'sqp-legacy'`
  - `ConstraintTolerance`
  - `FiniteDifferenceStepSize`
  - `FiniteDifferenceType`
  - `MaxFunctionEvaluations`
  - `MaxIterations`
  - `ObjectiveLimit`
  - `OptimalityTolerance`
  - `ScaleProblem`
  - `SpecifyConstraintGradient`
  - `SpecifyObjectiveGradient`
  - `StepTolerance`
  - `TypicalX`

- Generated code has limited error checking for options. The recommended way to update an option is to use `optimoptions`, not dot notation.

  ```matlab
  opts = optimoptions('fmincon','Algorithm','sqp');
  opts = optimoptions(opts,'MaxIterations',1e4); % Recommended
  opts.MaxIterations = 1e4; % Not recommended
  ```

- Do not load options from a file. Doing so can cause code generation to fail. Instead, create options in your code.

- Usually, if you specify an option that is not supported, the option is silently ignored during code generation. However, if you specify a plot function or output function by using dot notation, code generation can issue an error. For reliability, specify only supported options.

- Because output functions and plot functions are not supported, `fmincon` does not return the exit flag –1.

- Code generated from `fmincon` does not contain the `bestfeasible` field in a returned `output` structure.

Generated Code Not Multithreaded

By default, generated code for use outside the MATLAB environment uses linear algebra libraries that are not multithreaded. Therefore, this code can run significantly slower than code in the MATLAB environment.

If your target hardware has multiple cores, you can achieve better performance by using custom multithreaded LAPACK and BLAS libraries. To incorporate these libraries in your generated code, see "Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls" (MATLAB Coder).

See Also
codegen | fmincon | optimoptions

More About

- "Code Generation for Optimization Basics" on page 6-127
- "Static Memory Allocation for fmincon Code Generation" on page 6-131
- "Optimization Code Generation for Real-Time Applications" on page 6-133

Code Generation for Optimization Basics

Generate Code for fmincon

This example shows how to generate code for the fmincon optimization solver. Code generation requires a MATLAB Coder license. For details of code generation requirements, see "Code Generation in fmincon Background" on page 6-124.

The example uses the following simple objective function. To use this objective function in your own testing, copy the code to a file named rosenbrockwithgrad.m. Save the file on your MATLAB path.

```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
```

To generate code using the rosenbrockwithgrad objective function, create a file named test_rosen.m containing this code:

```matlab
function [x,fval] = test_rosen
opts = optimoptions('fmincon','Algorithm','sqp');
[x fval] = fmincon(@rosenbrockwithgrad,[-1,1],[],[],[],[],[-3,-3],[3,3],[],opts)
```

Generate code for the test_rosen file.

```matlab
codegen -config:mex test_rosen
```

After some time, codegen creates a MEX file named test_rosen_mex.mexw64 (the file extension will vary, depending on your system). You can run the resulting C code by entering test_rosen_mex. The results are the following or similar:

```
x =

    1.0000    1.0000


fval =

    1.3346e-11


ans =

    1.0000    1.0000
```

Modify Example for Efficiency

Following some of the suggestions in "Optimization Code Generation for Real-Time Applications" on page 6-133, set the configuration of the generated code to have fewer checks and to use static memory allocation.

```
cfg = coder.config('mex');
cfg.IntegrityChecks = false;
cfg.SaturateOnIntegerOverflow = false;
cfg.DynamicMemoryAllocation = 'Off';
```

Tighten the bounds on the problem from `[-3,3]` to `[-2,2]`. Also, set a looser optimality tolerance than the default `1e-6`.

```
function [x,fval] = test_rosen2
opts = optimoptions('fmincon','Algorithm','sqp',...
    'OptimalityTolerance',1e-5);
[x fval eflag output] = fmincon(@rosenbrockwithgrad,[-1,1],[],[],[],[],...
    [-2,-2],[2,2],[],opts)
```

Generate code for the `test_rosen2` file.

```
codegen -config cfg test_rosen2
```

Run the resulting code.

```
test_rosen2_mex

x =

    1.0000    1.0000


fval =

   2.0057e-11


eflag =

    2


output =

  struct with fields:

          iterations: 40
           funcCount: 155
           algorithm: 'sqp'
       constrviolation: 0
            stepsize: 5.9344e-08
        lssteplength: 1


ans =

    1.0000    1.0000
```

This solution is almost as good as the previous solution, with the `fval` output around `2e-11` compared to the previous `1e-11`.

Try limiting the number of allowed iterations to half of those taken in the previous computation.

```
function [x,fval] = test_rosen3
options = optimoptions('fmincon','Algorithm','sqp',...
    'MaxIterations',20);
[x fval eflag output] = fmincon(@rosenbrockwithgrad,[-1,1],[],[],[],[],...
    [-2,-2],[2,2],[],options)
```

Run `test_rosen3` in MATLAB.

```
test_rosen3

x =

    0.2852    0.0716


fval =

    0.5204


eflag =

     0


output =

  struct with fields:

        iterations: 20
         funcCount: 91
         algorithm: 'sqp'
           message: '↵Solver stopped prematurely.↵↵fmincon stopped because it exceeded the itera
     constrviolation: 0
          stepsize: 0.0225
      lssteplength: 1
      firstorderopt: 1.9504


ans =

    0.2852    0.0716
```

With this severe iteration limit, `fmincon` does not reach a good solution. The tradeoff between accuracy and speed can be difficult to manage.

To save function evaluations and possibly increase accuracy, use the built-in derivatives of the example by setting the `SpecifyObjectiveGradient` option to `true`.

```
function [x,fval] = test_rosen4
options = optimoptions('fmincon','Algorithm','sqp',...
    'SpecifyObjectiveGradient',true);
[x fval eflag output] = fmincon(@rosenbrockwithgrad,[-1,1],[],[],[],[],...
    [-2,-2],[2,2],[],options)
```

Generate code for `test_rosen4` using the same configuration as in `test_rosen2`.

```
codegen -config cfg test_rosen4
```

Run the resulting code.

```
test_rosen4_mex

x =

    1.0000    1.0000


fval =

    3.3610e-20


eflag =

      2


output =

  struct with fields:

         iterations: 40
          funcCount: 113
          algorithm: 'sqp'
      constrviolation: 0
           stepsize: 9.6356e-08
       lssteplength: 1


ans =

    1.0000    1.0000
```

Compared to `test_rosen2`, the number of iterations is the same at 40, but the number of function evaluations is lower at `113` instead of `155`. The result has a better (lower) objective function value of `3e-20` compared to `2e-11`.

See Also
`codegen` | `fmincon` | `optimoptions`

More About
* "Code Generation in fmincon Background" on page 6-124
* "Code Generation for quadprog Background" on page 11-58
* "Static Memory Allocation for fmincon Code Generation" on page 6-131
* "Optimization Code Generation for Real-Time Applications" on page 6-133

Static Memory Allocation for fmincon Code Generation

This example shows how to use static memory allocation in code generation even when some matrix sizes change during a computation.

The problem is a simple nonlinear minimization with both a nonlinear constraint function and linear constraints. The sizes of the linear constraint matrices change at each iteration, which causes the memory requirements to increase at each iteration. The example shows how to use the `coder.varsize` command to set the appropriate variable sizes for static memory allocation.

The `nlp_for_loop.m` file contains the objective function, linear constraints, and nonlinear constraint function. Copy the following code to create this file on your MATLAB path.

```matlab
function nlp_for_loop
% Driver for an example fmincon use case.  Adding constraints increases the
% minimum and uses more memory.

maxIneq = 4; % Number of linear inequality constraints
nVar = 5; % Number of problem variables x

A = zeros(0,nVar);
b = zeros(0,1);

% The next step is required for static memory support. Because you concatenate
% constraints in a "for" loop, you need to limit the dimensions of the
% constraint matrices.
%coder.varsize('var name', [maxRows, maxCols], [canRowsChange, canColsChange]);
coder.varsize('A',[maxIneq,nVar],[true,false]);
coder.varsize('b',[maxIneq,1],[true,false]);

Aeq = [1,0,0,0,1];
beq = 0;
lb = [];
ub = [];

% Initial point
x0 = [2;-3;0;0;-2];

options = optimoptions('fmincon','Algorithm','sqp','Display','none');
for idx = 1:maxIneq
    % Add a new linear inequality constraint at each iteration
    A = [A; circshift([1,1,0,0,0],idx-1)];
    b = [b; -1];

    [x,fval,exitflag] = fmincon(@rosenbrock_nd,x0,A,b,Aeq,beq,...
        lb,ub,@circleconstr,options);
    % Set initial point to found point
    x0 = x;
    % Print fval, ensuring that the datatypes are consistent with the
    % corresponding fprintf format specifiers
    fprintf('%i Inequality Constraints; fval: %f; Exitflag: %i \n',...
        int32(numel(b)),fval,int32(exitflag));
end

end
```

```
function fval = rosenbrock_nd(x)
fval = 100*sum((x(2:end)-x(1:end-1).^2).^2 + (1-x(1:end-1)).^2);
end


function [c,ceq] = circleconstr(x)

radius = 2;
ceq = [];
c = sum(x.^2) - radius^2;

end
```

To generate code from this file using static memory allocation, set the coder configuration as follows.

```
cfg = coder.config('mex');
cfg.DynamicMemoryAllocation = 'Off'; % No dynamic memory allocation
cfg.SaturateOnIntegerOverflow = false; % No MATLAB integer saturation checking
cfg.IntegrityChecks = false; % No checking for out-of-bounds access in arrays
```

Generate code for the `nlp_for_loop.m` file.

```
codegen -config cfg nlp_for_loop
```

Run the resulting MEX file.

```
nlp_for_loop_mex
```

```
1 Inequality Constraints; fval: 542.688894; Exitflag: 1
2 Inequality Constraints; fval: 793.225322; Exitflag: 1
3 Inequality Constraints; fval: 1072.945843; Exitflag: 1
4 Inequality Constraints; fval: 1400.000000; Exitflag: 1
```

The function value increases at each iteration because the problem has more constraints.

See Also

More About
- "Code Generation in fmincon Background" on page 6-124
- "Code Generation for Optimization Basics" on page 6-127
- "Optimization Code Generation for Real-Time Applications" on page 6-133

Optimization Code Generation for Real-Time Applications

Time Limits on Generated Code

Embedded applications might have requirements that limit how long code can run before returning an answer. Such requirements can be problematic, because solvers give no time guarantees for optimization. This topic outlines techniques for estimating how long your embedded code will run before returning a result, and describes changes you can make to your code to shorten the time requirement.

For general advice on writing efficient code for code generation, see "MATLAB Code Design Considerations for Code Generation" (MATLAB Coder).

Match the Target Environment

To estimate the execution time of generated code before code generation, set your MATLAB environment to match the target environment as closely as possible.

- Check the clock speeds of your target hardware and your computer. Scale your benchmarking results accordingly.
- Set `maxNumCompThreads` in MATLAB to 1, because the default LAPACK and BLAS libraries generated by MATLAB Coder are single-threaded.

  ```
  lastN = maxNumCompThreads(1);
  ```

  After you finish benchmarking, reset the `maxNumCompThreads` value:

  ```
  N = maxNumCompThreads(lastN);
  % Alternatively,
  % N = maxNumCompThreads('automatic');
  ```

  > **Note** If your target hardware has multiple cores and you use custom multithreaded LAPACK and BLAS libraries, then set `maxNumCompThreads` to the number of threads on the target hardware. See "Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls" (MATLAB Coder).

- If you have an Embedded Coder license, see these topics for details on reliable ways to evaluate the resulting performance of your embedded code: "Speed Up Linear Algebra in Code Generated from a MATLAB Function Block" (Embedded Coder), "Speed Up Matrix Operations in Code Generated from a MATLAB Function Block" (Embedded Coder), "Verification" (Embedded Coder), and "Performance" (Embedded Coder).

Set Coder Configuration

To set the configuration for code generation, call `coder.config`.

```
cfg = coder.config('mex');
```

To save time in the generated code, turn off integrity checks and checks for integer saturation. Solvers do not rely on these checks to function properly, assuming that the objective function and nonlinear constraint function do not require them. For details, see "Control Run-Time Checks" (MATLAB Coder).

```
cfg.IntegrityChecks = false;
cfg.SaturateOnIntegerOverflow = false;
```

Typically, generated code runs faster when using static memory allocation, although this allocation can increase the amount of generated code. Also, some hardware does not support dynamic memory allocation. To use static memory allocation, specify this setting.

```
cfg.DynamicMemoryAllocation = 'Off';
```

You can improve the performance of your code by selecting different types of BLAS, the underlying linear algebra subprograms. To learn how to set the BLAS for your generated code, see "Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls" (MATLAB Coder). If you want the embedded application to run in parallel, you must supply BLAS or LAPACK libraries that support parallel computation on your system. Similarly, when you have parallel hardware, you can improve the performance of your code by setting custom LAPACK calls. See "Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls" (MATLAB Coder).

Benchmark the Solver

Run your MEX generated code in a loop of 1000 evaluations using a set of input parameters that is typical of your application. Find both the total time and the maximum of the evaluation times. Try the parameters that you think might cause the solver to take too long, and test them and other parameters. If the MEX application returns satisfactory results in reasonable timeframes, then you can expect that the deployed application will do the same.

Set Initial Point

One of the most important factors affecting both runtime and solution quality is the initial point for the optimization x0. When parameters change slowly between solver calls, the solution from the previous call is typically a good starting point for the next call. See "Follow Equation Solution as a Parameter Changes" on page 13-25, which also shows how a jump in the solution time can occur because the solution switches "Basins of Attraction" on page 4-23.

If your optimization problem does not have parameters changing slowly, and includes only a few control variables, then trying to estimate a response from previous solutions can be worthwhile. Construct a model of the solution as a function of the parameters, either as a quadratic in the parameters or as a low-dimensional interpolation, and use the predicted solution point as a starting point for the solver.

Set Options Appropriately

You can sometimes speed a solution by adjusting parameters. If you set the MaxIterations option to allow only a few iterations, then the solver stops quickly. For example, if the solver is fmincon, enter this code.

```
opts = optimoptions('fmincon','Algorithm','sqp','MaxIterations',50);
[x,fval,exitflag] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

However, the result can be far from an optimum. Ensure that an inaccurate result does not overly affect your system. Set MaxIterations as large as possible while still meeting your time constraint. You can estimate this value by measuring how long an iteration takes, or by measuring how long a function evaluation takes, and then either setting the MaxFunctionEvaluations option or the

`MaxIterations` option. For an example, see "Code Generation for Optimization Basics" on page 6-127.

For further suggestions on settings that can speed the solver, see "Solver Takes Too Long" on page 4-9. Note that some suggestions in this topic do not apply because of limitations in code generation. See "Code Generation in fmincon Background" on page 6-124 or "Code Generation for quadprog Background" on page 11-58.

Global Minimum

You might want a global minimum, not just a local minimum, as a solution. Searching for a global minimum can take a great deal of time, and is not guaranteed to work. For suggestions, see "Searching for a Smaller Minimum" on page 4-22.

See Also
codegen | fmincon | optimoptions | quadprog

More About
- "Code Generation in fmincon Background" on page 6-124
- "Code Generation for quadprog Background" on page 11-58
- "Code Generation for Optimization Basics" on page 6-127
- "Generate Code for quadprog" on page 11-60
- "Static Memory Allocation for fmincon Code Generation" on page 6-131

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

$$K_1(x, w_1) = \sin(w_1 x_1)\cos(w_1 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 \leq 1,$$

$$K_2(x, w_2) = \sin(w_2 x_2)\cos(w_2 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 \leq 1,$$

for all values of w_1 and w_2 over the ranges

$1 \leq w_1 \leq 100,$
$1 \leq w_2 \leq 100.$

Note that the semi-infinite constraints are one-dimensional, that is, vectors. Because the constraints must be in the form $K_i(x, w_i) \leq 0$, you need to compute the constraints as

$$K_1(x, w_1) = \sin(w_1 x_1)\cos(w_1 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 - 1 \leq 0,$$

$$K_2(x, w_2) = \sin(w_2 x_2)\cos(w_2 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 - 1 \leq 0.$$

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.
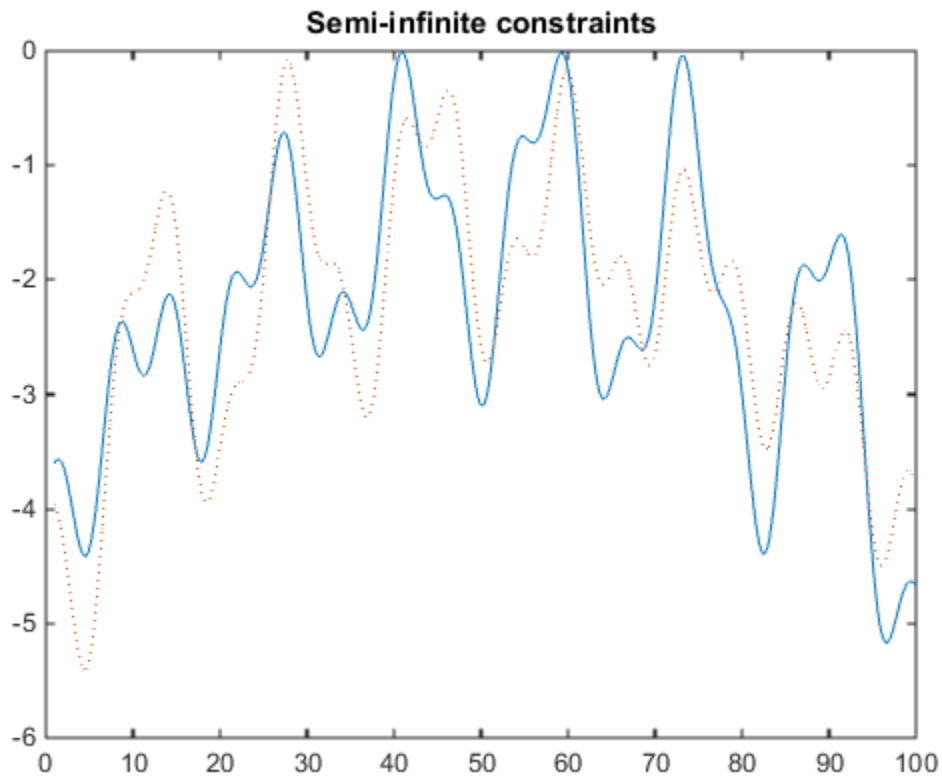
This plot shows how peaks in both constraints are on the constraint boundary.

The plot command inside `mycon.m` slows down the computation. Remove this line to improve the speed.

See Also

`fseminf`

Related Examples

- "Two-Dimensional Semi-Infinite Constraint" on page 6-139
- "Analyzing the Effect of Uncertainty Using Semi-Infinite Programming" on page 6-142

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

$$K_1(x, w) = \sin(w_1 x_1)\cos(w_2 x_2) - \frac{1}{1000}(w_1 - 50)^2 - \sin(w_1 x_3) - x_3 + \dots$$

$$\sin(w_2 x_2)\cos(w_1 x_1) - \frac{1}{1000}(w_2 - 50)^2 - \sin(w_2 x_3) - x_3 \le 1.5,$$

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 =
    0.2523    0.1715    0.1938
```

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 =
    -0.0370
```

This call to `mycon` produces the following surf plot, which shows the semi-infinite constraint at x.

Semi-infinite constraint

See Also

`fseminf`
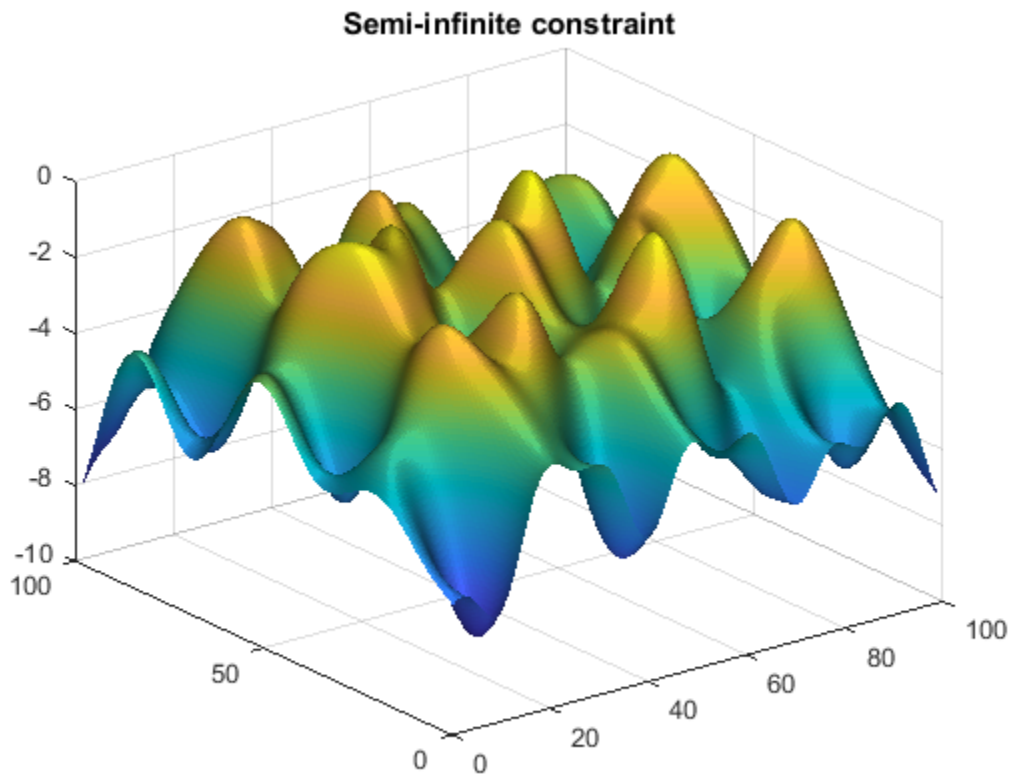
Related Examples

- "One-Dimensional Semi-Infinite Constraints" on page 6-136
- "Analyzing the Effect of Uncertainty Using Semi-Infinite Programming" on page 6-142

Analyzing the Effect of Uncertainty Using Semi-Infinite Programming

This example shows how to use semi-infinite programming to investigate the effect of uncertainty in the model parameters of an optimization problem. We will formulate and solve an optimization problem using the function `fseminf`, a semi-infinite programming solver in Optimization Toolbox™.

The problem illustrated in this example involves the control of air pollution. Specifically, a set of chimney stacks are to be built in a given geographic area. As the height of each chimney stack increases, the ground level concentration of pollutants from the stack decreases. However, the construction cost of each chimney stack increases with height. We will solve a problem to minimize the cumulative height of the chimney stacks, hence construction cost, subject to ground level pollution concentration not exceeding a legislated limit. This problem is outlined in the following reference:

Air pollution control with semi-infinite programming, A.I.F. Vaz and E.C. Ferreira, XXVIII Congreso Nacional de Estadistica e Investigacion Operativa, October 2004

In this example we will first solve the problem published in the above article as the *Minimal Stack Height* problem. The models in this problem are dependent on several parameters, two of which are wind speed and direction. All model parameters are assumed to be known exactly in the first solution of the problem.

We then extend the original problem by allowing the wind speed and direction parameters to vary within given ranges. This will allow us to analyze the effects of uncertainty in these parameters on the optimal solution to this problem.

Minimal Stack Height Problem

Consider a 20km-by-20km region, R, in which ten chimney stacks are to be placed. These chimney stacks release several pollutants into the atmosphere, one of which is sulfur dioxide. The x, y locations of the stacks are fixed, but the height of the stacks can vary.

Constructors of the stacks would like to minimize the total height of the stacks, thus minimizing construction costs. However, this is balanced by the conflicting requirement that the concentration of sulfur dioxide at any point on the ground in the region R must not exceed the legislated maximum.

First, let's plot the chimney stacks at their initial height. Note that we have zoomed in on a 4km-by-4km subregion of R which contains the chimney stacks.

```
h0 = [210;210;180;180;150;150;120;120;90;90];
plotChimneyStacks(h0, 'Chimney Stack Initial Height');
```
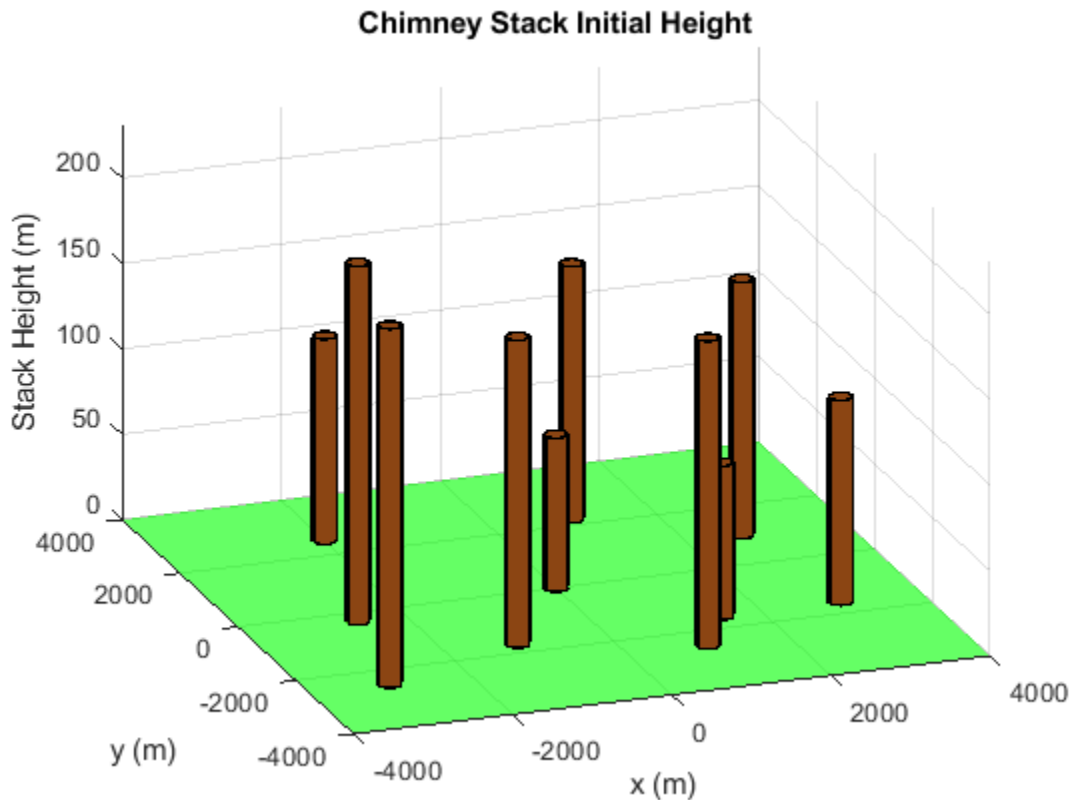
There are two environment related parameters in this problem, the wind speed and direction. Later in this example we will allow these parameters to vary, but for the first problem we will set these parameters to typical values.

```
% Wind direction in radians
theta0 = 3.996;
% Wind speed in m/s
U0 = 5.64;
```

Now let's plot the ground level concentration of sulfur dioxide (SO2) over the entire region R (remember that the plot of chimney stacks was over a smaller region). The SO2 concentration has been calculated with the chimney stacks set to their initial heights.

We can see that the concentration of SO2 varies over the region of interest. There are two features of the Sulfur Dioxide graph of note:

- SO2 concentration rises in the top left hand corner of the (x,y) plane
- SO2 concentration is approximately zero throughout most of the region

In very simple terms, the first feature is due to the prevailing wind, which is blowing SO2 toward the top left hand corner of the (x,y) plane in this example. The second factor is due to SO2 being transported to the ground via diffusion. This is a slower process compared to the prevailing wind and thus SO2 only reaches ground level in the top left hand corner of the region of interest.

For a more detailed discussion of atmospheric dispersion from chimney stacks, consult the reference cited in the introduction.

The pink plane indicates a SO2 concentration of $0.000125 gm^{-3}$. This is the legislated maximum for which the Sulfur Dioxide concentration must not exceed in the region R. It can be clearly seen from the graph that the SO2 concentration exceeds the maximum for the initial chimney stack height.

Examine the MATLAB file `concSulfurDioxide` to see how the sulfur dioxide concentration is calculated.

```
plotSulfurDioxide(h0, theta0, U0, ...
    'Sulfur Dioxide Concentration at Initial Stack Height');
```


Sulfur Dioxide Concentration at Initial Stack Height
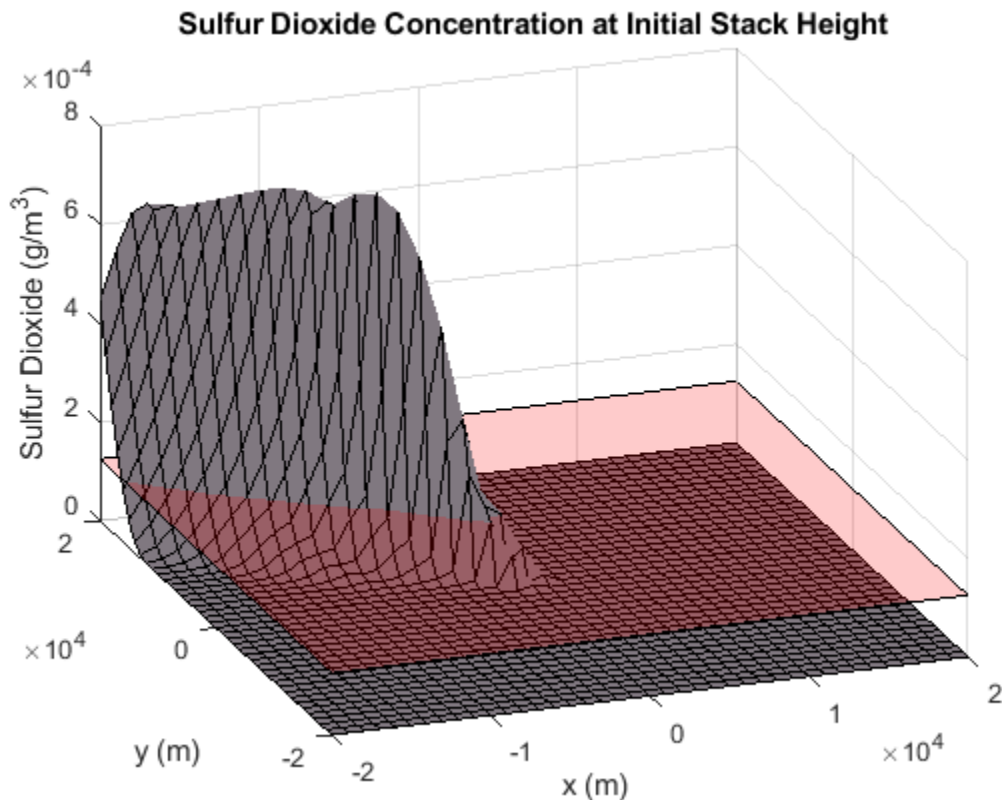
How `fseminf` Works

Before we solve the minimal stack height problem, we will outline how `fseminf` solves a semi-infinite problem. A general semi-infinite programming problem can be stated as:

$$\min f(x)$$

such that

$Ax < = b$ (Linear inequality constraints)

$Aeq * x = beq$ (Linear equality constraints)

$c(x) < = 0$ (Nonlinear Inequality Constraints)

$ceq(x) = 0$ (Nonlinear Equality Constraints)

$l <\ = x <\ = u$ (Bounds)

and

$K_j(x, w) <\ = 0$, where $w \in I_j$ for $j = 1, \ldots, n_{inf}$ (Nonlinear semi-infinite constraints)

This algorithm allows you to specify constraints for a nonlinear optimization problem that must be satisfied over intervals of an auxiliary variable, w. Note that for `fseminf`, this variable is restricted to be either 1 or 2 dimensional for each semi-infinite constraint.

The function `fseminf` solves the general semi-infinite problem by starting from an initial value, x_0, and using an iterative procedure to obtain an optimum solution, x_{opt}.

The key component of the algorithm is the handling of the "semi-infinite" constraints, K_j. At x_{opt} it is required that the K_j must be feasible at every value of w in the interval I_j. This constraint can be simplified by considering all the local maxima of K_j with respect to w in the interval I_j. The original constraint is equivalent to requiring that the value of K_j at each of the above local maxima is feasible.

`fseminf` calculates an approximation to all the local maximum values of each semi-infinite constraint, K_j. To do this, `fseminf` first calculates each semi-infinite constraint over a mesh of w values. A simple differencing scheme is then used to calculate all the local maximum values of K_j from the evaluated semi-infinite constraint.

As we will see later, you create this mesh in your constraint function. The spacing you should use for each w coordinate of the mesh is supplied to your constraint function by `fseminf`.

At each iteration of the algorithm, the following steps are performed:

1 Evaluate K_j over a mesh of w-values using the current mesh spacing for each w-coordinate.

2 Calculate an approximation to all the local maximum values of K_j using the evaluation of K_j from step 1.

3 Replace each K_j in the general semi-infinite problem with the set of local maximum values found in steps 1-2. The problem now has a finite number of nonlinear constraints. `fseminf` uses the SQP algorithm used by `fmincon` to take one iteration step of the modified problem.

4 Check if any of the SQP algorithm's stopping criteria are met at the new point x. If any criteria are met the algorithm terminates; if not, `fseminf` continues to step 5. For example, if the first order optimality value for the problem defined in step 3 is less than the specified tolerance then `fseminf` will terminate.

5 Update the mesh spacing used in the evaluation of the semi-infinite constraints in step 1.

Writing the Nonlinear Constraint Function

Before we can call `fseminf` to solve the problem, we need to write a function to evaluate the nonlinear constraints in this problem. The constraint to be implemented is that the ground level Sulfur Dioxide concentration must not exceed $0.000125 gm^{-3}$ at every point in region R.

This is a semi-infinite constraint, and the implementation of the constraint function is explained in this section. For the minimal stack height problem we have implemented the constraint in the MATLAB file `airPollutionCon`.

type `airPollutionCon.m`

```
function [c, ceq, K, s] = airPollutionCon(h, s, theta, U)
%AIRPOLLUTIONCON Constraint function for air pollution demo
%
%   [C, CEQ, K, S] = AIRPOLLUTIONCON(H, S, THETA, U) calculates the
%   constraints for the air pollution Optimization Toolbox (TM) demo. This
%   function first creates a grid of (X, Y) points using the supplied grid
%   spacing, S. The following constraint is then calculated over each point
%   of the grid:
%
%   Sulfur Dioxide concentration at the specified wind direction, THETA and
%   wind speed U <= 1.25e-4 g/m^3
%
%   See also AIRPOLLUTION

%   Copyright 2008 The MathWorks, Inc.

% Initial sampling interval
if nargin < 2 || isnan(s(1,1))
    s = [1000 4000];
end

% Define the grid that the "infinite" constraints will be evaluated over
w1x = -20000:s(1,1):20000;
w1y = -20000:s(1,2):20000;
[t1,t2] = meshgrid(w1x,w1y);

% Maximum allowed sulphur dioxide
maxsul = 1.25e-4;

% Calculate the constraint over the grid
K = concSulfurDioxide(t1, t2, h, theta, U) - maxsul;

% Rescale constraint to make it 0(1)
K = 1e4*K;

% No finite constraints
c = [];
ceq = [];
```

This function illustrates the general structure of a constraint function for a semi-infinite programming problem. In particular, a constraint function for `fseminf` can be broken up into three parts:

1. Define the initial mesh size for the constraint evaluation

Recall that `fseminf` evaluates the "semi-infinite" constraints over a mesh as part of the overall calculation of these constraints. When your constraint function is called by `fseminf`, the mesh spacing you should use is supplied to your function. `Fseminf` will initially call your constraint function with the mesh spacing, `s`, set to NaN. This allows you to initialize the mesh size for the constraint evaluation. Here, we have one "infinite" constraint in two "infinite" variables. This means we need to initialize the mesh size to a 1-by-2 matrix, in this case, `s = [1000 4000]`.

2. Define the mesh that will be used for the constraint evaluation

A mesh that will be used for the constraint evaluation needs to be created. The three lines of code following the comment "Define the grid that the "infinite" constraints will be evaluated over" in `airPollutionCon` can be modified for most 2-d semi-infinite programming problems.

3. Calculate the constraints over the mesh

Once the mesh has been defined, the constraints can be calculated over it. These constraints are then returned to `fseminf` from the above constraint function.

Note that in this problem, we have also rescaled the constraints so that they vary on a scale which is closer to that of the objective function. This helps `fseminf` to avoid scaling issues associated with objectives and constraints which vary on disparate scales.

Solve the Optimization Problem

We can now call `fseminf` to solve the problem. The chimney stacks must all be at least 10m tall and we use the initial stack height specified earlier. Note that the third input argument to `fseminf` below (1) indicates that there is only one semi-infinite constraint.

```
lb = 10*ones(size(h0));
[hsopt, sumh, exitflag] = fseminf(@(h)sum(h), h0, 1, ...
    @(h,s) airPollutionCon(h,s,theta0,U0), [], [], [], [], lb);
```

```
Local minimum possible. Constraints satisfied.

fseminf stopped because the predicted change in the objective function
is less than the value of the function tolerance and constraints
are satisfied to within the value of the constraint tolerance.
```

```
fprintf('\nMinimum computed cumulative height of chimney stacks : %7.2f m\n', sumh);
```

```
Minimum computed cumulative height of chimney stacks : 3667.19 m
```

The minimum cumulative height computed by `fseminf` is considerably higher than the initial total height of the chimney stacks. We will see how the minimum cumulative height changes when parameter uncertainty is added to the problem later in the example. For now, let's plot the chimney stacks at their optimal height.

Examine the MATLAB file `plotChimneyStacks` to see how the plot was generated.

```
plotChimneyStacks(hsopt, 'Chimney Stack Optimal Height');
```

Chimney Stack Optimal Height
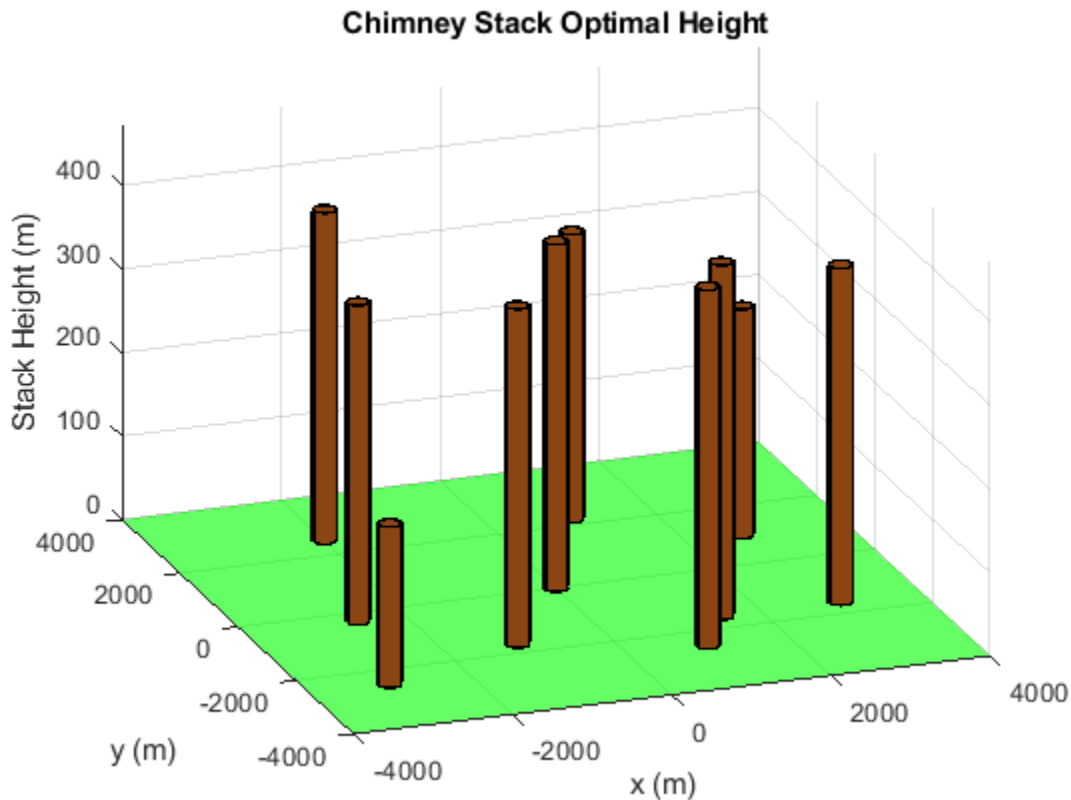
Check the Optimization Results

Recall that `fseminf` determines that the semi-infinite constraint is satisfied everywhere by ensuring that discretized maxima of the constraint are below the specified bound. We can verify that the semi-infinite constraint is satisfied everywhere by plotting the ground level sulfur dioxide concentration for the optimal stack height.

Note that the sulfur dioxide concentration takes its maximum possible value in the upper left corner of the (x, y) plane, i.e. at x = -20000m, y = 20000m. This point is marked by the blue dot in the figure below and verified by calculating the sulfur dioxide concentration at this point.

Examine the MATLAB file `plotSulfurDioxide` to see how the plots was generated.

```
titleStr = 'Optimal Sulfur Dioxide Concentration and its maximum (blue)';
xMaxSD = [-20000 20000];
plotSulfurDioxide(hsopt, theta0, U0, titleStr, xMaxSD);
```

Optimal Sulfur Dioxide Concentration and its maximum (blue)
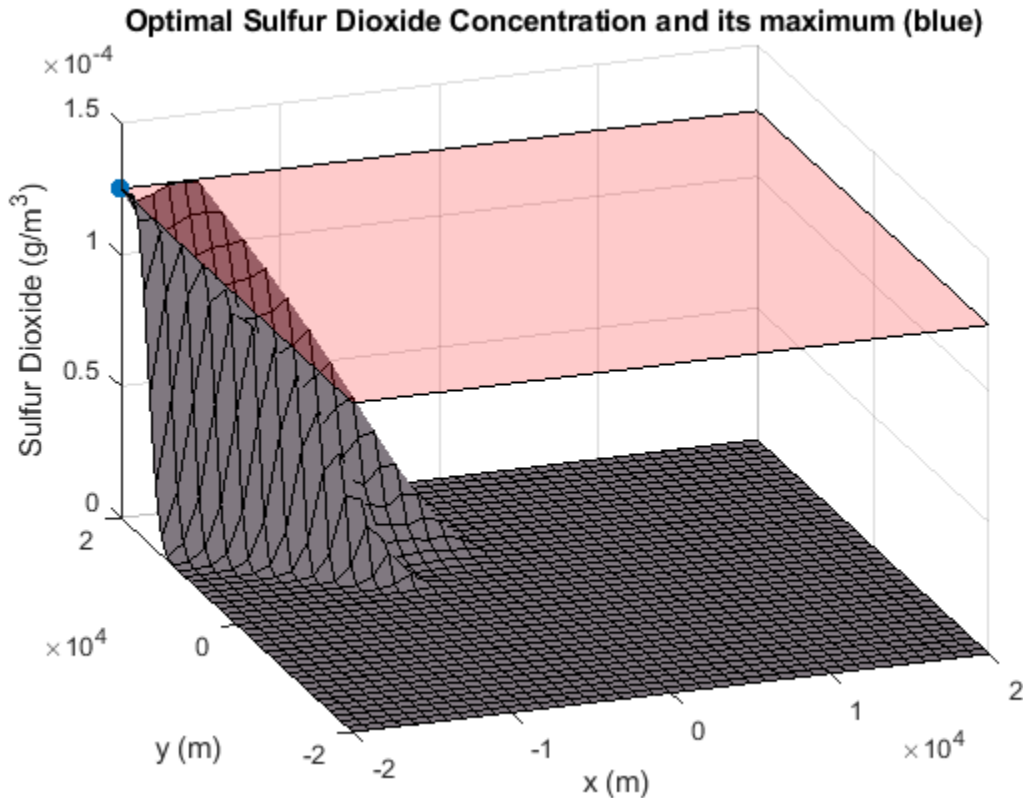
```
SO2Max = concSulfurDioxide(-20000, 20000, hsopt, theta0, U0);
fprintf('Sulfur Dioxide Concentration at x = -20000m, y = 20000m : %e g/m^3\n', SO2Max);
```

```
Sulfur Dioxide Concentration at x = -20000m, y = 20000m : 1.250000e-04 g/m^3
```

Considering Uncertainty in the Environmental Factors

The sulfur dioxide concentration depends on several environmental factors which were held at fixed values in the above problem. Two of the environmental factors are wind speed and wind direction. See the reference cited in the introduction for a more detailed discussion of all the problem parameters.

We can investigate the change in behavior for the system with respect to the wind speed and direction. In this section of the example, we want to make sure that the sulfur dioxide limits are satisfied even if the wind direction changes from 3.82 rad to 4.18 rad and mean wind speed varies between 5 and 6.2 m/s.

We need to implement a semi-infinite constraint to ensure that the sulfur dioxide concentration does not exceed the limit in region R. This constraint is required to be feasible for all pairs of wind speed and direction.

Such a constraint will have four "infinite" variables (wind speed and direction and the x-y coordinates of the ground). However, any semi-infinite constraint supplied to `fseminf` can have no more than two "infinite" variables.

To implement this constraint in a suitable form for `fseminf`, we recall the SO2 concentration at the optimum stack height in the previous problem. In particular, the SO2 concentration takes its

maximum possible value at x = -20000m, y = 20000m. To reduce the number of "infinite" variables, we will assume that the SO2 concentration will also take its maximum value at this point when uncertainty is present. We then require that SO2 concentration at this point is below $0.000125 gm^{-3}$ for all pairs of wind speed and direction.

This means that the "infinite" variables for this problem are wind speed and direction. To see how this constraint has been implemented, inspect the MATLAB file `uncertainAirPollutionCon`.

```
type uncertainAirPollutionCon.m
```

```
function [c, ceq, K, s] = uncertainAirPollutionCon(h, s)
%UNCERTAINAIRPOLLUTIONCON Constraint function for air pollution demo
%
%   [C, CEQ, K, S] = UNCERTAINAIRPOLLUTIONCON(H, S) calculates the
%   constraints for the fseminf Optimization Toolbox (TM) demo. This
%   function first creates a grid of wind speed/direction points using the
%   supplied grid spacing, S. The following constraint is then calculated
%   over each point of the grid:
%
%   Sulfur Dioxide concentration at x = -20000m, y = 20000m <= 1.25e-4
%   g/m^3
%
%   See also AIRPOLLUTIONCON, AIRPOLLUTION

%   Copyright 2008 The MathWorks, Inc.

% Maximum allowed sulphur dioxide
maxsul = 1.25e-4;

% Initial sampling interval
if nargin < 2 || isnan(s(1,1))
    s = [0.02 0.04];
end

% Define the grid that the "infinite" constraints will be evaluated over
w1x = 3.82:s(1,1):4.18; % Wind direction
w1y = 5.0:s(1,2):6.2;    % Wind speed
[t1,t2] = meshgrid(w1x,w1y);

% We assume the maximum SO2 concentration is at [x, y] = [-20000, 20000]
% for all wind speed/direction pairs. We evaluate the SO2 constraint over
% the [theta, U] grid at this point.
K = concSulfurDioxide(-20000, 20000, h, t1, t2) - maxsul;

% Rescale constraint to make it O(1)
K = 1e4*K;

% No finite constraints
c = [];
ceq = [];
```

This constraint function can be divided into same three sections as before:

1. Define the initial mesh size for the constraint evaluation

The code following the comment "Initial sampling interval" initializes the mesh size.

2. Define the mesh that will be used for the constraint evaluation

The next section of code creates a mesh (now in wind speed and direction) using a similar construction to that used in the initial problem.

3. Calculate the constraints over the mesh

The remainder of the code calculates the SO2 concentration at each point of the wind speed/direction mesh. These constraints are then returned to `fseminf` from the above constraint function.

We can now call `fseminf` to solve the stack height problem considering uncertainty in the environmental factors.

```
[hsopt2, sumh2, exitflag2] = fseminf(@(h)sum(h), h0, 1, ...
    @uncertainAirPollutionCon, [], [], [], [], lb);
```

Local minimum found that satisfies the constraints.

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

```
fprintf('\nMinimal computed cumulative height of chimney stacks with uncertainty: %7.2f m\n', sum
```

Minimal computed cumulative height of chimney stacks with uncertainty: 3812.14 m

We can now look at the difference between the minimum computed cumulative stack height for the problem with and without parameter uncertainty. You should be able to see that the minimum cumulative height increases when uncertainty is added to the problem. This expected increase in height allows the SO2 concentration to remain below the legislated maximum for all wind speed/direction pairs in the specified range.

We can check that the sulfur dioxide concentration does not exceed the limit over the region of interest via inspection of a sulfur dioxide plot. For a given (x, y) point, we plot the maximum SO2 concentration for the wind speed and direction in the stated ranges. Note that we have zoomed in on the upper left corner of the X-Y plane.

```
titleStr = 'Optimal Sulfur Dioxide Concentration under Uncertainty';
thetaRange = 3.82:0.02:4.18;
URange = 5:0.2:6.2;
XRange = [-20000,-15000];
YRange = [15000,20000];
plotSulfurDioxideUncertain(hsopt2, thetaRange, URange, XRange, YRange, titleStr);
```
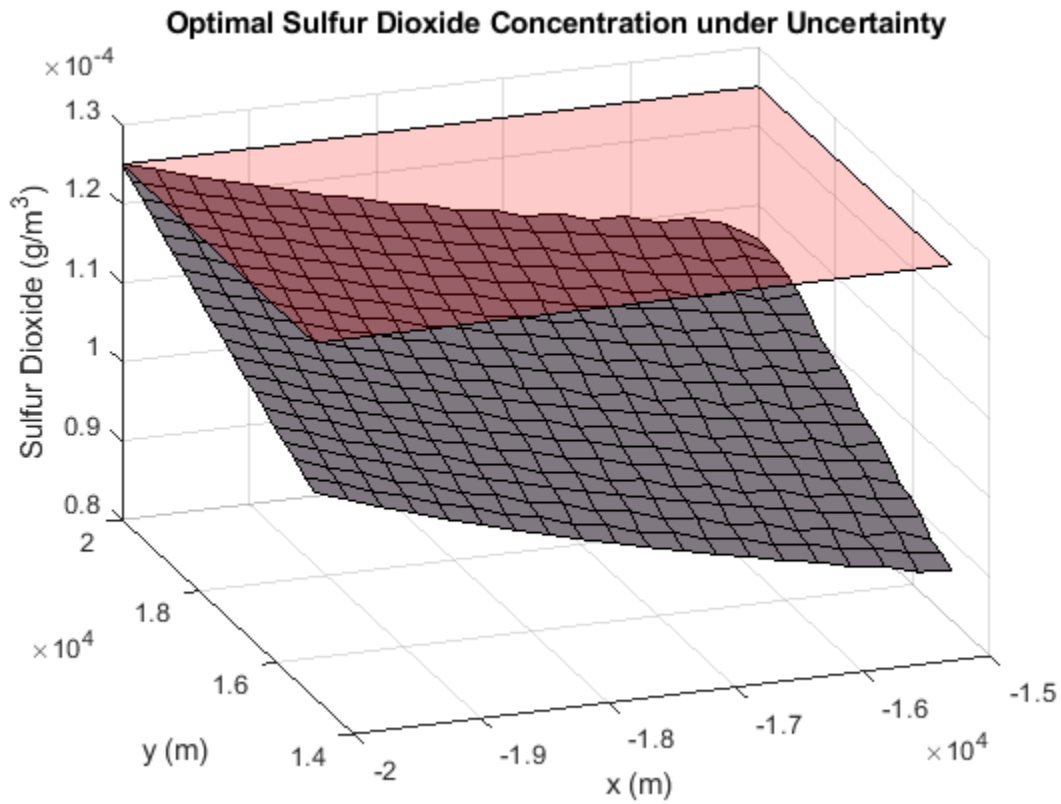
We finally plot the chimney stacks at their optimal height when there is uncertainty in the problem definition.

```
plotChimneyStacks(hsopt2, 'Chimney Stack Optimal Height under Uncertainty');
```

Chimney Stack Optimal Height under Uncertainty
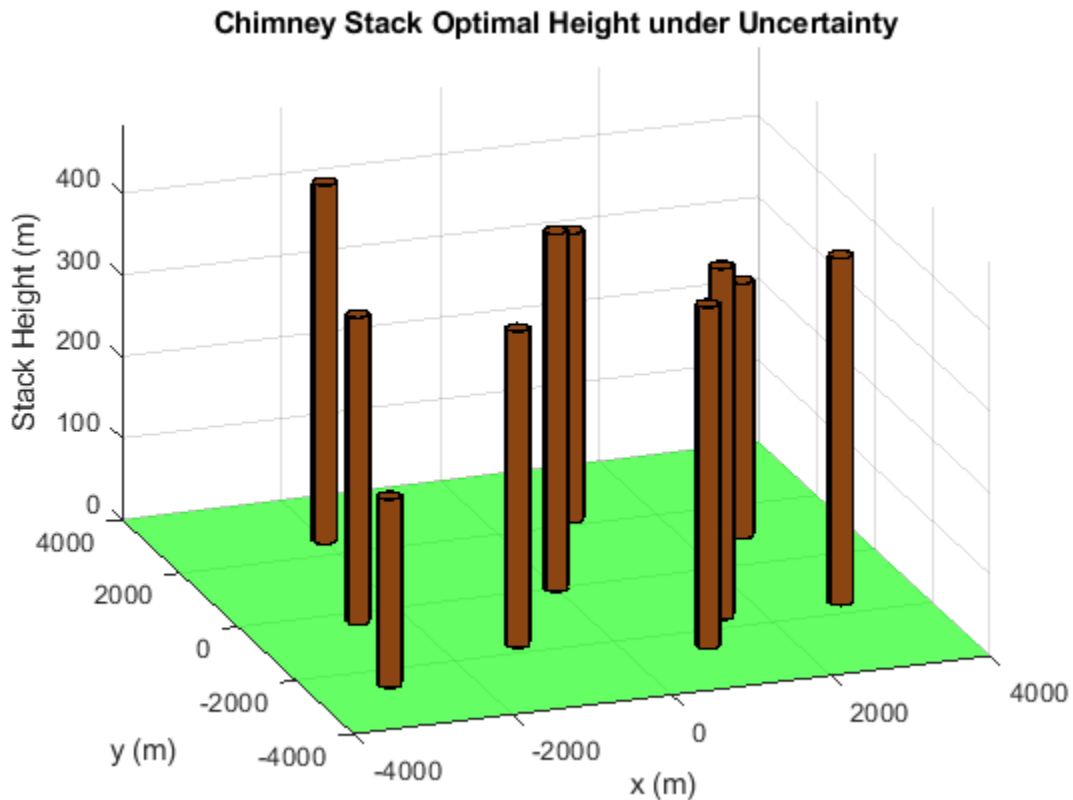
There are many options available for the semi-infinite programming algorithm, `fseminf`. Consult the Optimization Toolbox™ User's Guide for details, in the Using Optimization Toolbox Solvers chapter, under Constrained Nonlinear Optimization: fseminf Problem Formulation and Algorithm.

See Also

More About

- "One-Dimensional Semi-Infinite Constraints" on page 6-136
- "Two-Dimensional Semi-Infinite Constraint" on page 6-139

Nonlinear Problem-Based

Rational Objective Function, Problem-Based

The problem-based approach to optimization involves creating optimization variables and expressing the objective and constraints in terms of those variables.

A rational function is a quotient of polynomials. When the objective function is a rational function of optimization variables or other supported function, you can create the objective function expression directly from the variables. In contrast, when your objective function is not a supported function, you must create a MATLAB® function that represents the objective and then convert the function to an expression by using `fcn2optimexpr`. See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8.

For example, write the objective function

$$f = \frac{(x-y)^2}{4+(x+y)^4}\frac{x+y^2}{1+y^2}$$

in terms of two optimization variables `x` and `y`.

```
x = optimvar('x');
y = optimvar('y');
f = (x-y)^2/(4+(x+y)^4)*(x+y^2)/(1+y^2);
```

To find the minimum of this objective function, create an optimization problem with `f` as the objective, set an initial point, and call `solve`.

```
prob = optimproblem('Objective',f);
x0.x = -1;
x0.y = 1;
[sol,fval,exitflag,output] = solve(prob,x0)
```

```
Solving problem using fminunc.

Local minimum found.

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

sol = struct with fields:
    x: -2.1423
    y: 0.7937


fval = -1.0945

exitflag =
    OptimalSolution


output = struct with fields:
       iterations: 9
        funcCount: 10
         stepsize: 1.7073e-06
      lssteplength: 1
     firstorderopt: 1.4999e-07
         algorithm: 'quasi-newton'
```

```
       message: '...'
        solver: 'fminunc'
```

The exit flag shows that the reported solution is a local minimum. The output structure shows that the solver took just 30 function evaluations to reach the minimum.

See Also

`fcn2optimexpr`

More About

*   "Problem-Based Optimization Workflow" on page 10-2
*   "Problem-Based Optimization Setup"

Solve Constrained Nonlinear Optimization, Problem-Based

This example shows how to find the minimum of a nonlinear objective function with a nonlinear constraint by using the problem-based approach. For a video showing the solution to a similar problem, see Problem-Based Nonlinear Programming.

To find the minimum value of a nonlinear objective function using the problem-based approach, first write the objective function as a file or anonymous function. The objective function for this example is

$$f(x, y) = e^x\left(4x^2 + 2y^2 + 4xy + 2y - 1\right).$$

type objfunx

```
function f = objfunx(x,y)
f = exp(x).*(4*x.^2 + 2*y.^2 + 4*x.*y + 2*y - 1);
end
```

Create the optimization problem variables x and y.

```
x = optimvar('x');
y = optimvar('y');
```

Create the objective function as an expression in the optimization variables.

```
obj = objfunx(x,y);
```

Create an optimization problem with obj as the objective function.

```
prob = optimproblem('Objective',obj);
```

Create a nonlinear constraint that the solution lies in a tilted ellipse, specified as

$$\frac{xy}{2} + (x + 2)^2 + \frac{(y - 2)^2}{2} \le 2.$$

Create the constraint as an inequality expression in the optimization variables.

```
TiltEllipse = x.*y/2 + (x+2).^2 + (y-2).^2/2 <= 2;
```

Include the constraint in the problem.

```
prob.Constraints.constr = TiltEllipse;
```

Create a structure representing the initial point as x = −3, y = 3.

```
x0.x = -3;
x0.y = 3;
```

Review the problem.

```
show(prob)

  OptimizationProblem :

    Solve for:
       x, y
```

```
minimize :
    (exp(x) .* (((((4 .* x.^2) + (2 .* y.^2)) + ((4 .* x) .* y))
 + (2 .* y)) - 1))


subject to constr:
    ((((x .* y) ./ 2) + (x + 2).^2) + ((y - 2).^2 ./ 2)) <= 2
```

Solve the problem.

```
[sol,fval] = solve(prob,x0)
```

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol = struct with fields:
    x: -5.2813
    y: 4.6815

fval = 0.3299
```

Try a different start point.

```
x0.x = -1;
x0.y = 1;
[sol2,fval2] = solve(prob,x0)
```

Solving problem using fmincon.

Feasible point with lower objective function value found.

Local minimum found that satisfies the constraints.

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

sol2 = struct with fields:
    x: -0.8210
    y: 0.6696

fval2 = 0.7626
```

Plot the ellipse, the objective function contours, and the two solutions.

```
f = @objfunx;
g = @(x,y) x.*y/2+(x+2).^2+(y-2).^2/2-2;
rnge = [-5.5 -0.25 -0.25 7];
fimplicit(g,'k-')
axis(rnge);
```

```
hold on
fcontour(f,rnge,'LevelList',logspace(-1,1))
plot(sol.x,sol.y,'ro','LineWidth',2)
plot(sol2.x,sol2.y,'ko','LineWidth',2)
legend('Constraint','f Contours','Global Solution','Local Solution','Location','northeast');
hold off
```
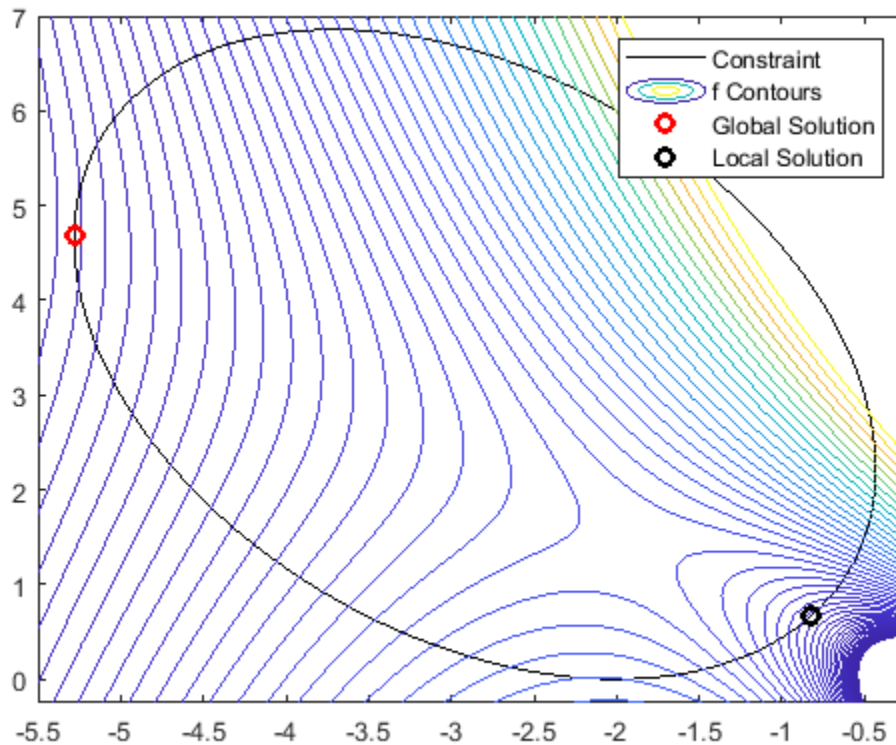

The solutions are on the nonlinear constraint boundary. The contour plot shows that these are the only local minima. The plot also shows that there is a stationary point near [–2,3/2], and local maxima near [–2,0] and [–1,4].

Convert Objective Function Using `fcn2optimexpr`

For some objective functions or software versions, you must convert nonlinear functions to optimization expressions by using `fcn2optimexpr`. See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8. Pass the `x` and `y` variables in the `fcn2optimexpr` call to indicate which optimization variable corresponds to each `objfunx` input.

```
obj = fcn2optimexpr(@objfunx,x,y);
```

Create an optimization problem with `obj` as the objective function just as before.

```
prob = optimproblem('Objective',obj);
```

The remainder of the solution process is identical.

See Also
`fcn2optimexpr`

More About
- "Problem-Based Optimization Workflow" on page 10-2
- "Problem-Based Nonlinear Optimization"

Convert Nonlinear Function to Optimization Expression

This section shows how to choose whether to convert a nonlinear function to an optimization expression or to create the expression out of supported operations on optimization variables. The section also shows how to convert a function, if necessary, by using `fcn2optimexpr`.

Use Supported Operations When Possible

Generally, create your objective or nonlinear constraint functions by using supported operations on optimization variables and expressions. Doing so has these advantages:

*   `solve` includes gradients calculated by automatic differentiation. See "Effect of Automatic Differentiation in Problem-Based Optimization" on page 7-23.

*   `solve` has a wider choice of available solvers. When using `fcn2optimexpr`, `solve` uses only `fmincon` or `fminunc`.

In general, supported operations include all elementary mathematical operations: addition, subtraction, multiplication, division, powers, and elementary functions such as exponential and trigonometric functions and their inverses. Nonsmooth operations such as `max`, `abs`, `if`, and `case` are not supported. For the complete description, see "Supported Operations on Optimization Variables and Expressions" on page 10-43.

For example, suppose that your objective function is

$$f(x, y, r) = 100(y - x^2)^2 + (r - x)^2$$

where r is a parameter that you supply, and the problem is to minimize f over x and y. This objective function is a sum of squares, and takes the minimal value of 0 at the point $x = r$, $y = r^2$.

The objective function is a polynomial, so you can write it in terms of elementary operations on optimization variables.

```
r = 2;
x = optimvar('x');
y = optimvar('y');
f = 100*(y - x^2)^2 + (r - x)^2;
prob = optimproblem("Objective",f);
x0.x = -1;
x0.y = 2;
[sol,fval] = solve(prob,x0)

Solving problem using lsqnonlin.

Local minimum found.

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

sol = struct with fields:
    x: 2.0000
    y: 4.0000


fval = 1.4644e-20
```

To solve the same problem by converting the objective function using `fcn2optimexpr` (not recommended), first write the objective as an anonymous function.

```
fun = @(x,y)100*(y - x^2)^2 + (r - x)^2;
prob.Objective = fcn2optimexpr(fun,x,y);
[sol2,fval2] = solve(prob,x0)
```

```
Solving problem using fminunc.

Local minimum found.

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

sol2 = struct with fields:
    x: 2.0000
    y: 3.9998


fval2 = 1.7143e-09
```

Notice that `solve` uses `fminunc` this time instead of the more efficient `lsqnonlin`, and the reported solution for y is slightly different than the correct solution 4. Furthermore, the reported `fval` is about 1e-9 instead of 1e-20 (the actual solution value is exactly 0). These slight inaccuracies are due to `solve` not using the more efficient solver.

The remainder of this example shows how to convert a function to an optimization expression using `fcn2optimexpr`.

Function File

To use a function file in the problem-based approach, you must convert the file to an expression using `fcn2optimexpr`.

For example, the `expfn3.m` file contains the following code:

```
type expfn3.m

function [f,g,mineval] = expfn3(u,v)
mineval = min(eig(u));
f = v'*u*v;
f = -exp(-f);
t = u*v;
g = t'*t + sum(t) - 3;
```

This function is not entirely composed of supported operations because of `min(eig(u))`. Therefore, to use `expfn3(u,v)` as an optimization expression, you must first convert it using `fcn2optimexpr`.

To use `expfn3` as an optimization expression, first create optimization variables of the appropriate sizes.

```
u = optimvar('u',3,3,'LowerBound',-1,'UpperBound',1); % 3-by-3 variable
v = optimvar('v',3,'LowerBound',-2,'UpperBound',2); % 3-by-1 variable
```

Convert the function file to an optimization expressions using `fcn2optimexpr`.

```
[f,g,mineval] = fcn2optimexpr(@expfn3,u,v);
```

Because all returned expressions are scalar, you can save computing time by specifying the expression sizes using the `'OutputSize'` name-value pair argument. Also, because `expfn3` computes all of the outputs, you can save more computing time by using the `ReuseEvaluation` name-value pair.

```
[f,g,mineval] = fcn2optimexpr(@expfn3,u,v,'OutputSize',[1,1],'ReuseEvaluation',true)

f =
  Nonlinear OptimizationExpression

    [argout,~,~] = expfn3(u, v)


g =
  Nonlinear OptimizationExpression

    [~,argout,~] = expfn3(u, v)


mineval =
  Nonlinear OptimizationExpression

    [~,~,argout] = expfn3(u, v)
```

Anonymous Function

To use a general nonlinear function handle in the problem-based approach, convert the handle to an optimization expression using `fcn2optimexpr`. For example, write a function handle equivalent to `mineval` and convert it.

```
fun = @(u)min(eig(u));
funexpr = fcn2optimexpr(fun,u,'OutputSize',[1,1])

funexpr =
  Nonlinear OptimizationExpression

    anonymousFunction2(u)

  where:

    anonymousFunction2 = @(u)min(eig(u));
```

Create Objective

To use the objective expression as an objective function, create an optimization problem.

```
prob = optimproblem;
prob.Objective = f;
```

Define Constraints

Define the constraint `g <= 0` in the optimization problem.

```
prob.Constraints.nlcons1 = g <= 0;
```

Also define the constraints that `u` is symmetric and that mineval $\geq -1/2$.

```
prob.Constraints.sym = u == u.';
prob.Constraints.mineval = mineval >= -1/2;
```

View the problem.

```
show(prob)
```

```
  OptimizationProblem :

    Solve for:
      u, v

    minimize :
      [argout,~,~] = expfn3(u, v)


    subject to nlcons1:
      arg_LHS <= 0

      where:

        [~,arg_LHS,~] = expfn3(u, v);

    subject to sym:
      u(2, 1) - u(1, 2) == 0
      u(3, 1) - u(1, 3) == 0
      -u(2, 1) + u(1, 2) == 0
      u(3, 2) - u(2, 3) == 0
      -u(3, 1) + u(1, 3) == 0
      -u(3, 2) + u(2, 3) == 0

    subject to mineval:
      arg_LHS >= (-0.5)

      where:

        [~,~,arg_LHS] = expfn3(u, v);

    variable bounds:
      -1 <= u(1, 1) <= 1
      -1 <= u(2, 1) <= 1
      -1 <= u(3, 1) <= 1
      -1 <= u(1, 2) <= 1
      -1 <= u(2, 2) <= 1
      -1 <= u(3, 2) <= 1
      -1 <= u(1, 3) <= 1
      -1 <= u(2, 3) <= 1
      -1 <= u(3, 3) <= 1

      -2 <= v(1) <= 2
      -2 <= v(2) <= 2
      -2 <= v(3) <= 2
```

Solve Problem

To solve the problem, call `solve`. Set an initial point `x0`.

```
rng default % For reproducibility
x0.u = 0.25*randn(3);
```

```
x0.u = x0.u + x0.u.';
x0.v = 2*randn(3,1);
[sol,fval,exitflag,output] = solve(prob,x0)

Solving problem using fmincon.

Feasible point with lower objective function value found.


Local minimum found that satisfies the constraints.

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

sol = struct with fields:
    u: [3x3 double]
    v: [3x1 double]


fval = -403.4288

exitflag =
    OptimalSolution


output = struct with fields:
        iterations: 79
         funcCount: 1266
    constrviolation: 1.1102e-16
          stepsize: 2.8033e-05
         algorithm: 'interior-point'
      firstorderopt: 6.2840e-04
       cgiterations: 126
           message: '...'
       bestfeasible: [1x1 struct]
            solver: 'fmincon'
```

View the solution.

```
disp(sol.u)

    0.9578    0.5301   -0.9277
    0.5301    0.4811    0.4510
   -0.9277    0.4510    0.8787
```

```
disp(sol.v)

    2.0000
   -2.0000
    2.0000
```

The solution matrix u is symmetric. All values of v are at the bounds.

Copyright 2018–2020 The MathWorks, Inc.

See Also
```
fcn2optimexpr | solve
```

More About

- "Problem-Based Optimization Workflow" on page 10-2
- "Problem-Based Workflow for Solving Equations" on page 10-4
- "Problem-Based Optimization Setup"

Constrained Electrostatic Nonlinear Optimization, Problem-Based

Consider the electrostatics problem of placing 20 electrons in a conducting body. The electrons will arrange themselves in a way that minimizes their total potential energy, subject to the constraint of lying inside the body. All the electrons are on the boundary of the body at a minimum. The electrons are indistinguishable, so the problem has no unique minimum (permuting the electrons in one solution gives another valid solution). This example was inspired by Dolan, Moré, and Munson [1].

The objective and nonlinear constraint functions for this example are all "Supported Operations on Optimization Variables and Expressions" on page 10-43. Therefore, `solve` uses automatic differentiation to calculate gradients. See "Automatic Differentiation in Optimization Toolbox" on page 10-41. Without automatic differentiation, this example stops early by reaching the `MaxFunctionEvaluations` tolerance. For an equivalent solver-based example using Symbolic Math Toolbox™, see "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

Problem Geometry

This example involves a conducting body defined by the following inequalities. For each electron with coordinates (x, y, z),

$$z \leq - |x| - |y|$$
$$x^2 + y^2 + (z + 1)^2 \leq 1 \, .$$

These constraints form a body that looks like a pyramid on a sphere. To view the body, enter the following code.

```
[X,Y] = meshgrid(-1:.01:1);
Z1 = -abs(X) - abs(Y);
Z2 = -1 - sqrt(1 - X.^2 - Y.^2);
Z2 = real(Z2);
W1 = Z1; W2 = Z2;
W1(Z1 < Z2) = nan; % Only plot points where Z1 > Z2
W2(Z1 < Z2) = nan; % Only plot points where Z1 > Z2
hand = figure; % Handle to the figure, for later use
set(gcf,'Color','w') % White background
surf(X,Y,W1,'LineStyle','none');
hold on
surf(X,Y,W2,'LineStyle','none');
view(-44,18)
```
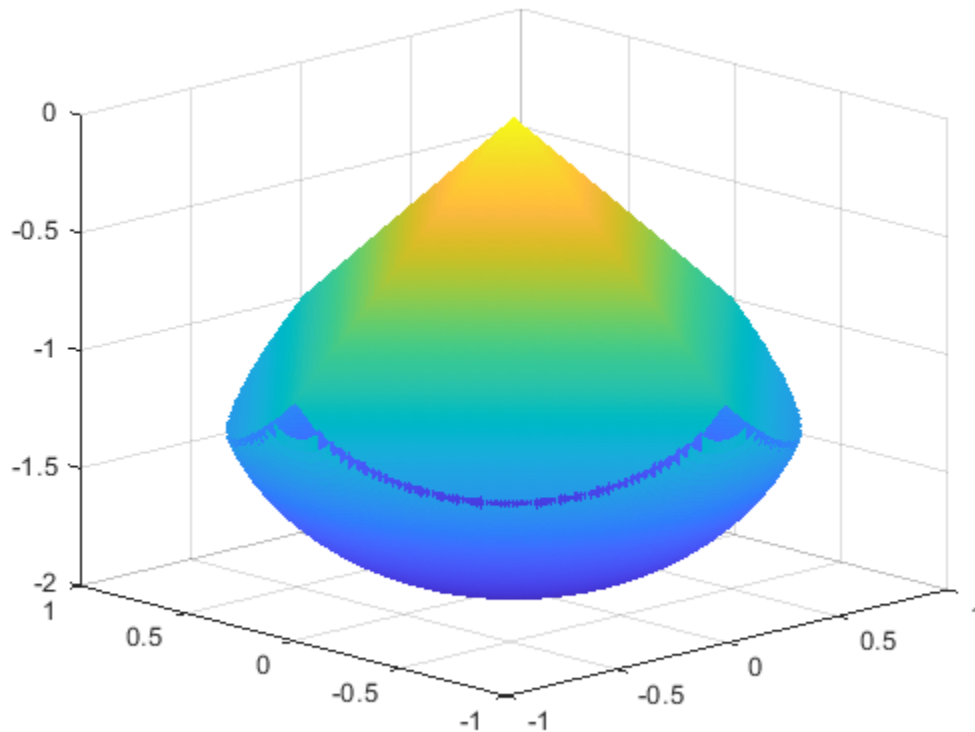
A slight gap exists between the upper and lower surfaces of the figure. This gap is an artifact of the general plotting routine used to create the figure. The routine erases any rectangular patch on one surface that touches the other surface.

Define Problem Variables

The problem has twenty electrons. The constraints give bounds on each x and y value from –1 to 1, and the z value from –2 to 0. Define the variables for the problem.

```
N = 20;
x = optimvar('x',N,'LowerBound',-1,'UpperBound',1);
y = optimvar('y',N,'LowerBound',-1,'UpperBound',1);
z = optimvar('z',N,'LowerBound',-2,'UpperBound',0);
elecprob = optimproblem;
```

Define Constraints

The problem has two types of constraints. The first, a spherical constraint, is a simple polynomial inequality for each electron separately. Define this spherical constraint.

```
elecprob.Constraints.spherec = (x.^2 + y.^2 + (z+1).^2) <= 1;
```

The preceding constraint command creates a vector of ten constraints. View the constraint vector using show.

```
show(elecprob.Constraints.spherec)
```

```
((x.^2 + y.^2) + (z + 1).^2) <= arg_RHS
```

```
where:
```

```
arg2 = 1;
arg1 = arg2(ones(1,20));
arg_RHS = arg1(:);
```

The second type of constraint in the problem is linear. You can express the linear constraints in different ways. For example, you can use the `abs` function to represent an absolute value constraint. To express the constraints this way, write a MATLAB function and convert it to an expression using `fcn2optimexpr`. See "Convert Nonlinear Function to Optimization Expression" on page 7-8. For a preferable approach that uses only differentiable functions, write the absolute value constraint as four linear inequalities. Each constraint command returns a vector of 20 constraints.

```
elecprob.Constraints.plane1 = z <= -x-y;
elecprob.Constraints.plane2 = z <= -x+y;
elecprob.Constraints.plane3 = z <= x-y;
elecprob.Constraints.plane4 = z <= x+y;
```

Define Objective Function

The objective function is the potential energy of the system, which is a sum over each electron pair of the inverse of their distances:

$$\text{energy} = \sum_{i < j} \frac{1}{\|\text{electron}(i) - \text{electron}(j)\|} \, .$$

Define the objective function as an optimization expression. For good performance, write the objective function in a vectorized fashion. See "Create Efficient Optimization Problems" on page 10-28.

```
energy = optimexpr(1);
for ii = 1:(N-1)
    jj = (ii+1):N;
    tempe = (x(ii) - x(jj)).^2 + (y(ii) - y(jj)).^2 + (z(ii) - z(jj)).^2;
    energy = energy + sum(tempe.^(-1/2));
end
elecprob.Objective = energy;
```

Run 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
x0 = randn(N,3);
for ii=1:N
    x0(ii,:) = x0(ii,:)/norm(x0(ii,:))/2;
    x0(ii,3) = x0(ii,3) - 1;
end
init.x = x0(:,1);
init.y = x0(:,2);
init.z = x0(:,3);
```

Solve the problem by calling `solve`.

```
[sol,fval,exitflag,output] = solve(elecprob,init)
```

```
Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol = struct with fields:
    x: [20x1 double]
    y: [20x1 double]
    z: [20x1 double]


fval = 163.0099

exitflag =
    OptimalSolution


output = struct with fields:
          iterations: 94
           funcCount: 150
      constrviolation: 0
            stepsize: 2.8395e-05
           algorithm: 'interior-point'
       firstorderopt: 8.1308e-06
        cgiterations: 0
             message: '...'
         bestfeasible: [1x1 struct]
              solver: 'fmincon'
```

View Solution

Plot the solution as points on the conducting body.

```
figure(hand)
plot3(sol.x,sol.y,sol.z,'r.','MarkerSize',25)
hold off
```
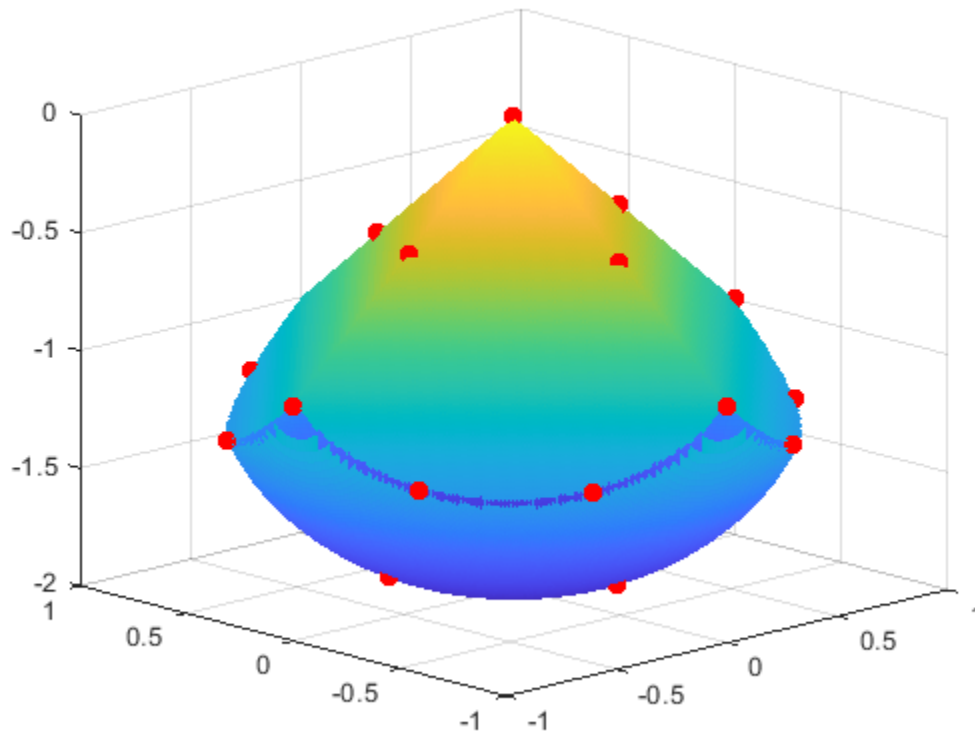
The electrons are distributed fairly evenly on the constraint boundary. Many electrons are on the edges and the pyramid point.

Reference

[1] 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.

See Also

More About

- "Problem-Based Optimization Workflow" on page 10-2
- "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97
- "Problem-Based Optimization Setup"

Problem-Based Nonlinear Minimization with Linear Constraints

This example shows how to minimize a nonlinear function subject to linear equality constraints by using the problem-based approach, where you formulate the constraints in terms of optimization variables. This example also shows how to convert an objective function file to an optimization expression by using `fcn2optimexpr`.

The example "Minimization with Linear Equality Constraints, Trust-Region Reflective Algorithm" on page 6-90 uses a solver-based approach involving the gradient and Hessian. Solving the same problem using the problem-based approach is straightforward, but takes more solution time because the problem-based approach currently does not use gradient or Hessian information.

Create Problem and Objective

The problem is 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 a set of linear equality constraints `Aeq*x = beq`. Start by creating an optimization problem and variables.

```
prob = optimproblem;
N = 1000;
x = optimvar('x',N);
```

The objective function is in the `brownfgh.m` file included in your Optimization Toolbox™ installation. You must convert the function to an optimization expression using `fcn2optimexpr` because optimization variables are excluded from appearing in an exponent. See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8.

```
prob.Objective = fcn2optimexpr(@brownfgh,x,'OutputSize',[1,1]);
```

Include Constraints

To obtain the `Aeq` and `beq` matrices in your workspace, execute this command.

```
load browneq
```

Include the linear constraints in the problem.

```
prob.Constraints = Aeq*x == beq;
```

Review and Solve Problem

Review the problem objective.

```
show(prob.Objective)
```

```
  brownfgh(x)
```

The problem has one hundred linear equality constraints, so the resulting constraint expression is too lengthy to include in the example. To show the constraints, uncomment and run the following line.

```
% show(prob.Constraints)
```

Set an initial point as a structure with field `x`.

```
x0.x = -ones(N,1);
x0.x(2:2:N) = 1;
```

Solve the problem by calling `solve`.

```
[sol,fval,exitflag,output] = solve(prob,x0)
```

```
Solving problem using fmincon.

Solver stopped prematurely.

fmincon stopped because it exceeded the function evaluation limit,
options.MaxFunctionEvaluations = 3.000000e+03.

sol = struct with fields:
    x: [1000x1 double]


fval = 207.5463

exitflag =
    SolverLimitExceeded


output = struct with fields:
           iterations: 2
            funcCount: 3007
       constrviolation: 3.7548e-13
             stepsize: 1.9303
            algorithm: 'interior-point'
        firstorderopt: 2.6432
          cgiterations: 0
              message: '...'
          bestfeasible: [1x1 struct]
               solver: 'fmincon'
```

The solver stops prematurely because it exceeds the function evaluation limit. To continue the optimization, restart the optimization from the final point, and allow for more function evaluations.

```
options = optimoptions(prob,'MaxFunctionEvaluations',1e5);
[sol,fval,exitflag,output] = solve(prob,sol,'Options',options)
```

```
Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol = struct with fields:
    x: [1000x1 double]


fval = 205.9313
```

```
exitflag =
    OptimalSolution


output = struct with fields:
          iterations: 32
           funcCount: 33061
       constrviolation: 1.4211e-14
            stepsize: 6.4910e-06
           algorithm: 'interior-point'
        firstorderopt: 5.1729e-06
          cgiterations: 0
             message: '...'
          bestfeasible: [1x1 struct]
               solver: 'fmincon'
```

Compare with Solver-Based Solution

To solve the problem using the solver-based approach as shown in "Minimization with Linear Equality Constraints, Trust-Region Reflective Algorithm" on page 6-90, convert the initial point to a vector. Then set options to use the gradient and Hessian information provided in `brownfgh`.

```
xstart = x0.x;
fun = @brownfgh;
opts = optimoptions('fmincon','SpecifyObjectiveGradient',true,'HessianFcn','objective',...
    'Algorithm','trust-region-reflective');
[x,fval,exitflag,output] = ...
   fmincon(fun,xstart,[],[],Aeq,beq,[],[],opts);

Local minimum possible.

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

fprintf("Fval = %g\nNumber of iterations = %g\nNumber of function evals = %g.\n",...
    fval,output.iterations,output.funcCount)

Fval = 205.931
Number of iterations = 22
Number of function evals = 23.
```

The solver-based solution in "Minimization with Linear Equality Constraints, Trust-Region Reflective Algorithm" on page 6-90 uses the gradients and Hessian provided in the objective function. By using that derivative information, the solver `fmincon` converges to the solution in 22 iterations, using only 23 function evaluations. The solver-based solution has the same final objective function value as this problem-based solution.

However, constructing the gradient and Hessian functions without using symbolic math is difficult and prone to error. For an example showing how to use symbolic math to calculate derivatives, see "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

See Also
`fcn2optimexpr`

More About

- "Problem-Based Optimization Workflow" on page 10-2
- "Minimization with Linear Equality Constraints, Trust-Region Reflective Algorithm" on page 6-90
- "Problem-Based Optimization Setup"

Effect of Automatic Differentiation in Problem-Based Optimization

When using automatic differentiation, the problem-based `solve` function generally requires fewer function evaluations and can operate more robustly.

By default, `solve` uses automatic differentiation to evaluate the gradients of objective and nonlinear constraint functions, when applicable. Automatic differentiation applies to functions that are expressed in terms of operations on optimization variables without using the `fcn2optimexpr` function. See "Automatic Differentiation in Optimization Toolbox" on page 10-41 and "Convert Nonlinear Function to Optimization Expression" on page 7-8.

Minimization Problem

Consider the problem of minimizing the following objective function:

$$\text{fun1} = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2$$

$$\text{fun2} = \exp\left(-\sum (x_i - y_i)^2\right)\exp(-\exp(-y_1))\text{sech}(y_2)$$

$$\text{objective} = \text{fun1} - \text{fun2}.$$

The function $\text{sech}(y) = 1/\cosh(y)$, and cosh is one of the supported functions described in "Supported Operations on Optimization Variables and Expressions" on page 10-43. Create an optimization problem representing these variables and the objective function expression.

```
prob = optimproblem;
x = optimvar('x',2);
y = optimvar('y',2);
fun1 = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
fun2 = exp(-sum((x - y).^2))*exp(-exp(-y(1)))/cosh(y(2));
prob.Objective = fun1 - fun2;
```

The minimization is subject to the nonlinear constraint $x_1^2 + x_2^2 + y_1^2 + y_2^2 \le 4$.

```
prob.Constraints.cons = sum(x.^2 + y.^2) <= 4;
```

Solve Problem and Examine Solution Process

Solve the problem starting from an initial point.

```
init.x = [-1;2];
init.y = [1;-1];
[xproblem,fvalproblem,exitflagproblem,outputproblem] = solve(prob,init);
```

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

```
disp(fvalproblem)
```

    -0.5500

```
disp(outputproblem.funcCount)

    77
```

```
disp(outputproblem.iterations)

    46
```

The `output` structure shows that `solve` calls `fmincon`, which requires 77 function evaluations and 46 iterations to solve the problem. The objective function value at the solution is `fvalproblem =` `-0.55`.

Solve Problem Without Automatic Differentiation

To determine the efficiency gains from automatic differentiation, set `solve` name-value pair arguments to use finite difference gradients instead.

```
[xfd,fvalfd,exitflagfd,outputfd] = solve(prob,init,...
    "ObjectiveDerivative",'finite-differences',"ConstraintDerivative",'finite-differences');
```

```
Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

```
disp(fvalfd)

   -0.5500
```

```
disp(outputfd.funcCount)

   269
```

```
disp(outputfd.iterations)

    47
```

Using a finite difference gradient approximation causes `solve` to take 269 function evaluations compared to 77. The number of iterations is nearly the same, as is the reported objective function value at the solution. The final solution points are the same to display precision.

```
disp([xproblem.x,xproblem.y])

   0.8671    1.0433
   0.7505    0.5140
```

```
disp([xfd.x,xfd.y])

   0.8671    1.0433
   0.7505    0.5140
```

In summary, the main effect of automatic differentiation in optimization is to lower the number of function evaluations.

See Also
`solve`

More About

Supply Derivatives in Problem-Based Workflow

Why Include Derivatives?

This example shows how to include derivative information in nonlinear problem-based optimization when automatic derivatives do not apply, or when you want to include a Hessian, which is not provided using automatic differentiation. Including gradients or a Hessian in an optimization can give the following benefits:

- More robust results. Finite differencing steps sometimes reach points where the objective or a nonlinear constraint function is undefined, not finite, or complex.

- Increased accuracy. Analytic gradients can be more accurate than finite difference estimates.

- Faster convergence. Including a Hessian can lead to faster convergence, meaning fewer iterations.

- Improved performance. Analytic gradients can be faster to calculate than finite difference estimates, especially for problems with a sparse structure. For complicated expressions, however, analytic gradients can be slower to calculate.

Automatic Differentiation Applied to Optimization

Starting in R2020b, the `solve` function can use automatic differentiation for supported functions in order to supply gradients to solvers. These automatic derivatives apply only to gradients (first derivatives), not Hessians (second derivatives).

Automatic differentiation applies when you do not use `fcn2optimexpr` to create an objective or constraint function. If you need to use `fcn2optimexpr`, this example shows how to include derivative information.

The way to use derivatives in problem-based optimization without automatic differentiation is to convert your problem using `prob2struct`, and then edit the resulting objective and constraint functions. This example shows a hybrid approach where automatic differentiation supplies derivatives for part of the objective function.

Create Optimization Problem

With 2-D control variables x and y, use the objective function

$$\text{fun1} = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2$$

$$\text{fun2} = \exp\left(-\sum(x_i - y_i)^2\right)\exp(-\exp(-y_1))\text{sech}(y_2)$$

$$\text{objective} = \text{fun1} - \text{fun2}.$$

Include the constraint that the sum of squares of x and y is no more than 4.

fun2 is not based on supported functions for optimization expressions; see "Supported Operations on Optimization Variables and Expressions" on page 10-43. Therefore, to include fun2 in an optimization problem, you must convert it to an optimization expression using fcn2optimexpr. (fun2 is not supported because it uses the unsupported sech function. However, sech(x) = 1/cosh(x), and cosh is a supported function. To use 1/cosh and automatic differentiation, see "Effect of Automatic Differentiation in Problem-Based Optimization" on page 7-23.)

To use AD on the supported functions, set up the problem without the unsupported function fun2, and include fun2 later.

```
prob = optimproblem;
x = optimvar('x',2);
y = optimvar('y',2);
fun1 = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
fun2 = @(x,y)exp(-sum((x - y).^2))*exp(-exp(-y(1)))*sech(y(2));
prob.Objective = fun1;
prob.Constraints.cons = sum(x.^2 + y.^2) <= 4;
```

Convert Problem to Solver-Based Form

To include derivatives of fun2, first convert the problem without fun2 to a structure using prob2struct.

```
problem = prob2struct(prob,...
    'ObjectiveFunctionName','generatedObjectiveBefore');
```

During the conversion, prob2struct creates function files that represent the objective and nonlinear constraint functions. By default, these functions have the names 'generatedObjective.m' and 'generatedConstraints.m'. The objective function file without fun2 is 'generatedObjectiveBefore.m'.

The generated objective and constraint functions include gradients.

Calculate Derivatives and Keep Track of Variables

Calculate the derivatives associated with fun2. If you have a Symbolic Math Toolbox license, you can use the gradient or jacobian function to help compute the derivatives. See "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

The solver-based approach has one control variable. Each optimization variable (x and y, in this example) is a portion of the control variable. You can find the mapping between optimization variables and the single control variable in the generated objective function file 'generatedObjectiveBefore.m'. The beginning of the file contains these lines of code or similar lines.

```
%% Variable indices.
xidx = 1:2;
```

```
yidx = 3:4;

%% Map solver-based variables to problem-based.
x = inputVariables(xidx);
x = x(:);
y = inputVariables(yidx);
y = y(:);
```

In this code, the control variable has the name `inputVariables`.

Alternatively, you can find the mapping by using `varindex`.

```
idx = varindex(prob);
disp(idx.x)
```

```
     1     2
```

```
disp(idx.y)
```

```
     3     4
```

The full objective function includes the negative of `fun2`:

```
fun2 = @(x,y)exp(-sum((x - y).^2))*exp(-exp(-y(1)))*sech(y(2));
```

Using standard calculus, calculate `gradfun2`, the gradient of `fun2`.

$$
\text{gradfun2} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ -\sigma_1 + d\exp(-y_1) \\ -\sigma_2 - d\tanh(y_2) \end{bmatrix},
$$

where

$$
\begin{aligned}
d &= \text{fun2}(x; y) \\
\sigma_1 &= -2(x_1 - y_1)d \\
\sigma_2 &= -2(x_2 - y_2)d \,.
\end{aligned}
$$

Edit the Objective and Constraint Files

Edit `'generatedObjectiveBefore.m'` to include `fun2`.

```
%% Compute objective function.
arg2 = x(1);
arg4 = (x(2) - arg2.^2);
arg5 = 100;
arg6 = arg4.^2;
arg8 = (1 - x(1));
obj = ((arg5 .* arg6) + arg8.^2);
fun2 = exp(-sum((x - y).^2))*exp(-exp(-y(1)))*sech(y(2));
obj = obj - fun2;
```

Include the calculated gradients in the objective function file by editing `'generatedObjectiveBefore.m'`. If you have a software version that does not perform the

gradient calculation, include all of these lines. If your software performs the gradient calculation, include only the bold lines, which calculate the gradient of `fun2`.

```
%% Compute objective gradient.
if nargout > 1
    arg9 = 1;
    arg10 = (-(arg9.*2.*(arg8(:))));
    arg11 = zeros([2,size(arg10,2)]);
    arg11(1,:) = arg10;
    arg12 = ((arg9.*arg5(:)).*2.*(arg4(:)));
    arg13 = zeros([2,size(arg12,2)]);
    arg13(2,:) = arg12;
    arg14 = ((-((arg9.*arg5(:)).*2.*(arg4(:)))).*2.*(arg2(:)));
    arg15 = zeros([2,size(arg14,2)]);
    arg15(1,:) = arg14;
    arg16 = zeros([4, 1]);
    arg16(xidx,:) = arg11 + arg13 + arg15;
    grad = arg16(:);
    % Calculate gradient of fun2
    d = fun2;
    sigma1 = -2*(x(1) - y(1))*d;
    sigma2 = -2*(x(2) - y(2))*d;
    gradfun2 = zeros(4, 1);
    gradfun2(1) = sigma1;
    gradfun2(2) = sigma2;
    gradfun2(3) = -sigma1 + d*exp(-y(1));
    gradfun2(4) = -sigma2 - d*tanh(y(2));
    % Subtract the gradient of fun2
    grad = grad - gradfun2;
end
```

Recall that the nonlinear constraint is `x(1)^2 + x(2)^2 + y(1)^2 + y(2)^2 <= 4`. The gradient of this constraint function is `2*[x;y]`. If your software calculates the constraint gradient and includes it in the generated constraint file, then you do not need to do anything more. If your software does not calculate the constraint gradient, then include the gradient of the nonlinear constraint by editing `'generatedConstraints.m'` to include these lines.

```
%% Insert gradient calculation here.
% If you include a gradient, notify the solver by setting the
% SpecifyConstraintGradient option to true.

if nargout > 2
    cineqGrad = 2*[x;y];
    ceqGrad = [];
end
```

Run Problem With and Without Gradients

Run the problem using both the solver-based and problem-based (no gradient) methods to see the differences. To run the solver-based problem using derivative information, create appropriate options in the problem structure.

```
options = optimoptions('fmincon','SpecifyObjectiveGradient',true,...
    'SpecifyConstraintGradient',true);
problem.options = options;
```

Nonlinear problems require a nonempty `x0` field in the problem structure.

```
x0 = [-1;2;1;-1];
problem.x0 = x0;
```

Call `fmincon` on the problem structure.

```
[xsolver,fvalsolver,exitflagsolver,outputsolver] = fmincon(problem)
```

```
Local minimum found that satisfies the constraints.

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

<stopping criteria details>

xsolver =

    0.8671
    0.7505
    1.0433
    0.5140


fvalsolver =

   -0.5500


exitflagsolver =

     1


outputsolver =

  struct with fields:

         iterations: 46
          funcCount: 77
      constrviolation: 0
           stepsize: 7.4091e-06
          algorithm: 'interior-point'
      firstorderopt: 7.5203e-07
        cgiterations: 9
            message: '↵Local minimum found that satisfies the constraints.↵Optimization complete
```

Compare this solution with the one obtained from `solve` without derivative information.

```
init.x = x0(1:2);
init.y = x0(3:4);
prob.Objective = prob.Objective - fcn2optimexpr(fun2,x,y);
[xproblem,fvalproblem,exitflagproblem,outputproblem] = solve(prob,init,...
    "ConstraintDerivative","finite-differences",...
    "ObjectiveDerivative","finite-differences")
```

```
Solving problem using fmincon.

Local minimum found that satisfies the constraints.
```

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

<stopping criteria details>

xproblem =

  struct with fields:

    x: [2×1 double]
    y: [2×1 double]


fvalproblem =

  -0.5500


exitflagproblem =

    OptimalSolution


outputproblem =

  struct with fields:

         iterations: 47
          funcCount: 269
      constrviolation: 0
           stepsize: 8.9896e-08
          algorithm: 'interior-point'
       firstorderopt: 1.3377e-08
         cgiterations: 9
            message: '↵Local minimum found that satisfies the constraints.↵Optimization complete
        bestfeasible: [1×1 struct]
             solver: 'fmincon'
```

Both solutions report the same function value to display precision, and both require roughly the same number of iterations (46 using gradient information, 47 without). However, the solution with gradients requires only 77 function evaluations, compared to 269 for the solution without gradients.

Include Hessian

To include a Hessian, you must use `prob2struct`, even if all your functions are supported for optimization expressions. This example shows how to use a Hessian for the `fmincon interior-point` algorithm. The `fminunc trust-region` algorithm and the `fmincon trust-region-reflective` algorithm use a different syntax; see "Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms" on page 2-21.

As described in "Hessian for fmincon interior-point algorithm" on page 2-21, the Hessian is the Hessian of the Lagrangian.

$$\nabla^2_{xx} L(x, \lambda) = \nabla^2 f(x) + \sum \lambda_{g,i} \nabla^2 g_i(x) + \sum \lambda_{h,i} \nabla^2 h_i(x) \,. \tag{7-1}$$

Include this Hessian by creating a function file to compute the Hessian, and creating a `HessianFcn` option for `fmincon` to use the Hessian. To create the Hessian in this case, create the second derivatives of the objective and nonlinear constraints separately.

The second derivative matrix of the objective function for this example is complicated. Its objective function listing `hessianfun(x)` was created by Symbolic Math Toolbox using the same approach as described in "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

```
function hessfun = hessianfun(in1)
%HESSIANFUN
%     HESSFUN = HESSIANFUN(IN1)

%     This function was generated by Symbolic Math Toolbox version 8.5.
%     15-Mar-2020 10:12:23

x1 = in1(1,:);
x2 = in1(2,:);
y1 = in1(3,:);
y2 = in1(4,:);
t2 = cosh(y2);
t3 = sinh(y2);
t4 = x1.*2.0;
t5 = x2.*2.0;
t6 = y1.*2.0;
t7 = y2.*2.0;
t8 = -y1;
t10 = -y2;
t15 = x1.*4.0e+2;
t9 = -t6;
t11 = -t7;
t12 = 1.0./t2;
t14 = exp(t8);
t16 = t8+x1;
t17 = t10+x2;
t18 = -t15;
t13 = t12.^2;
t19 = t4+t9;
t20 = t5+t11;
t21 = t16.^2;
t22 = t17.^2;
t23 = -t14;
t24 = exp(t23);
t25 = t19.^2;
t26 = t20.^2;
t27 = -t21;
t28 = -t22;
t29 = t27+t28;
t30 = exp(t29);
t31 = t12.*t24.*t30.*2.0;
t33 = t3.*t13.*t14.*t24.*t30;
t34 = t3.*t13.*t19.*t24.*t30;
t35 = t3.*t13.*t20.*t24.*t30;
t36 = t12.*t24.*t25.*t30;
t37 = t12.*t24.*t26.*t30;
t42 = t12.*t14.*t19.*t24.*t30;
t43 = t12.*t14.*t20.*t24.*t30;
t44 = t12.*t20.*t23.*t24.*t30;
```

```
t45 = t12.*t19.*t20.*t24.*t30;
t32 = -t31;
t38 = -t34;
t39 = -t35;
t40 = -t36;
t41 = -t37;
t46 = -t45;
t49 = t43+t45;
t47 = t18+t46;
t48 = t38+t45;
t50 = t32+t37+t39;
t51 = t32+t36+t42;
t52 = t33+t34+t44+t46;
hessfun = reshape([t31+t40-x2.*4.0e+2+x1.^2.*1.2e+3+2.0,t47,...
    t51,t48,t47,t31+t41+2.0e+2,t49,t50,t51,...
    t49,t31+t40-t42.*2.0+t12.*t14.*t24.*t30-t12.*t24.*t30.*exp(t9),...
    t52,t48,t50,t52,t35.*2.0+t41+t12.*t24.*t30.*3.0-t3.^2.*t12.^3.*t24.*t30.*2.0],...
    [4,4]);
```

In contrast, the Hessian of the nonlinear inequality constraint is simple; it is twice the identity matrix.

```
hessianc = 2*eye(4);
```

Create the Hessian of the Lagrangian as a function handle.

```
H = @(x,lam)(hessianfun(x) + hessianc*lam.ineqnonlin);
```

Create options to use this Hessian.

```
problem.options.HessianFcn = H;
```

Solve the problem and display the number of iterations and number of function evaluations. The solution is approximately the same as before.

```
[xhess,fvalhess,exitflaghess,outputhess] = fmincon(problem);
disp(outputhess.iterations)
disp(outputhess.funcCount)
```

```
Local minimum found that satisfies the constraints.

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

    31

    59
```

This time, `fmincon` takes only 31 iterations compared to over 45, and only 59 function evaluations compared to over 75. In summary, providing an analytic Hessian calculation can improve the efficiency of the solution process, but developing a function to calculate the Hessian can be difficult.

See Also
`fcn2optimexpr` | `prob2struct` | `varindex`

More About

- "Effect of Automatic Differentiation in Problem-Based Optimization" on page 7-23
- "Including Gradients and Hessians" on page 2-19
- "Obtain Generated Function Details" on page 7-35

Obtain Generated Function Details

This example shows how to find the values of extra parameters in functions generated by `prob2struct`.

Create a nonlinear problem and convert the problem to a structure using `prob2struct`. Name the generated objective function and nonlinear constraint function.

```
x = optimvar('x',2);
fun = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
prob = optimproblem('Objective',fun);
mycon = dot(x,x) <= 4;
prob.Constraints.mycon = mycon;
x0.x = [-1;1.5];
problem = prob2struct(prob,x0,'ObjectiveFunctionName','rosenbrock',...
    'ConstraintFunctionName','circle2');
```

Examine the first line of the generated constraint function `circle2`.

```
type circle2

function [cineq, ceq, cineqGrad, ceqGrad] = circle2(inputVariables, extraParams)
%circle2 Compute constraint values and gradients
%
%   [CINEQ, CEQ] = circle2(INPUTVARIABLES, EXTRAPARAMS) computes the
%   inequality constraint values CINEQ and the equality constraint values
%   CEQ at the point INPUTVARIABLES, using the extra parameters in
%   EXTRAPARAMS.
%
%   [CINEQ, CEQ, CINEQGRAD, CEQGRAD] = circle2(INPUTVARIABLES,
%   EXTRAPARAMS) additionally computes the inequality constraint gradient
%   values CINEQGRAD and the equality constraint gradient values CEQGRAD
%   at the current point.
%
%   Auto-generated by prob2struct on 04-Jun-2020 07:51:14

%% Compute inequality constraints.
Hineq = extraParams{1};
fineq = extraParams{2};
rhsineq = extraParams{3};
Hineqmvec = Hineq*inputVariables(:);
cineq = 0.5*dot(inputVariables(:), Hineqmvec) + dot(fineq, inputVariables(:)) + rhsineq;


%% Compute equality constraints.
ceq = [];

%% Compute constraint gradients.
if nargout > 2
    cineqGrad = Hineqmvec + fineq;
    ceqGrad = [];
end

end
```

The `circle2` function has a second input named `extraParams`. To find the values of this input, use the `functions` function on the function handle stored in `problem.nonlcon`.

```
F = functions(problem.nonlcon)

F = struct with fields:
            function: '@(x)circle2(x,extraParams)'
                type: 'anonymous'
                file: 'G:\jobarchive\Boptim\2020_06_03_h21m02s52_job1400917_pass\matlab\toolbox\
           workspace: {[1×1 struct]}
    within_file_path: ''
```

To access the extra parameters, view the `workspace` field of `F`.

```
ws = F.workspace

ws = 1×1 cell array
    {1×1 struct}
```

Continue to extract the information at deeper levels until you see all the extra parameters.

```
ws1 = ws{1}

ws1 = struct with fields:
    extraParams: {[2×2 double]  [2×1 double]  [-4]}
```

```
ep = ws1.extraParams

ep=1×3 cell array
    {2×2 double}    {2×1 double}    {[-4]}
```

```
ep{1}

ans =
    (1,1)        2
    (2,2)        2
```

```
ep{2}

ans =
   All zero sparse: 2×1
```

```
ep{3}

ans = -4
```

Now you can read the `circle2` file listing and understand what all of the variables mean.

```
Hineq = 2*speye(2);
fineq = sparse([0;0]);
rhsineq = -4;
```

See Also
```
prob2struct
```

More About

- "Problem-Based Optimization Workflow" on page 10-2
- "Problem-Based Workflow for Solving Equations" on page 10-4

Output Function for Problem-Based Optimization

This example shows how to use an output function to plot and store the history of the iterations for a nonlinear problem. This history includes the evaluated points, the search directions that the solver uses to generate points, and the objective function values at the evaluated points.

For the solver-based approach to this example, see "Output Functions for Optimization Toolbox™" on page 3-30.

Plot functions have the same syntax as output functions, so this example also applies to plot functions, too.

For both the solver-based approach and for the problem-based approach, write the output function as if you are using the solver-based approach. In the solver-based approach, you use a single vector variable, usually denoted x, instead of a collection of optimization variables of various sizes. So to write an output function for the problem-based approach, you must understand the correspondence between your optimization variables and the single solver-based x. To map between optimization variables and x, use `varindex`. In this example, to avoid confusion with an optimization variable named x, use "in" as the vector variable name.

Problem Description

The problem is to minimize the following function of variables x and y:

$$f = \exp(x)\left(4x^2 + 2y^2 + 4xy + 2y + 1\right).$$

In addition, the problem has two nonlinear constraints:

$$x + y - xy \geq 1.5$$
$$xy \geq 10.$$

Problem-Based Setup

To set up the problem in the problem-based approach, define optimization variables and an optimization problem object.

```
x = optimvar('x');
y = optimvar('y');
prob = optimproblem;
```

Define the objective function as an expression in the optimization variables.

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

Include the objective function in `prob`.

```
prob.Objective = f;
```

To include the nonlinear constraints, create optimization constraint expressions.

```
cons1 = x + y - x*y >= 1.5;
cons2 = x*y >= -10;
prob.Constraints.cons1 = cons1;
prob.Constraints.cons2 = cons2;
```

Because this is a nonlinear problem, you must include an initial point structure x0. Use x0.x = −1 and x0.y = 1.

```
x0.x = -1;
x0.y = 1;
```

Output Function

The outfun output function records a history of the points generated by fmincon during its iterations. The output function also plots the points and keeps a separate history of the search directions for the sqp algorithm. The search direction is a vector from the previous point to the next point that fmincon tries. During its final step, the output function saves the history in workspace variables, and saves a history of the objective function values at each iterative step.

For the required syntax of optimization output functions, see "Output Function and Plot Function Syntax" on page 15-27.

An output function takes a single vector variable as an input. But the current problem has two variables. To find the mapping between the optimization variables and the input variable, use varindex.

```
idx = varindex(prob);
idx.x
```

```
ans = 1
```

```
idx.y
```

```
ans = 2
```

The mapping shows that x is variable 1 and y is variable 2. So, if the input variable is named in, then x = in(1) and y = in(2).

```
type outfun
```

```
function stop = outfun(in,optimValues,state,idx)
    persistent history searchdir fhistory
    stop = false;

    switch state
        case 'init'
            hold on
            history = [];
            fhistory = [];
            searchdir = [];
        case 'iter'
        % Concatenate current point and objective function
        % value with history. in must be a row vector.
            fhistory = [fhistory; optimValues.fval];
            history = [history; in(:)']; % Ensure in is a row vector
        % Concatenate current search direction with
        % searchdir.
            searchdir = [searchdir;...
                        optimValues.searchdirection(:)'];
            plot(in(idx.x),in(idx.y),'o');
        % Label points with iteration number and add title.
        % Add .15 to idx.x to separate label from plotted 'o'
            text(in(idx.x)+.15,in(idx.y),...
```

```
                num2str(optimValues.iteration));
            title('Sequence of Points Computed by fmincon');
        case 'done'
            hold off
            assignin('base','optimhistory',history);
            assignin('base','searchdirhistory',searchdir);
            assignin('base','functionhistory',fhistory);
        otherwise
    end
end
```

Include the output function in the optimization by setting the `OutputFcn` option. Also, set the `Algorithm` option to use the `'sqp'` algorithm instead of the default `'interior-point'` algorithm. Pass `idx` to the output function as an extra parameter in the last input. See "Passing Extra Parameters" on page 2-57.

```
outputfn = @(in,optimValues,state)outfun(in,optimValues,state,idx);
opts = optimoptions('fmincon','Algorithm','sqp','OutputFcn',outputfn);
```

Run Optimization Using Output Function

Run the optimization, including the output function, by using the `'Options'` name-value pair argument.

```
[sol,fval,eflag,output] = solve(prob,x0,'Options',opts)
```
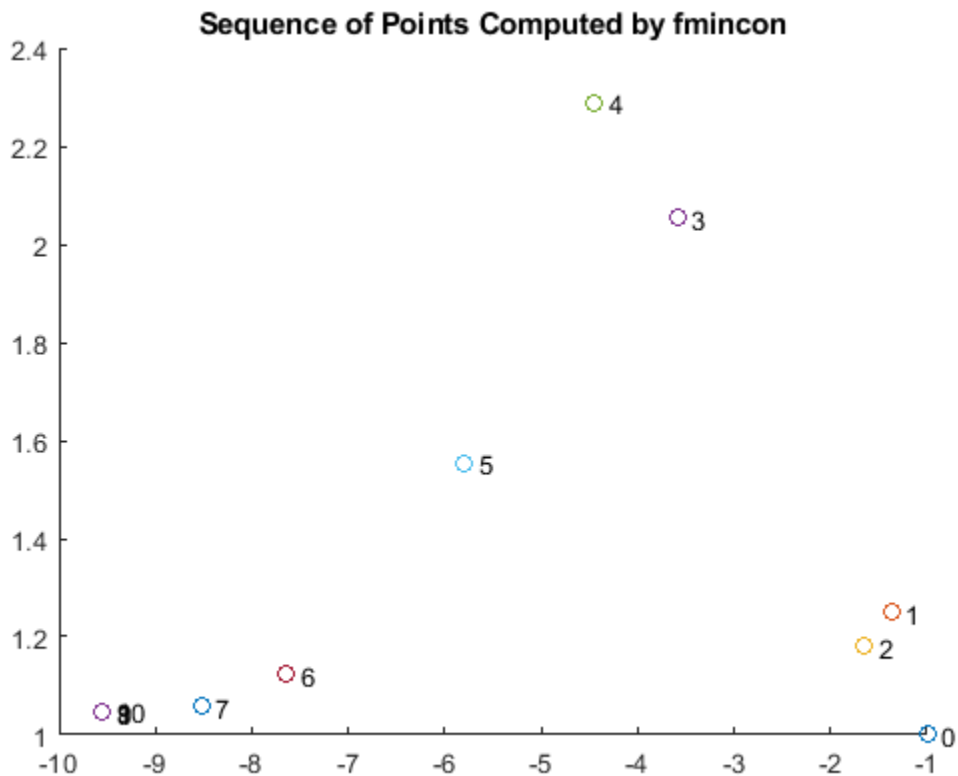
```
Solving problem using fmincon.
```

```
Local minimum found that satisfies the constraints.

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

sol = struct with fields:
    x: -9.5474
    y: 1.0474


fval = 0.0236

eflag =
    OptimalSolution


output = struct with fields:
         iterations: 10
          funcCount: 22
          algorithm: 'sqp'
            message: '...'
     constrviolation: 1.2434e-14
           stepsize: 1.4785e-07
        lssteplength: 1
       firstorderopt: 7.1930e-10
        bestfeasible: [1x1 struct]
              solver: 'fmincon'
```

Examine the iteration history. Each row of the `optimhistory` matrix represents one point. The last few points are very close, which explains why the plotted sequence shows overprinted numbers for points 8, 9, and 10.

```
disp('Locations');disp(optimhistory)
```

```
Locations
   -1.0000    1.0000
   -1.3679    1.2500
   -1.6509    1.1813
   -3.5870    2.0537
   -4.4574    2.2895
   -5.8015    1.5531
   -7.6498    1.1225
   -8.5223    1.0572
   -9.5463    1.0464
   -9.5474    1.0474
   -9.5474    1.0474
```

Examine the `searchdirhistory` and `functionhistory` arrays.

```
disp('Search Directions');disp(searchdirhistory)
```

```
Search Directions
        0         0
   -0.3679    0.2500
   -0.2831   -0.0687
   -1.9360    0.8725
   -0.8704    0.2358
```

```
     -1.3441    -0.7364
     -2.0877    -0.6493
     -0.8725    -0.0653
     -1.0241    -0.0108
     -0.0011     0.0010
      0.0000    -0.0000
```

disp('Function Values');disp(functionhistory)

```
Function Values
     1.8394
     1.8513
     1.7757
     0.9839
     0.6343
     0.3250
     0.0978
     0.0517
     0.0236
     0.0236
     0.0236
```

Unsupported Functions Require `fcn2optimexpr`

If your objective function or nonlinear constraint functions are not composed of elementary functions, you must convert the functions to optimization expressions using `fcn2optimexpr`. See "Convert Nonlinear Function to Optimization Expression" on page 7-8. For the present example:

```
fun = @(x,y)exp(x)*(4*x^2 + 2*y^2 + 4*x*y + 2*y + 1);
f = fcn2optimexpr(fun,x,y);
```

For the list of supported functions, see "Supported Operations on Optimization Variables and Expressions" on page 10-43.

See Also
varindex

More About
- "Output Functions for Optimization Toolbox™" on page 3-30
- "Problem-Based Optimization Workflow" on page 10-2
- "Problem-Based Optimization Setup"

Solve Nonlinear Feasibility Problem, Problem-Based

This example shows how to find a point that satisfies all the constraints in a problem, with no objective function to minimize.

Problem Definition

For example, suppose that you have the following constraints:

$$(y + x^2)^2 + 0.1y^2 \le 1$$
$$y \le \exp(-x) - 3$$
$$y \le x - 4 .$$

Do any points (x, y) satisfy all of the constraints?

Problem-Based Solution

Create an optimization problem that has only constraints, no objective function.

```
x = optimvar('x');
y = optimvar('y');
prob = optimproblem;
cons1 = (y + x^2)^2 + 0.1*y^2 <= 1;
cons2 = y <= exp(-x) - 3;
cons3 = y <= x - 4;
prob.Constraints.cons1 = cons1;
prob.Constraints.cons2 = cons2;
prob.Constraints.cons3 = cons3;
show(prob)

  OptimizationProblem :

    Solve for:
       x, y

    minimize :

    subject to cons1:
       ((y + x.^2).^2 + (0.1 .* y.^2)) <= 1

    subject to cons2:
       y <= (exp(-x) - 3)

    subject to cons3:
       y - x <= -4
```

Create a pseudorandom start point structure x0 with fields x and y for the optimization variables.

```
rng default
x0.x = randn;
x0.y = randn;
```

Solve the problem starting from x0.

```
[sol,~,exitflag,output] = solve(prob,x0)
```

```
Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol = struct with fields:
    x: 1.7903
    y: -3.0102


exitflag =
    OptimalSolution


output = struct with fields:
           iterations: 6
            funcCount: 9
        constrviolation: 0
             stepsize: 0.2906
            algorithm: 'interior-point'
         firstorderopt: 0
          cgiterations: 0
               message: '...'
           bestfeasible: [1x1 struct]
                solver: 'fmincon'
```

The solver finds a feasible point.

Initial Point Importance

The solver can fail to find a solution when starting from some initial points. Set the initial point x0.x = -1, x0.y = -4 and solve the problem starting from x0.

```
x0.x = -1;
x0.y = -4;
[sol2,~,exitflag2,output2] = solve(prob,x0)
```

```
Solving problem using fmincon.

Converged to an infeasible point.

fmincon stopped because it is unable to find a point locally that satisfies
the constraints within the value of the constraint tolerance.

sol2 = struct with fields:
    x: -2.1266
    y: -4.6657


exitflag2 =
    NoFeasiblePointFound


output2 = struct with fields:
           iterations: 147
```

```
        funcCount: 315
    constrviolation: 1.4609
           stepsize: 9.7766e-11
          algorithm: 'interior-point'
      firstorderopt: 0
        cgiterations: 297
            message: '...'
         bestfeasible: []
              solver: 'fmincon'
```

Check the infeasibilities at the returned point.

```
inf1 = infeasibility(cons1,sol2)
```

```
inf1 = 1.1974
```

```
inf2 = infeasibility(cons2,sol2)
```

```
inf2 = 0
```

```
inf3 = infeasibility(cons3,sol2)
```

```
inf3 = 1.4609
```

Both `cons1` and `cons3` are infeasible at the solution `sol2`. The results highlight the importance of using multiple start points to investigate and solve a feasibility problem.

Visualize Constraints

To visualize the constraints, plot the points where each constraint function is zero by using `fimplicit`. The `fimplicit` function passes numeric values to its functions, whereas the `evaluate` function requires a structure. To tie these functions together, use the `evaluateExpr` helper function, which appears at the end of this example on page 7-0 This function simply puts passed values into a structure with the appropriate names.

Note: If you use the live script file for this example, the `evaluateExpr` function is already included at the end of the file. Otherwise, you need to create this function at the end of your .m file or add it as a file on the MATLAB® path.

Avoid a warning that occurs because the `evaluateExpr` function does not work on vectorized inputs.

```
s = warning('off','MATLAB:fplot:NotVectorized');
cc1 = (y + x^2)^2 + 0.1*y^2 - 1;
fimplicit(@(a,b)evaluateExpr(cc1,a,b),[-2 2 -4 2],'r')
hold on
cc2 = y - exp(-x) + 3;
fimplicit(@(a,b)evaluateExpr(cc2,a,b),[-2 2 -4 2],'k')
cc3 = y - x + 4;
fimplicit(@(x,y)evaluateExpr(cc3,x,y),[-2 2 -4 2],'b')
hold off
```
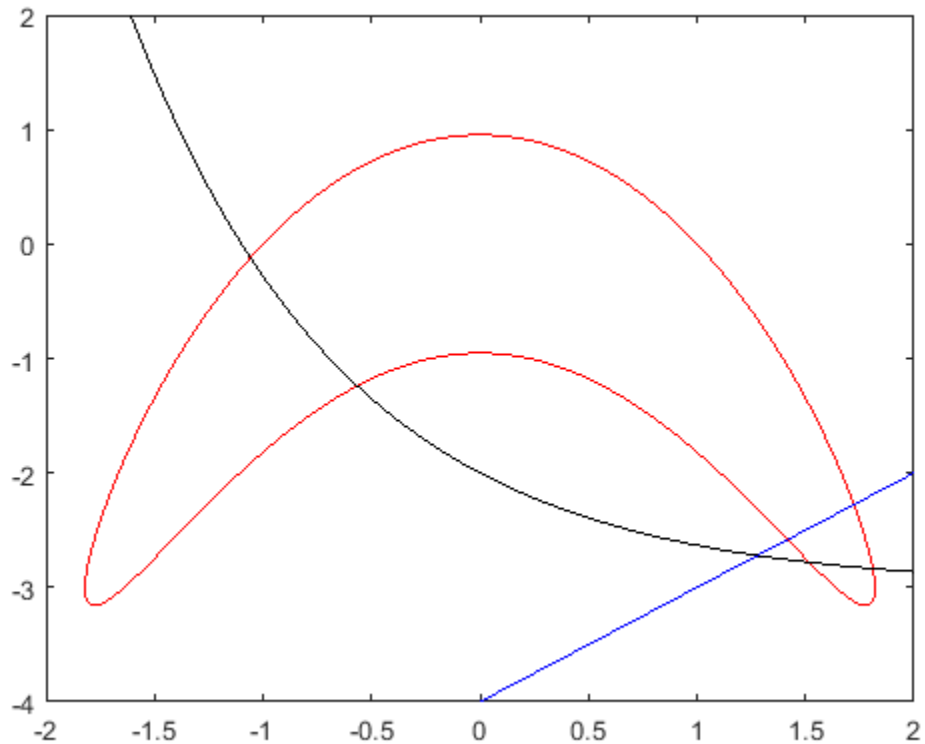
```
warning(s);
```

The feasible region is inside the red outline and below the black and blue lines. The feasible region is at the lower right of the red outline.

Helper Function

This code creates the `evaluateExpr` helper function.

```
function p = evaluateExpr(expr,x,y)
pt.x = x;
pt.y = y;
p = evaluate(expr,pt);
end
```

See Also
"Problem-Based Optimization Workflow" on page 10-2

More About
- "Investigate Linear Infeasibilities" on page 9-158
- "Solve Feasibility Problem" (Global Optimization Toolbox)
- "Problem-Based Optimization Workflow" on page 10-2

8

Multiobjective Algorithms and Examples

Multiobjective Optimization Algorithms

In this section...
"Multiobjective Optimization Definition" on page 8-2
"Algorithms" on page 8-3

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

  $$\frac{F_i(x) - F^*_i}{w_i} . \tag{8-1}$$

  This minimization is supposed to be accomplished while satisfying all types of constraints: $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, and $l \le x \le u$.

  If you set all weights equal to 1 (or any other positive constant), the goal attainment problem is the same as the unscaled goal attainment problem. If the F^*_i are positive, and you set all weights as $w_i = F^*_i$, the goal attainment problem becomes minimizing the relative difference between the functions $F_i(x)$ and the goals F^*_i.

  In other words, the goal attainment problem is to minimize a slack variable γ, defined as the maximum over i of the expressions in "Equation 8-1". This implies the expression that is the formal statement of the goal attainment problem:

  $$\min_{x,\, \gamma} \gamma$$

  such that $F(x) - w \cdot \gamma \le F^*$, $c(x) \le 0$, $ceq(x) = 0$, $A \cdot x \le b$, $Aeq \cdot x = beq$, and $l \le x \le u$.

- `fminimax` addresses the problem of minimizing the maximum of a set of nonlinear functions, subject to all types of constraints:

  $$\min_{x} \max_{i} F_i(x)$$

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

  Clearly, this problem is a special case of the unscaled goal attainment problem, with $F^*_i = 0$ and $w_i = 1$.

Algorithms

Goal Attainment Method

This section describes the goal attainment method of Gembicki [3]. This method uses a set of design goals, $F^* = \{F_1^*, F_2^*, ..., F_m^*\}$, 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

$$\underset{\gamma \in \Re,\ x \in \Omega}{\text{minimize}} \gamma \tag{8-2}$$

such that $F_i(x) - w_i\gamma \le F_i^*, \quad i = 1, ..., m$.

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.
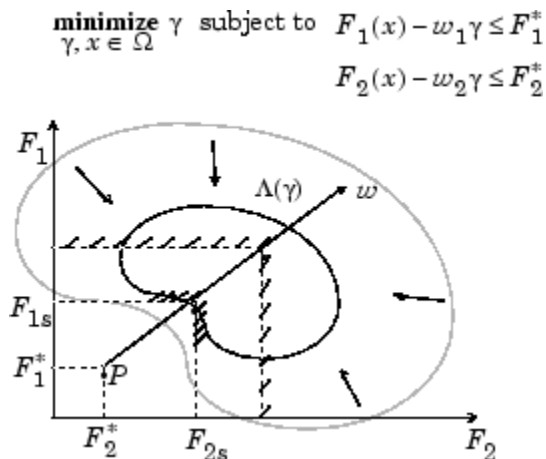
Figure 8-1, Geometrical Representation of the Goal Attainment Method

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

Algorithm Improvements for the Goal Attainment Method

The goal attainment method has the advantage that it can be posed as a nonlinear programming problem. Characteristics of the problem can also be exploited in a nonlinear programming algorithm.

In sequential quadratic programming (SQP), the choice of merit function for the line search is not easy because, in many cases, it is difficult to "define" the relative importance between improving the objective function and reducing constraint violations. This has resulted in a number of different schemes for constructing the merit function (see, for example, Schittkowski [36]). In goal attainment programming there might be a more appropriate merit function, which you can achieve by posing "Equation 8-2" as the minimax problem

$$\underset{x \in \mathfrak{R}^n}{\text{minimize}} \ \underset{i}{\max}\{\Lambda_i\}, \tag{8-3}$$

where

$$\Lambda_i = \frac{F_i(x) - F_i^*}{w_i}, \quad i = 1, ..., m.$$

Following the argument of Brayton et al. [1] for minimax optimization using SQP, using the merit function of "Equation 6-44" for the goal attainment problem of "Equation 8-3" gives

$$\psi(x, \gamma) = \gamma + \sum_{i=1}^{m} r_i \cdot \max\{0, F_i(x) - w_i\gamma - F_i^*\}. \tag{8-4}$$

When the merit function of "Equation 8-4" 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. [1], a solution is therefore to set $\psi(x)$ equal to the worst case objective, i.e.,

$$\psi(x) = \underset{i}{\max}\Lambda_i. \tag{8-5}$$

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 8-5" then becomes infinite for arbitrary violations of the constraints.

To overcome this problem while still retaining the features of "Equation 8-5", the merit function is combined with that of "Equation 6-45", giving the following:

$$\psi(x) = \sum_{i=1}^{m} \begin{cases} r_i \cdot \max\{0, F_i(x) - w_i\gamma - F_i^*\} & \text{if } w_i = 0 \\ \underset{i}{\max}\Lambda_i, \ i = 1, ..., m & \text{otherwise.} \end{cases} \tag{8-6}$$

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 6-26.

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 6-44".

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

$$\min_x \max_i \left(\frac{f_i(x) - goal_i}{weight_i} \right) = \min_x \max_i f_i(x),$$

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.

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] Madsen, K. and H. Schjaer-Jacobsen, "Algorithms for Worst Case Tolerance Optimization," *IEEE Trans. of Circuits and Systems*, Vol. CAS-26, Sept. 1979.

[7] 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.

Compare fminimax and fminunc

A minimax problem minimizes the maximum of a set of objective functions. Why not minimize this maximum function, which is a scalar function? The answer is that the maximum is not smooth, and Optimization Toolbox™ solvers such as `fminunc` require smoothness.

For example, define `fun(x)` as three linear objective functions in two variables, and `fun2` as the maximum of these three objectives.

```
a = [1;1];
b = [-1;1];
c = [0;-1];
a0 = 2;
b0 = -3;
c0 = 4;
fun = @(x)[x*a+a0,x*b+b0,x*c+c0];
fun2 = @(x)max(fun(x),[],2);
```

Plot the maximum of the three objectives.

```
[X,Y] = meshgrid(linspace(-5,5));
Z = fun2([X(:),Y(:)]);
Z = reshape(Z,size(X));
surf(X,Y,Z,'LineStyle','none')
view(-118,28)
```
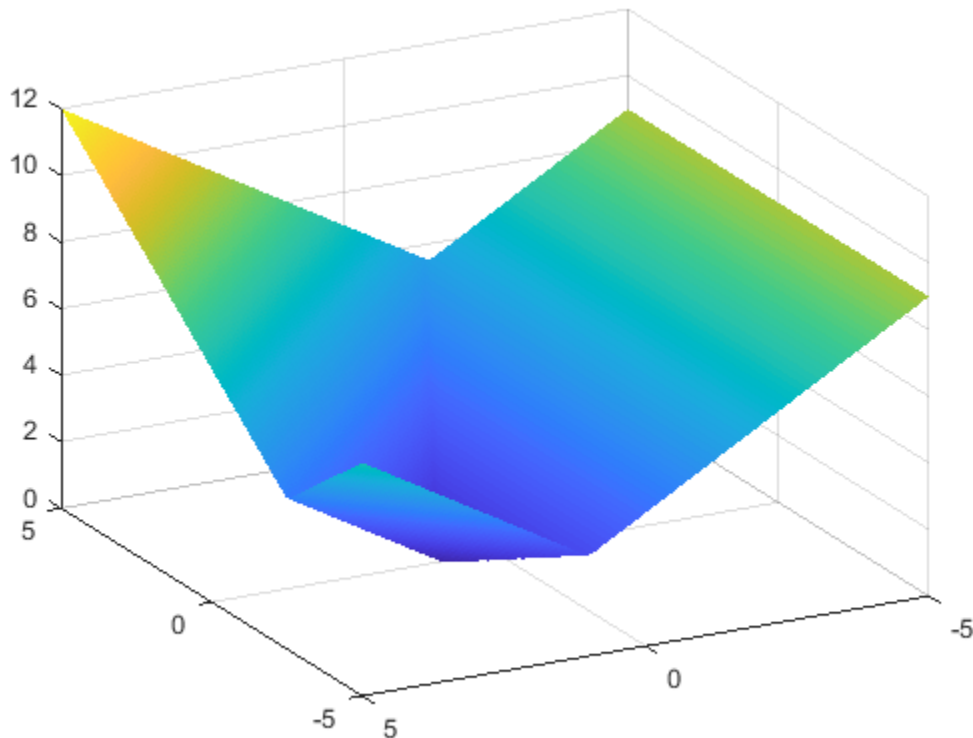

`fminimax` finds the minimax point easily.

```
x0 = [0,0];
[xm,fvalm,maxfval] = fminimax(fun,x0)
```

```
Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

xm = 1×2

   -2.5000    2.2500


fvalm = 1×3

    1.7500    1.7500    1.7500


maxfval = 1.7500
```

However, `fminunc` stops at a point that is far from the minimax point.

```
[xu,fvalu] = fminunc(fun2,x0)
```

```
Local minimum possible.

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

xu = 1×2

        0    1.0000


fvalu = 3.0000
```

`fminimax` finds a better (smaller) solution.

```
fprintf("fminimax finds a point with objective %g,\nwhile fminunc finds a point with objective %g
```

```
fminimax finds a point with objective 1.75,
while fminunc finds a point with objective 3.
```

See Also
`fminimax`

More About
- "Multiobjective Optimization"

Using fminimax with a Simulink® Model

This example shows how to tune the parameters of a Simulink model. The model, `optsim`, ships in the `optim/demos` folder of your MATLAB® installation. The model includes a nonlinear process plant modeled as a Simulink block diagram.

Plant with Actuator Saturation

The plant is an under-damped third-order model with actuator limits. The actuator limits are a saturation limit and a slew rate limit. The actuator saturation limit cuts off input values greater than 2 units or less than –2 units. The slew rate limit of the actuator is 0.8 units/sec. The closed-loop response of the system to a step input is shown in Closed-Loop Response on page 8-0 . 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.
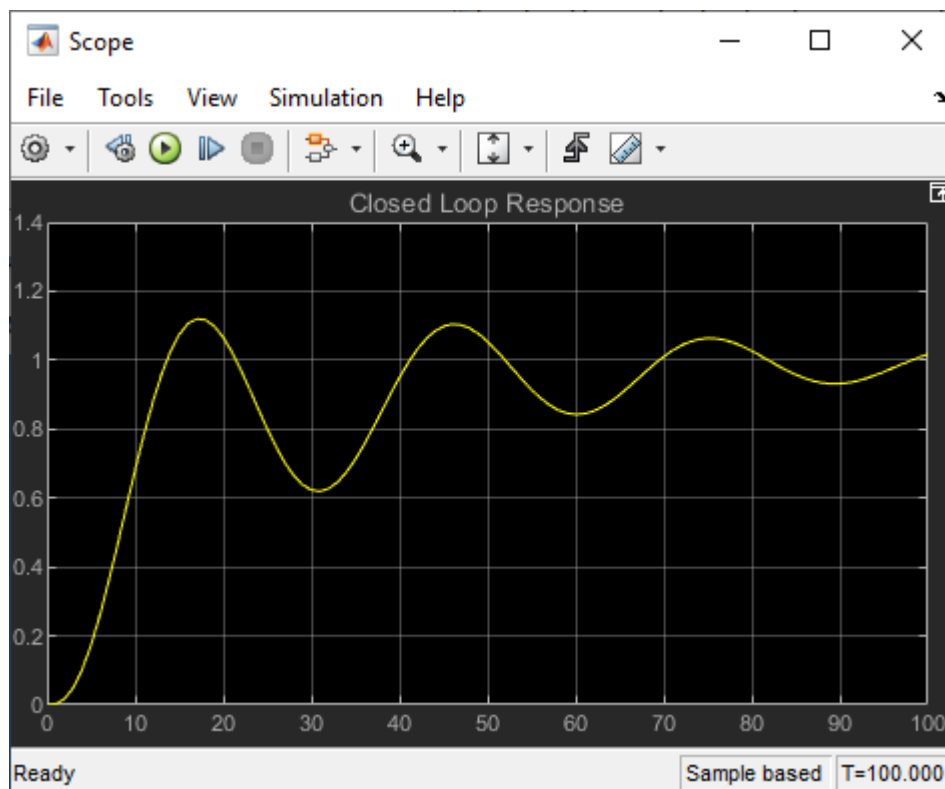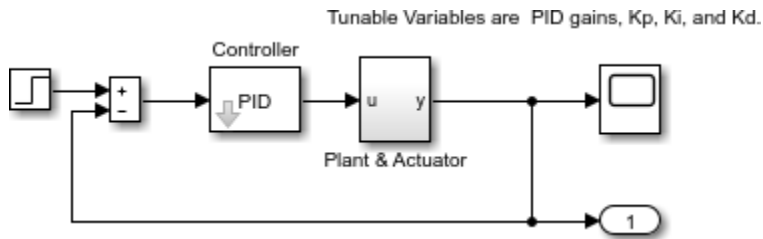
Closed-Loop Model

To optimize this system, minimize the maximum value of the output at any time t between 0 and 100. (In contrast, in "lsqnonlin with a Simulink® Model" on page 12-18 the solution involves minimizing the error between the output and the input signal.)

The code for this example is contained in the function `runtrackmm` at the end of this example. on page 8-0   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 $g \le 0$, the constraint in the function is `g = -yout(20:100) + 0.95`.

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.

Call `runtrackmm`.

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

		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	0.9996	0.00397	1	-0.0195	Hessian modified twice
4	29	0.9996	3.464e-05	1	0.000687	Hessian modified
5	35	0.9996	2.272e-09	1	-0.0175	Hessian modified twice

```
Local minimum possible. Constraints satisfied.

fminimax stopped because the predicted change in the objective function
is less than the value of the function tolerance and constraints
are satisfied to within the value of the constraint tolerance.

Kp = 0.5894

Ki = 0.0605

Kd = 5.5295
```

The last value in the Objective value column of the output shows that the maximum value for all the time steps is just under 1. The closed loop response with this result is shown in the figure Closed-Loop Response Using fminimax on page 8-0 .

This solution differs from the solution obtained in "lsqnonlin with a Simulink® Model" on page 12-18 because you are solving different problem formulations.
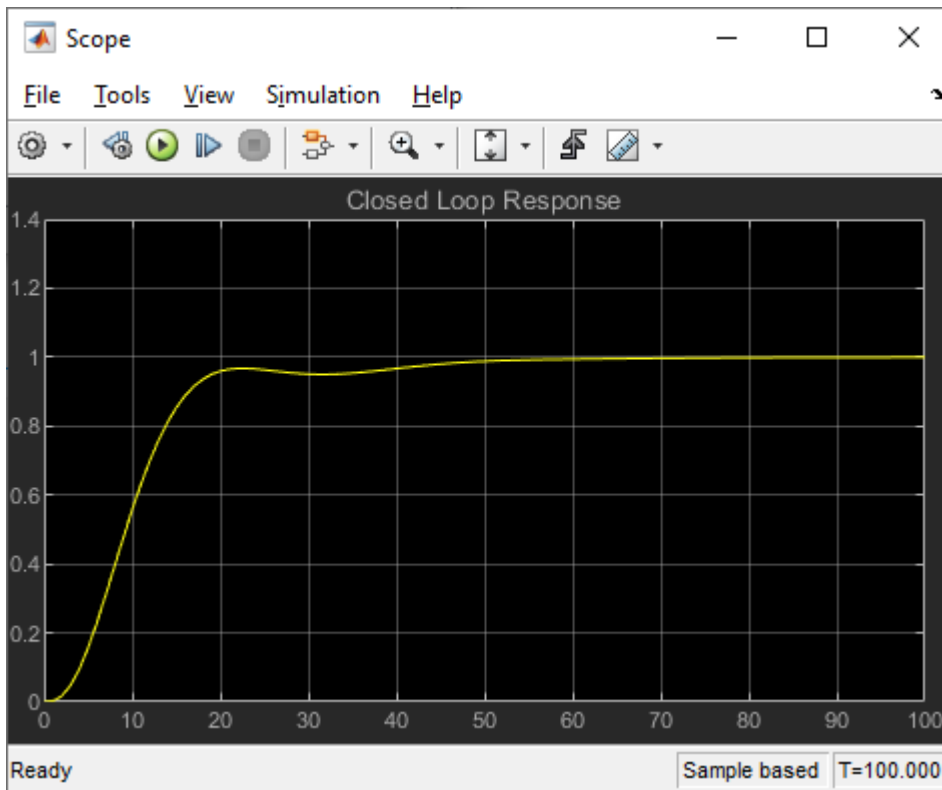
Closed-Loop Response Using `fminimax`

```
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
opt = simset('solver','ode5','SrcWorkspace','Current');
options = optimset('Display','iter',...
    'TolX',0.001,'TolFun',0.001);
pid = fminimax(@trackmmobj,pid0,[],[],[],[],[],[],...
    @trackmmcon,options);
Kp = pid(1); Ki = pid(2); Kd = pid(3);

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

```matlab
        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);

            [~,~,yout] = sim('optsim',[0 100],opt);

            pold = pid;
        end
    end

end
```

See Also

fminimax

More About

- "Multiobjective Optimization"

Signal Processing Using fgoalattain

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

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

$$
\begin{aligned}
H(f) &= \sum_{n=0}^{2M} h(n)e^{-j2\pi fn} \\
&= A(f)e^{-j2\pi fM}, \\
A(f) &= \sum_{n=0}^{M-1} a(n)\cos(2\pi fn),
\end{aligned}
\tag{8-7}
$$

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 8-3). 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','EqualityGoalCount',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','EqualityGoalCount',length(goal));
[a,fval,attainfactor,exitflag]=fgoalattain(@(x)filtmin(x,w0),...
    a0,goal,weight,[],[],[],[],[],[],[],options);

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

Compare the magnitude response computed with the initial coefficients and the final coefficients ("Magnitude Response with Initial and Final Magnitude Coefficients" on page 8-14). Note that you could use the `firpm` function in Signal Processing Toolbox™ software to design this filter.
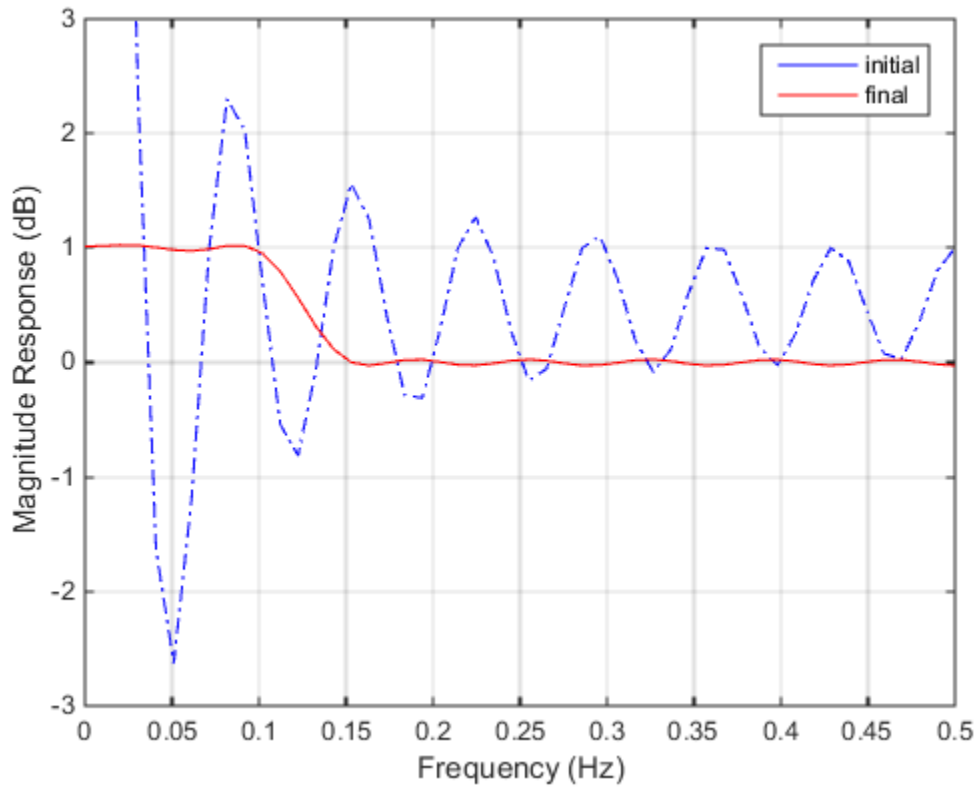
Magnitude Response with Initial and Final Magnitude Coefficients

See Also
`fgoalattain`

More About
- "Multi-Objective Goal Attainment Optimization" on page 8-18
- "Minimax Optimization" on page 8-24

Generate and Plot Pareto Front

This example shows how to generate and plot a Pareto front for a 2-D multiobjective function using `fgoalattain`.

The two objective functions in this example are shifted and scaled versions of the convex function $\sqrt{1 + x^2}$. The code for the objective functions appears in the `simple_mult` helper function at the end of this example on page 8-0   .

Both objective functions decrease in the region $x \leq 0$ and increase in the region $x \geq 1$. In between 0 and 1, $f_1(x)$ increases and $f_2(x)$ decreases, so a tradeoff region exists. Plot the two objective functions for x ranging from $-1/2$ to $3/2$.

```
t = linspace(-1/2,3/2);
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'})
```
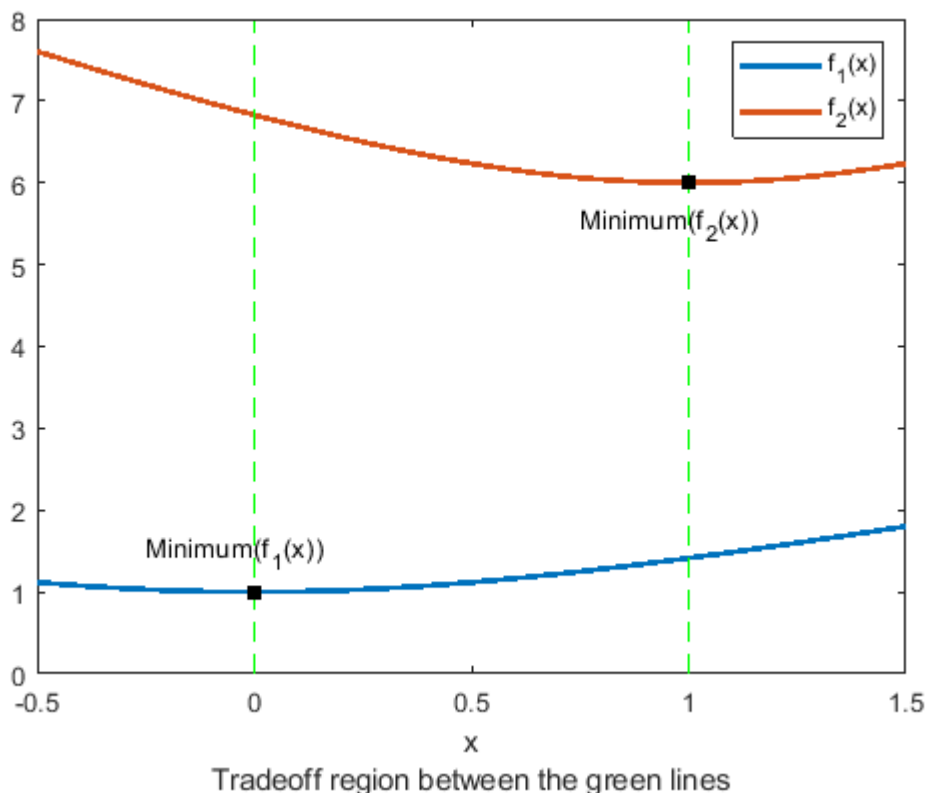
To find the Pareto front, first find the unconstrained minima of the two objective functions. In this case, you can see in the plot that the minimum of $f_1(x)$ is 1, and the minimum of $f_2(x)$ is 6, but in general you might need to use an optimization routine to find the minima.

In general, write a function that returns a particular component of the multiobjective function. (The `pickindex` helper function at the end of this example on page 8-0　returns the kth objective function value.) 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 objective function. You can simultaneously achieve these goals only if the objective 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),'ko');
xlabel('f_1')
ylabel('f_2')
```
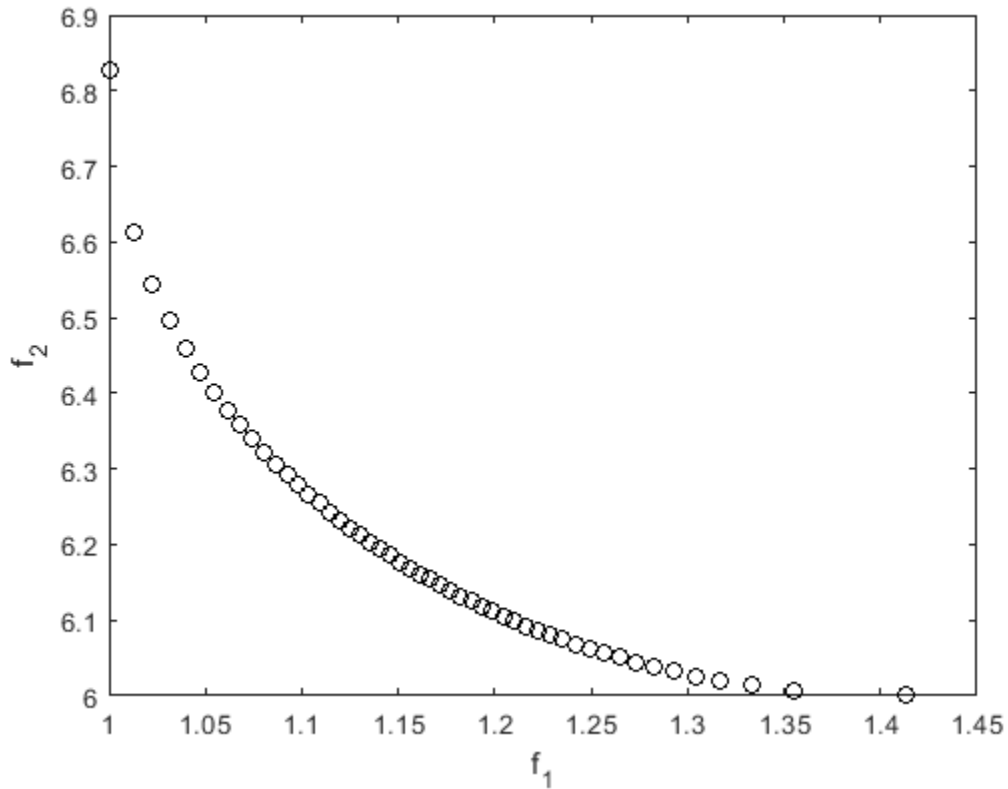
You can see the tradeoff between the two objective functions.

Helper Functions

The following code creates the `simple_multi` function.

```
function f = simple_mult(x)
f(:,1) = sqrt(1+x.^2);
f(:,2) = 4 + 2*sqrt(1+(x-1).^2);
end
```

The following code creates the `pickindex` function.

```
function z = pickindex(x,k)
z = simple_mult(x); % evaluate both objectives
z = z(k); % return objective k
end
```

See Also
`fgoalattain`

More About

Multi-Objective Goal Attainment Optimization

This example shows how to solve a pole-placement problem using the multiobjective goal attainment method. This algorithm is implemented in the function `fgoalattain`.

Equation that Describes Evolution of System

Consider a 2-input 2-output unstable plant. The equation describing the evolution of the system $x(t)$ is

$$\frac{dx}{dt} = Ax(t) + Bu(t),$$

where $u(t)$ is the input (control) signal. The output of the system is

$$y(t) = Cx(t).$$

The matrices A, B, and C are

```
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 ];
```

Optimization Objective

Suppose that the control signal $u(t)$ is set as proportional to the output $y(t)$:

$$u(t) = Ky(t)$$

for some matrix K.

This means that the evolution of the system $x(t)$ is:

$$\frac{dx}{dt} = Ax(t) + BKCx(t) = (A + BKC)x(t).$$

The object of the optimization is to design K to have the following two properties:

1. The real parts of the eigenvalues of $(A + BKC)$ are smaller than [–5, –3, –1]. (This is called pole placement in the control literature.)

2. abs(K) <= 4 (each element of K is between -4 and 4)

In order to solve the optimization, first set the multiobjective goals:

```
goal = [-5, -3, -1];
```

Set the weights equal to the goals to ensure same percentage under- or over-attainment in the goals.

```
weight = abs(goal);
```

Initialize the output feedback controller

```
K0 = [ -1 -1; -1 -1];
```

Set upper and lower bounds on the controller

```
lb = repmat(-4,size(K0))
```

```
lb = 2×2

    -4    -4
    -4    -4
```

```
ub = repmat(4,size(K0))
```

```
ub = 2×2

     4     4
     4     4
```

Set optimization display parameter to give output at each iteration:

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

Create a vector-valued function eigfun that returns the eigenvalues of the closed loop system. This function requires additional parameters (namely, the matrices *A, B,* and *C*); the most convenient way to pass these is through an anonymous function:

```
eigfun = @(K) sort(eig(A+B*K*C));
```

Call Optimization Solver

To begin the optimization we call `fgoalattain`:

```
[K,~,attainfactor] = ...
      fgoalattain(eigfun,K0,goal,weight,[],[],[],[],lb,ub,[],options);
```

Iter	F-count	Attainment factor	Max constraint	Line search steplength	Directional derivative	Procedure
0	6	0	1.88521			
1	13	1.031	0.02998	1	0.745	
2	20	0.3525	0.06863	1	-0.613	
3	27	-0.1706	0.1071	1	-0.223	Hessian modified
4	34	-0.2236	0.06654	1	-0.234	Hessian modified twice
5	41	-0.3568	0.007894	1	-0.0812	
6	48	-0.3645	0.000145	1	-0.164	Hessian modified
7	55	-0.3645	0	1	-0.00515	Hessian modified
8	62	-0.3675	0.0001546	1	-0.00812	Hessian modified twice
9	69	-0.3889	0.008328	1	-0.00751	Hessian modified
10	76	-0.3862	0	1	0.00568	
11	83	-0.3863	2.955e-13	1	-0.998	Hessian modified twice

```
Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.
```

The value of the control parameters at the solution is:

```
K
```

```
K = 2×2

  -4.0000   -0.2564
```

```
  -4.0000   -4.0000
```

The eigenvalues of the closed loop system are in eigfun(K) as follows: (they are also held in output fval)

```
eigfun(K)
```

```
ans = 3×1
```

```
  -6.9313
  -4.1588
  -1.4099
```

The attainment factor indicates the level of goal achievement. A negative attainment factor indicates over-achievement, positive indicates under-achievement. The value attainfactor we obtained in this run indicates that the objectives have been over-achieved by almost 40 percent:

```
attainfactor
```

```
attainfactor = -0.3863
```

Evolution of System Via Solution to ODE

Here is how the system $x(t)$ evolves from time 0 to time 4, using the calculated feedback matrix K, starting from the point x(0) = [1;1;1].

First solve the differential equation:

```
[Times, xvals] = ode45(@(u,x)((A + B*K*C)*x),[0,4],[1;1;1]);
```

Then plot the result:

```
plot(Times,xvals)
legend('x_1(t)','x_2(t)','x_3(t)','Location','best')
xlabel('t');
ylabel('x(t)');
```
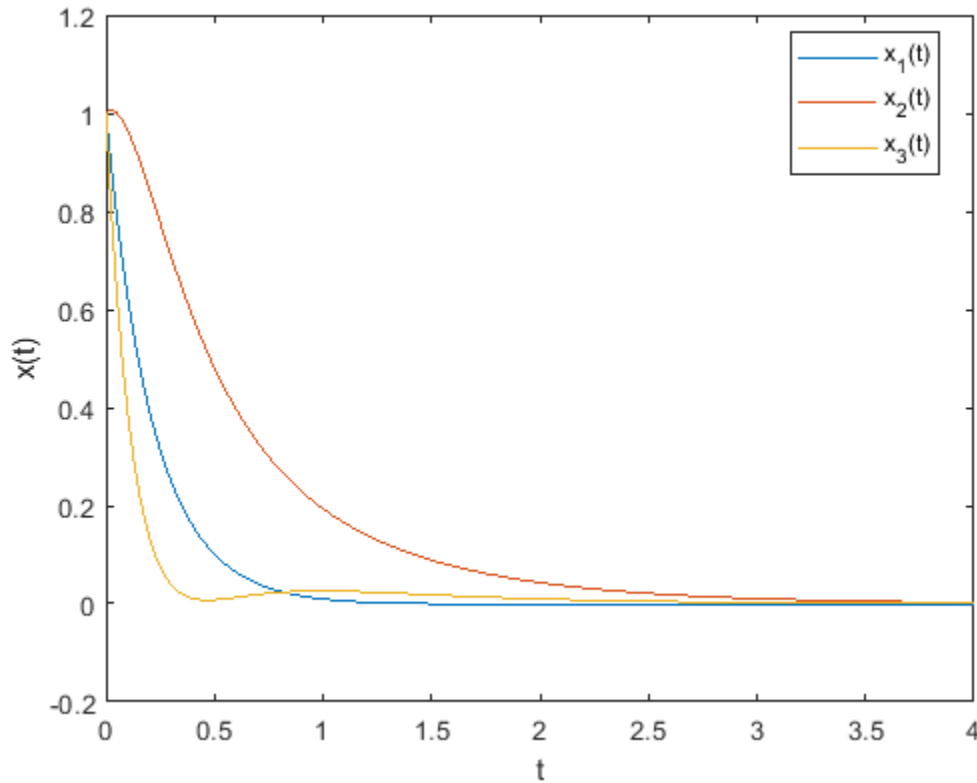
Set Goals To Be Achieved Exactly

Suppose we now require the eigenvalues to be as near as possible to the goal values, [–5, –3, –1]. Set `options.EqualityGoalCount` to the number of objectives that should be as near as possible to the goals (i.e., do not try to over-achieve):

All three objectives should be as near as possible to the goals.

```
options.EqualityGoalCount = 3;
```

Call Optimization Solver

We are ready to call the optimization solver:

```
[K,fval,attainfactor,exitflag,output,lambda] = ...
    fgoalattain(eigfun,K0,goal,weight,[],[],[],[],lb,ub,[],options);
```

		Attainment	Max	Line search	Directional	
Iter	F-count	factor	constraint	steplength	derivative	Procedure
0	6	0	1.88521			
1	13	1.031	0.02998	1	0.745	
2	20	0.3525	0.06863	1	-0.613	
3	27	0.1528	-0.009105	1	-0.22	Hessian modified
4	34	0.02684	0.03722	1	-0.166	Hessian modified
5	41	4.51e-17	0.005703	1	-0.116	Hessian modified
6	48	-1.242e-18	9.626e-06	1	-3.99e-15	Hessian modified
7	55	-6.783e-22	4.925e-11	1	6.55e-14	Hessian modified

```
Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.
```

The value of the control parameters at this solution is:

```
K
```

```
K = 2×2

   -1.5953    1.2040
   -0.4201   -2.9047
```

This time the eigenvalues of the closed loop system, which are also held in output fval, are as follows:

```
eigfun(K)
```

```
ans = 3×1

   -5.0000
   -3.0000
   -1.0000
```

The attainment factor is the level of goal achievement. A negative attainment factor indicates over-achievement, positive indicates under-achievement. The low attainfactor obtained indicates that the eigenvalues have almost exactly met the goals:

```
attainfactor
```

```
attainfactor = -6.7825e-22
```

Evolution of New System Via Solution to ODE

Here is how the system $x(t)$ evolves from time 0 to time 4, using the new calculated feedback matrix K, starting from the point x(0) = [1;1;1].

First solve the differential equation:

```
[Times, xvals] = ode45(@(u,x)((A + B*K*C)*x),[0,4],[1;1;1]);
```

Then plot the result:

```
plot(Times,xvals)
legend('x_1(t)','x_2(t)','x_3(t)','Location','best')
xlabel('t');
ylabel('x(t)');
```
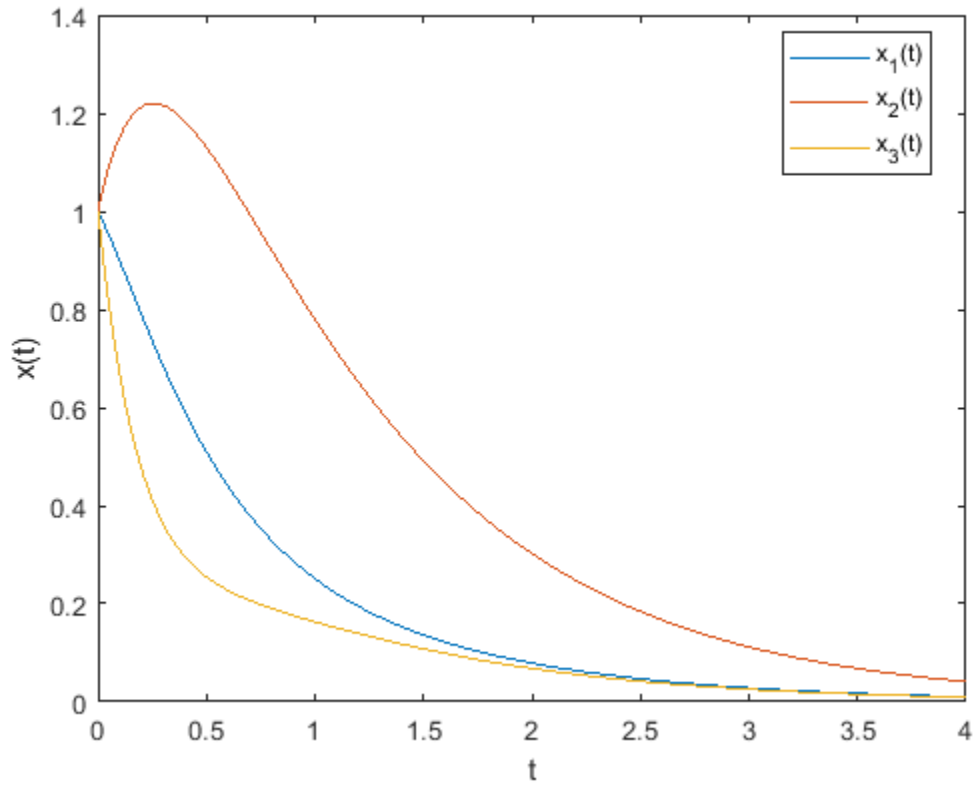
See Also

`fgoalattain`

More About

- "lsqnonlin with a Simulink® Model" on page 12-18

Minimax Optimization

This example shows how to solve a nonlinear filter design problem using a minimax optimization algorithm, `fminimax`, in Optimization Toolbox™. Note that to run this example you must have the Signal Processing Toolbox™ installed.

Set Finite Precision Parameters

Consider an example for the design of finite precision filters. For this, you need to specify not only the filter design parameters such as the cut-off frequency and number of coefficients, but also how many bits are available since the design is in finite precision.

```
nbits  = 8;          % How many bits have we to realize filter
maxbin = 2^nbits-1;  % Maximum number expressable in nbits bits
n      = 4;          % Number of coefficients (order of filter plus 1)
Wn     = 0.2;        % Cutoff frequency for filter
Rp     = 1.5;        % Decibels of ripple in the passband
w      = 128;        % Number of frequency points to take
```

Continuous Design First

This is a continuous filter design; we use `cheby1`, but we could also use `ellip`, `yulewalk` or `remez` here:

```
[b1,a1] = cheby1(n-1,Rp,Wn);

[h,w] = freqz(b1,a1,w);  % Frequency response
h = abs(h);              % Magnitude response
plot(w, h)
title('Frequency response using non-integer variables')
```

Frequency response using non-integer variables
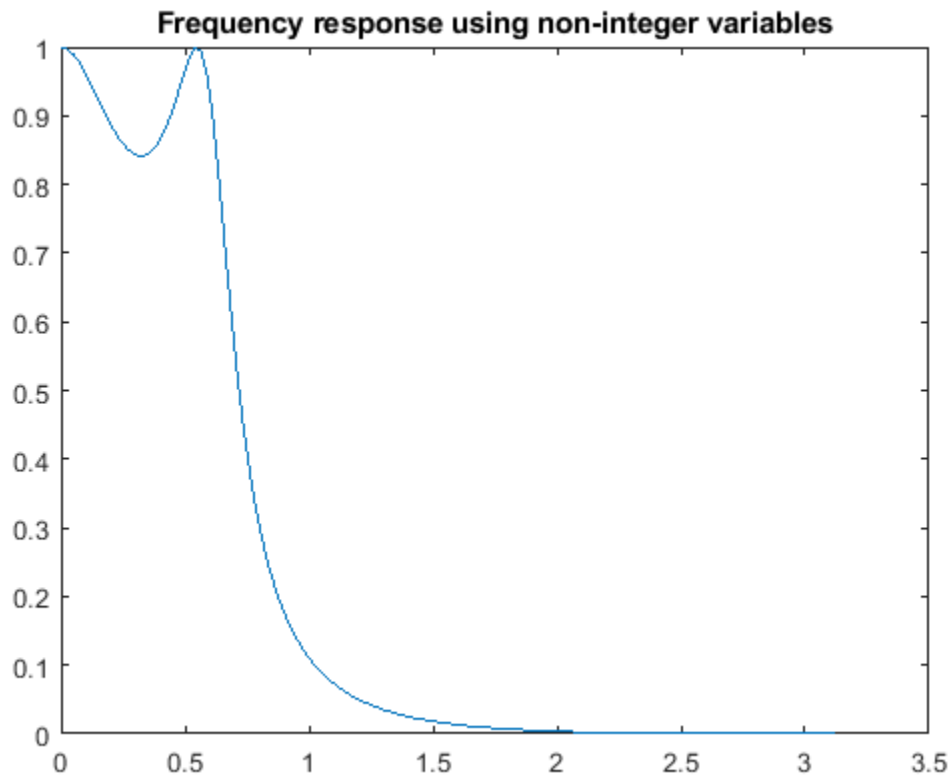
```matlab
x = [b1,a1];              % The design variables
```

Set Bounds for Filter Coefficients

We now set bounds on the maximum and minimum values:

```matlab
if (any(x < 0))
%   If there are negative coefficients - must save room to use a sign bit
%   and therefore reduce maxbin
    maxbin = floor(maxbin/2);
    vlb = -maxbin * ones(1, 2*n)-1;
    vub = maxbin * ones(1, 2*n);
else
%   otherwise, all positive
    vlb = zeros(1,2*n);
    vub = maxbin * ones(1, 2*n);
end
```

Scale Coefficients

Set the biggest value equal to maxbin and scale other filter coefficients appropriately.

```matlab
[m, mix] = max(abs(x));
factor =  maxbin/m;
x =  factor * x;     % Rescale other filter coefficients
xorig = x;

xmask = 1:2*n;
```

```matlab
% Remove the biggest value and the element that controls D.C. Gain
% from the list of values that can be changed.
xmask(mix) = [];
nx = 2*n;
```

Set Optimization Criteria

Using `optimoptions`, adjust the termination criteria to reasonably high values to promote short running times. Also turn on the display of results at each iteration:

```matlab
options = optimoptions('fminimax', ...
    'StepTolerance', 0.1, ...
    'OptimalityTolerance', 1e-4,...
    'ConstraintTolerance', 1e-6, ...
    'Display', 'iter');
```

Minimize the Absolute Maximum Values

We need to minimize absolute maximum values, so we set options.MinAbsMax to the number of frequency points:

```matlab
if length(w) == 1
    options = optimoptions(options,'AbsoluteMaxObjectiveCount',w);
else
    options = optimoptions(options,'AbsoluteMaxObjectiveCount',length(w));
end
```

Eliminate First Value for Optimization

Discretize and eliminate first value and perform optimization by calling FMINIMAX:

```matlab
[x, xmask] = elimone(x, xmask, h, w, n, maxbin)
```

x = *1×8*

```
   0.5441    1.6323    1.6323    0.5441   57.1653 -127.0000  108.0000  -33.8267
```

xmask = *1×6*

```
   1     2     3     4     5     8
```

```matlab
niters = length(xmask);
disp(sprintf('Performing %g stages of optimization.\n\n', niters));
```

```
Performing 6 stages of optimization.
```

```matlab
for m = 1:niters
    fun = @(xfree)filtobj(xfree,x,xmask,n,h,maxbin); % objective
    confun = @(xfree)filtcon(xfree,x,xmask,n,h,maxbin); % nonlinear constraint
    disp(sprintf('Stage: %g \n', m));
    x(xmask) = fminimax(fun,x(xmask),[],[],[],[],vlb(xmask),vub(xmask),...
        confun,options);
    [x, xmask] = elimone(x, xmask, h, w, n, maxbin);
end
```

```
Stage: 1
```

```
                Objective       Max     Line search     Directional
  Iter F-count     value    constraint   steplength      derivative   Procedure
    0     8            0     0.00329174
    1    17    0.0001845      3.34e-07            1         0.0143
```

Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

Stage: 2

```
                Objective       Max     Line search     Directional
  Iter F-count     value    constraint   steplength      derivative   Procedure
    0     7            0     0.0414182
    1    15     0.01649     0.0002558            1          0.261
    2    23     0.01544      6.126e-07           1         -0.0282    Hessian modified
```

Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

Stage: 3

```
                Objective       Max     Line search     Directional
  Iter F-count     value    constraint   steplength      derivative   Procedure
    0     6            0     0.0716961
    1    13     0.05943       2.6e-10            1          0.776
```

Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

Stage: 4

```
                Objective       Max     Line search     Directional
  Iter F-count     value    constraint   steplength      derivative   Procedure
    0     5            0     0.129938
    1    11     0.04278     -8.933e-10           1          0.183
```

Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

Stage: 5

```
                Objective       Max     Line search     Directional
  Iter F-count     value    constraint   steplength      derivative   Procedure
    0     4            0     0.0901749
    1     9     0.03867     -2.263e-10           1          0.256
```

Local minimum possible. Constraints satisfied.

```
fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

Stage: 6

              Objective       Max     Line search     Directional
 Iter F-count     value   constraint   steplength      derivative    Procedure
    0     3            0     0.11283
    1     7      0.05033    9.714e-17             1           0.197

Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.
```

Check Nearest Integer Values

See if nearby values produce a better filter.

```
xold = x;
xmask = 1:2*n;
xmask([n+1, mix]) = [];
x = x + 0.5;
for i = xmask
    [x, xmask] = elimone(x, xmask, h, w, n, maxbin);
end
xmask = 1:2*n;
xmask([n+1, mix]) = [];
x = x - 0.5;
for i = xmask
    [x, xmask] = elimone(x, xmask, h, w, n, maxbin);
end
if any(abs(x) > maxbin)
  x = xold;
end
```

Frequency Response Comparisons

We first plot the frequency response of the filter and we compare it to a filter where the coefficients are just rounded up or down:

```
subplot(211)
bo = x(1:n);
ao = x(n+1:2*n);
h2 = abs(freqz(bo,ao,128));
plot(w,h,w,h2,'o')
title('Optimized filter versus original')

xround = round(xorig)

xround = 1×8

     1     2     2     1    57  -127   108   -34


b = xround(1:n);
a = xround(n+1:2*n);
```

```
h3 = abs(freqz(b,a,128));
subplot(212)
plot(w,h,w,h3,'+')
title('Rounded filter versus original')
```
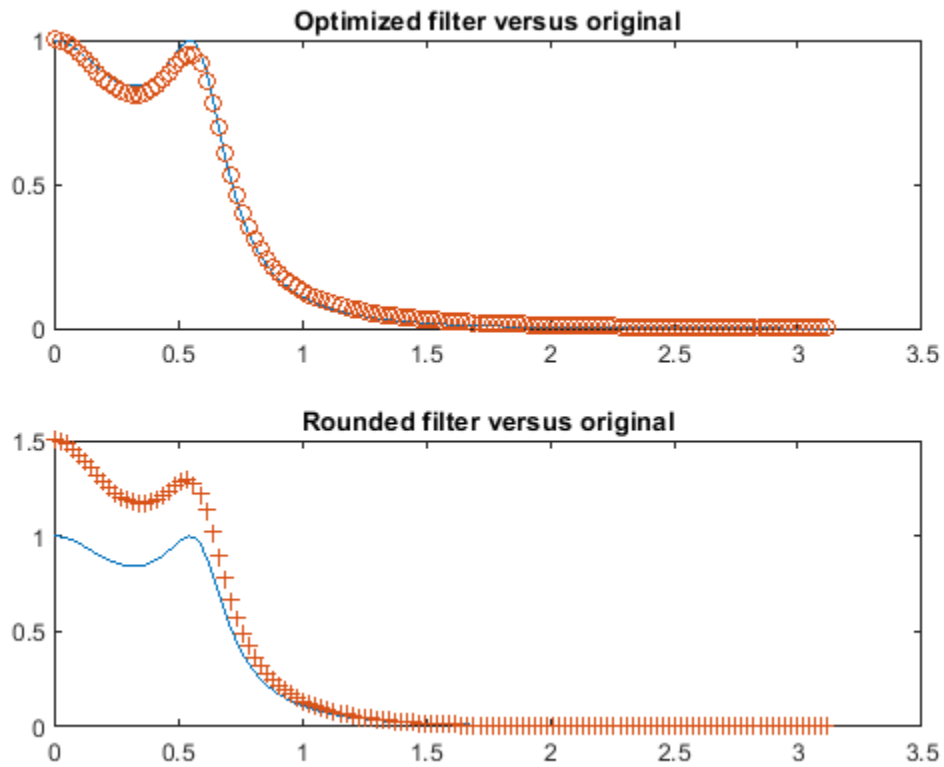


```
fig = gcf;
fig.NextPlot = 'replace';
```

See Also
fminimax

More About
- "lsqnonlin with a Simulink® Model" on page 12-18

Linear Programming and Mixed-Integer Linear Programming

Linear Programming Algorithms

In this section...
"Linear Programming Definition" on page 9-2
"Interior-Point linprog Algorithm" on page 9-2
"Interior-Point-Legacy Linear Programming" on page 9-6
"Dual-Simplex Algorithm" on page 9-9

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:

$A \cdot x \leq b$
$Aeq \cdot x = beq$
$l \leq x \leq u$.

Interior-Point linprog Algorithm

The `linprog 'interior-point'` algorithm is very similar to the "interior-point-convex quadprog Algorithm" on page 11-2. 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 first tries to simplify the problem by removing redundancies and simplifying constraints. The tasks performed during the presolve step can include the following:

• 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 only one variable. If so, check for feasibility, and then change the linear constraint to a bound.

• Check if any linear equality constraint involves only 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 then delete the rows.

• Determine 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 then fix the variables at their appropriate bounds.
- Change any linear inequality constraints to linear equality constraints by adding slack variables.

If the 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, and if the presolve has not produced the solution, the algorithm continues to its next steps. After reaching a stopping criterion, 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 [7], page 397.

Predictor-Corrector

Similar to the fmincon interior-point algorithm on page 6-30, the interior-point-convex algorithm tries to find a point where the Karush-Kuhn-Tucker (KKT) on page 3-12 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 `linprog` algorithm uses a different sign convention for the returned Lagrange multipliers than this discussion gives. This discussion uses the same sign as most literature. See `lambda`.

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

$$\begin{pmatrix} 0 & -\bar{A}^T & 0 & -I & I \\ \bar{A} & 0 & 0 & 0 & 0 \\ -I & 0 & -I & 0 & 0 \\ V & 0 & 0 & X & 0 \\ 0 & 0 & W & 0 & T \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta t \\ \Delta v \\ \Delta w \end{pmatrix} = - \begin{pmatrix} f - \bar{A}^T y - v + w \\ \bar{A} x - \bar{b} \\ u - x - t \\ VX \\ WT \end{pmatrix} = - \begin{pmatrix} r_d \\ r_p \\ r_{ub} \\ r_{vx} \\ r_{wt} \end{pmatrix}, \tag{9-1}$$

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

The iterative display reports these quantities:

Primal infeasibility $= \|r_p\|_1 + \|r_{ub}\|_1$

Dual infeasibility $= \|r_d\|_\infty$.

To solve "Equation 9-1", first convert it to the symmetric matrix form

$$\begin{pmatrix} -D & \bar{A}^T \\ \bar{A} & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = - \begin{pmatrix} R \\ r_p \end{pmatrix}, \tag{9-2}$$

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 9-2" from "Equation 9-1", notice that the second row of "Equation 9-2" is the same as the second matrix row of "Equation 9-1". The first row of "Equation 9-2" comes from solving the last two rows of "Equation 9-1" for Δv and Δw, and then solving for Δt.

"Equation 9-2" 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 9-2" is

$$\bar{A}\Delta x = -r_p$$

and the first set of rows is

$$-D\Delta x + \bar{A}^T \Delta y = -R.$$

Substituting

$$\Delta x = D^{-1}\bar{A}^T \Delta y + D^{-1}R$$

gives

$$\bar{A}D^{-1}\bar{A}^T \Delta y = -\bar{A}D^{-1}R - r_p. \tag{9-3}$$

Usually, the most efficient way to find the Newton step is to solve "Equation 9-3" 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. Afterward, 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 9-2" instead. The `linprog` interior-point algorithm chooses the solution algorithm based on the density of columns.

For more algorithm details, see Mehrotra [6].

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 [4].

The predictor-corrector algorithm is largely the same as the full `quadprog 'interior-point-convex'` version, except for the quadratic terms. See "Full Predictor-Corrector" on page 11-5.

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:

$$\|r_p\|_1 + \|r_{ub}\|_1 \le \rho\text{TolCon}$$
$$\|r_d\|_\infty \le \rho\text{TolFun}$$
$$r_c \le \text{TolFun},$$

where

$$r_c = \max_i(\min(|x_i v_i|, |x_i|, |v_i|), \min(|t_i w_i|, |t_i|, |w_i|)).$$

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 9-8). After preprocessing, the problem has the form

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x = b \\ 0 \le x \le u. \end{cases} \tag{9-4}$$

The upper bounds constraints are implicitly included in the constraint matrix A. With the addition of primal slack variables s, "Equation 9-4" becomes

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x = b \\ x + s = u \\ x \ge 0, \ s \ge 0. \end{cases} \tag{9-5}$$

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

$$\max b^T y - u^T w \text{ such that } \begin{cases} A^T \cdot y - w + z = f \\ z \ge 0, \ w \ge 0, \end{cases} \tag{9-6}$$

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 9-5" and the dual "Equation 9-6", are

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

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

(9-7)

where $x_i z_i$ and $s_i w_i$ denote component-wise multiplication.

The `linprog` algorithm uses a different sign convention for the returned Lagrange multipliers than this discussion gives. This discussion uses the same sign as most literature. See `lambda`.

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 9-7", 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 9-5".)

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

$$\Delta v_c = -\left(F^T(v)\right)^{-1} F\left(v + \Delta v_p\right) - \mu \widehat{e},$$

where $\mu > 0$ is called the *centering* parameter and must be chosen carefully. $\widehat{e}$ is a zero-one vector with the ones corresponding to the quadratic equations in $F(v)$, i.e., the perturbations are only applied to the complementarity conditions, which are all quadratic, but not to the feasibility conditions, which are all linear. The two directions are combined with a step length parameter $\alpha > 0$ and update v to obtain the new iterate v^+:

$$v^+ = v + \alpha\left(\Delta v_p + \Delta v_c\right),$$

where the step length parameter α is chosen so that

$$v^+ = [x^+;y^+;z^+;s^+;w^+]$$

satisfies

$$[x^+;z^+;s^+;w^+] > 0.$$

In solving for the preceding 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:

$$\max\left(\frac{\|r_b\|}{\max(1, \|b\|)}, \frac{\|r_f\|}{\max(1, \|f\|)}, \frac{\|r_u\|}{\max(1, \|u\|)}, \frac{\left|f^T x - b^T y + u^T w\right|}{\max\left(1, \left|f^T x\right|, \left|b^T y - u^T w\right|\right)}\right) \le tol,$$

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 ($\{x\}$ means those x with finite upper bounds), 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 9-7".

The measure of primal infeasibility is $||r_b||$, and the measure of dual infeasibility is $||r_f||$, where the norm is the Euclidean norm.

Preprocessing

The algorithm first tries to simplify the problem by removing redundancies and simplifying constraints. The tasks performed during the presolve step can include the following:

- 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 only one variable. If so, check for feasibility, and then change the linear constraint to a bound.
- Check if any linear equality constraint involves only 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 then delete the rows.
- Determine 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 then fix the variables at their appropriate bounds.
- Change any linear inequality constraints to linear equality constraints by adding slack variables.

If the 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, and if the presolve has not produced the solution, the algorithm continues to its next steps. After reaching a stopping criterion, 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.

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 9-8. For details, see Andersen and Andersen [1] and Nocedal and Wright [7], Chapter 13. This preprocessing reduces the original linear programming problem to the form of "Equation 9-4":

$$\min_{x} f^T x \text{ such that } \begin{cases} A \cdot x = b \\ 0 \le x \le u. \end{cases}$$

A and b are transformed versions of the original constraint matrices. This is the primal problem.

Primal feasibility can be defined in terms of the $^+$ function

$$x^+ = \begin{cases} x \text{ if } x > 0 \\ 0 \text{ if } x \le 0. \end{cases}$$

The measure of primal infeasibility is

$$\text{Primal infeasibility} = \sqrt{\left((\text{lb}-x)^+\right)^2 + \left((x - \text{ub})^+\right)^2 + \left((Ax - b)^+\right)^2 + |\text{Aeq}x - \text{beq}|^2}.$$

As explained in "Equation 9-6", the dual problem is to find vectors y and w, and a slack variable vector z that solve

$$\max b^T y - u^T w \text{ such that } \begin{cases} A^T \cdot y - w + z = f \\ z \ge 0, \ w \ge 0. \end{cases}$$

The `linprog` algorithm uses a different sign convention for the returned Lagrange multipliers than this discussion gives. This discussion uses the same sign as most literature. See `lambda`.

The measure of dual infeasibility is

$$\text{Dual infeasibility} = \left\| A^T y + z - w - f \right\|_2.$$

It is well known (for example, see [7]) 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 [7], 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.

Phase 1 Outline

In phase 1, the algorithm finds an initial basic feasible solution (see "Basic and Nonbasic Variables" on page 9-11 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.

During Phase 2, the solver repeatedly chooses an entering variable and a leaving variable. The algorithm chooses a leaving variable according to a technique suggested by Forrest and Goldfarb [3] called dual steepest-edge pricing. The algorithm chooses an entering variable using the variation of Harris' ratio test suggested by Koberstein [5]. To help alleviate degeneracy, the algorithm can introduce additional perturbations during Phase 2.

Phase 2 Outline

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 9-11 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 [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.

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 $\{a_{i_1}, a_{i_2}, ..., a_{i_m}\}$ is a basis for the column space of A, with index set $B = \{i_1, i_2, ..., i_m\}$, and that $N = \{1, 2, ..., n\}\backslash B$ is the complement of B. The submatrix A_B is called a *basis* and the complementary submatrix A_N is called a *nonbasis*. The vector of *basic variables* is x_B and the vector of *nonbasic variables* is x_N. At each iteration in phase 2, the algorithm replaces one column of the current basis with a column of the nonbasis and updates the variables x_B and x_N accordingly.

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

References

[1] Andersen, E. D., and K. D. Andersen. *Presolving in linear programming*. Math. Programming 71, 1995, pp. 221–245.

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

[3] Forrest, J. J., and D. Goldfarb. *Steepest-edge simplex algorithms for linear programming*. Math. Programming 57, 1992, pp. 341–374.

[4] 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 `https://www.maths.ed.ac.uk/~gondzio/software/correctors.ps`.

[5] 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.

[6] Mehrotra, S. "On the Implementation of a Primal-Dual Interior Point Method." *SIAM Journal on Optimization*, Vol. 2, 1992, pp 575–601.

[7] Nocedal, J., and S. J. Wright. *Numerical Optimization*, Second Edition. Springer Series in Operations Research, Springer-Verlag, 2006.

Typical Linear Programming Problem

This example solves the typical linear programming problem

$$\min_{x} f^{T}x \ \text{ such that} \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ x \geq 0. \end{cases}$$

Load the `sc50b.mat` file, which contains the matrices and vectors A, Aeq, b, beq, f, and the lower bounds lb.

```
load sc50b
```

The problem has 48 variables, 30 inequalities, and 20 equalities.

```
disp(size(A))
```

```
    30    48
```

```
disp(size(Aeq))
```

```
    20    48
```

Set options to use the `dual-simplex` algorithm and the iterative display.

```
options = optimoptions(@linprog,'Algorithm','dual-simplex','Display','iter');
```

The problem has no upper bound, so set `ub` to `[]`.

```
ub = [];
```

Solve the problem by calling `linprog`.

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

```
LP preprocessing removed 2 inequalities, 16 equalities,
16 variables, and 26 non-zero elements.

 Iter       Time              Fval  Primal Infeas    Dual Infeas
    0      0.038    0.000000e+00   0.000000e+00   1.305013e+00
    8      0.094   -1.587073e+02   3.760622e+02   0.000000e+00
   33      0.108   -7.000000e+01   0.000000e+00   0.000000e+00

Optimal solution found.
```

Examine the exit flag, objective function value at the solution, and number of iterations used by `linprog` to solve the problem.

```
exitflag,fval,output.iterations
```

```
exitflag = 1
```

```
fval = -70
```

```
ans = 33
```

You can also find the objective function value and number of iterations in the iterative display.

Maximize Long-Term Investments Using Linear Programming: Solver-Based

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. This example uses the solver-based approach.

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)
```


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:

$$x_2 + x_8 + x_9 = r_1 x_1$$
$$x_3 = r_2 x_2$$
$$x_4 = r_3 x_3$$
$$x_5 + x_7 = r_4 x_4 + r_6 x_6 + r_9 x_9$$

Write these equations in the form $Aeqx = beq$, 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);
```

After 5 years, the return for the initial $1000 is $1262.48

```
plotInvestments(N,PurchaseYears,Maturity,InterestRates,xsol)
```


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:

$$x_1 \leq \text{Pmax} \times \text{Capital}_0$$

$$x_2 \leq \text{Pmax} \times (\rho_1 x_1 + \rho_6 x_6)$$

$$x_3 \leq \text{Pmax} \times (\rho_2 x_2 + \rho_6^2 x_6 + \rho_8 x_8 + \rho_9 x_9)$$

$$x_4 \leq \text{Pmax} \times (\rho_3 x_3 + \rho_6^3 x_6 + \rho_8^2 x_8 + \rho_9^2 x_9)$$

$$x_5 \leq \text{Pmax} \times (\rho_4 x_4 + \rho_6^4 x_4 + \rho_8^3 x_8 + \rho_9^3 x_9)$$

$$x_6 \leq \text{Pmax} \times \text{Capital}_0$$

$$x_7 \leq \text{Pmax} \times (\rho_4 x_4 + \rho_6^4 x_4 + \rho_8^3 x_8 + \rho_9^3 x_9)$$

$$x_8 \leq \text{Pmax} \times (\rho_1 x_1 + \rho_6 x_6)$$

$$x_9 \leq \text{Pmax} \times (\rho_1 x_1 + \rho_6 x_6)$$

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);

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.
```

```
fprintf('After %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,-fval);
```

```
After 5 years, the return for the initial $1000 is $1207.78
```

```
plotInvestments(N,PurchaseYears,Maturity,InterestRates,xsol)
```


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);
```

```matlab
% 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);
```

```
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.
```

```
toc
```

```
Elapsed time is 0.025716 seconds.
```

```
fprintf('\nAfter %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,-fval);
```

```
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);
```

```
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.
```

```
toc
```

Elapsed time is 1.758542 seconds.

```
fprintf('\nAfter %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,-fval);
```

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')
```
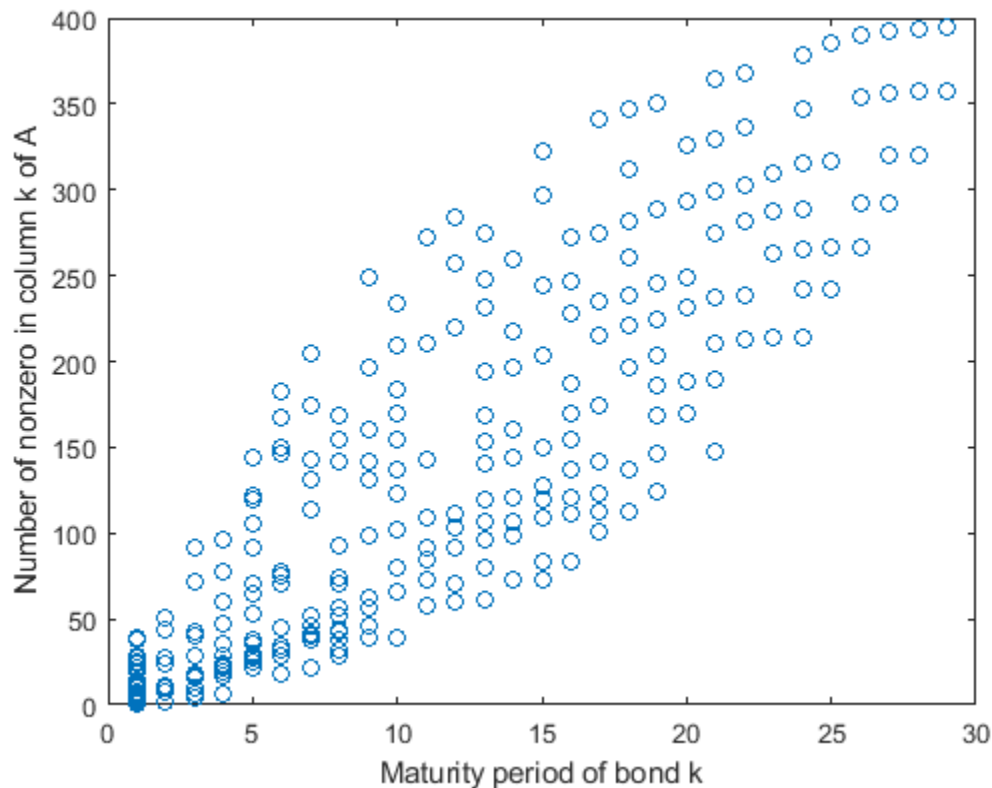

Dense columns in the constraints lead to dense blocks in the solver's internal matrices, yielding a loss of efficiency of its sparse methods. To speed up the solver, try the dual-simplex algorithm, which is less sensitive to column density.

```
% 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);
```

```
Optimal solution found.
```

```
toc
```

```
Elapsed time is 0.349881 seconds.
```

```
fprintf('\nAfter %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,-fval);
```

```
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)
```

```
The amount collected is 88.7137% of the maximum amount $5743.49 that you would obtain from invest
Your final wealth corresponds to a 5.57771% interest rate over the 30 year period.
```

```
plotInvestments(N,PurchaseYears,Maturity,InterestRates,xsol,false)
```

See Also

More About

- *"Maximize Long-Term Investments Using Linear Programming: Problem-Based"* on page 9-26

Maximize Long-Term Investments Using Linear Programming: Problem-Based

This example shows how to use the problem-based approach 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. For the solver-based approach, see "Maximize Long-Term Investments Using Linear Programming: Solver-Based" on page 9-15.

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 a fixed interest rate, compounds the investment 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 at the time of the investment.

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 using the optimization problem approach.

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 beginning of 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 `MaturityYears` represents the end of the year that bond k is available for sale.
- Entry k of vector `InterestRates` represents the percentage 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];
% Bond sale times
MaturityYears = PurchaseYears + Maturity - 1;
% Interest rates in percent
InterestRates = [0;0;0;0;0;2;4;6;6];
% Return after one year of interest
rt = 1 + InterestRates/100;

plotInvestments(N,PurchaseYears,Maturity,InterestRates)
```


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 $\beta_k = 1 + \rho_k/100$ and define r_k as the total return of bond k:

$$r_k = (1 + \rho_k/100)^{m_k} = \beta_k^{m_k}.$$

```
x = optimvar('x',nPtotal,'LowerBound',0);
% Total returns
r = rt.^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:

$$\max_x x_5 r_5 + x_7 r_7 + x_8 r_8$$

Create an optimization problem for maximization, and include the objective function.

```
interestprob = optimproblem('ObjectiveSense','maximize');
interestprob.Objective = x(5)*r(5) + x(7)*r(7) + x(8)*r(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:

$$x_2 + x_8 + x_9 = r_1 x_1$$
$$x_3 = r_2 x_2$$
$$x_4 = r_3 x_3$$
$$x_5 + x_7 = r_4 x_4 + r_6 x_6 + r_9 x_9$$

```
investconstr = optimconstr(T,1);
investconstr(1) = x(1) + x(6) == Capital_0;
investconstr(2) = x(2) + x(8) + x(9) == r(1)*x(1);
investconstr(3) = x(3) == r(2)*x(2);
investconstr(4) = x(4) == r(3)*x(3);
investconstr(5) = x(5) + x(7) == r(4)*x(4) + r(6)*x(6) + r(9)*x(9);
interestprob.Constraints.investconstr = investconstr;
```

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 on the solution vector x. There is no explicit upper bound on the solution vector.

```
x.LowerBound = 0;
```

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');
[sol,fval,exitflag] = solve(interestprob,'options',options)
```

```
Solving problem using linprog.

Solution found during presolve.

sol = struct with fields:
    x: [9x1 double]


fval = 1.2625e+03

exitflag =
    OptimalSolution
```

Visualize the Solution

The exit flag indicates that the solver found an optimal solution. The value `fval`, returned as the second output argument, corresponds to the final wealth. Look at the final sum of investments, and the investment allocation over time.

```
fprintf('After %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,fval);
```

```
After 5 years, the return for the initial $1000 is $1262.48
```

```
plotInvestments(N,PurchaseYears,Maturity,InterestRates,sol.x)
```


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:

$$x_1 \leq \text{Pmax} \times \text{Capital}_0$$

$$x_2 \leq \text{Pmax} \times (\beta_1 x_1 + \beta_6 x_6)$$

$$x_3 \leq \text{Pmax} \times (\beta_2 x_2 + \beta_6^2 x_6 + \beta_8 x_8 + \beta_9 x_9)$$

$$x_4 \leq \text{Pmax} \times (\beta_3 x_3 + \beta_6^3 x_6 + \beta_8^2 x_8 + \beta_9^2 x_9)$$

$$x_5 \leq \text{Pmax} \times (\beta_4 x_4 + \beta_6^4 x_4 + \beta_8^3 x_8 + \beta_9^3 x_9)$$

$$x_6 \leq \text{Pmax} \times \text{Capital}_0$$

$$x_7 \leq \text{Pmax} \times (\beta_4 x_4 + \beta_6^4 x_4 + \beta_8^3 x_8 + \beta_9^3 x_9)$$

$$x_8 \leq \text{Pmax} \times (\beta_1 x_1 + \beta_6 x_6)$$

$$x_9 \leq \text{Pmax} \times (\beta_1 x_1 + \beta_6 x_6)$$

```
% Maximum percentage to invest in any bond
Pmax = 0.6;

constrlimit = optimconstr(nPtotal,1);
constrlimit(1) = x(1) <= Pmax*Capital_0;
constrlimit(2) = x(2) <= Pmax*(rt(1)*x(1) + rt(6)*x(6));
constrlimit(3) = x(3) <= Pmax*(rt(2)*x(2) + rt(6)^2*x(6) + rt(8)*x(8) + rt(9)*x(9));
constrlimit(4) = x(4) <= Pmax*(rt(3)*x(3) + rt(6)^3*x(6) + rt(8)^2*x(8) + rt(9)^2*x(9));
constrlimit(5) = x(5) <= Pmax*(rt(4)*x(4) + rt(6)^4*x(6) + rt(8)^3*x(8) + rt(9)^3*x(9));
constrlimit(6) = x(6) <= Pmax*Capital_0;
constrlimit(7) = x(7) <= Pmax*(rt(4)*x(4) + rt(6)^4*x(6) + rt(8)^3*x(8) + rt(9)^3*x(9));
constrlimit(8) = x(8) <= Pmax*(rt(1)*x(1) + rt(6)*x(6));
constrlimit(9) = x(9) <= Pmax*(rt(1)*x(1) + rt(6)*x(6));

interestprob.Constraints.constrlimit = constrlimit;
```

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.

```
[sol,fval] = solve(interestprob,'options',options);
```

Solving problem using linprog.

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.

```
fprintf('After %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,fval);
```

After 5 years, the return for the initial $1000 is $1207.78

```
plotInvestments(N,PurchaseYears,Maturity,InterestRates,sol.x)
```

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.

```
% 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;
% Return after one year of interest
rt = 1 + InterestRates/100;
% Compute the return at the end of the maturity period for each bond:
r = rt.^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 at the end of the year
MaturityYears = PurchaseYears + Maturity - 1;
```

Compute the times when bonds can be bought or sold. The `buyindex` matrix holds the potential purchase times, and the `sellindex` matrix holds the potential sales times for each bond.

```
buyindex = false(nPtotal,T); % allocate nPtotal-by-T matrix
for ii = 1:T
    buyindex(:,ii) = PurchaseYears == ii;
end
sellindex = false(nPtotal,T);
for ii = 1:T
    sellindex(:,ii) = MaturityYears == ii;
end
```

Set up the optimization variables corresponding to the bonds.

```
x = optimvar('x',nPtotal,1,'LowerBound',0);
```

Create the optimization problem and objective function.

```
interestprob = optimproblem('ObjectiveSense','maximize');
interestprob.Objective = sum(x(sellindex(:,T)).*r(sellindex(:,T)));
```

For convenience, create a temporary array xBuy, whose columns represent the bonds we can buy at each time period.

```
xBuy = repmat(x,1,T).*double(buyindex);
```

Similarly, create a temporary array xSell, whose columns represent the bonds we can sell at each time period.

```
xSell = repmat(x,1,T).*double(sellindex);
```

The return generated for selling these bounds is

```
xReturnFromSell = xSell.*repmat(r,1,T);
```

Create the constraint that the amount you invest in each time period is the amount that you sold in the previous time period.

```
interestprob.Constraints.InitialInvest = sum(xBuy(:,1)) == Capital_0;
interestprob.Constraints.InvestConstraint = sum(xBuy(:,2:T),1) == sum(xReturnFromSell(:,1:T-1),1)
```

Solution with No Holding Limit

Solve the problem.

```
tic
[sol,fval,exitflag] = solve(interestprob,'options',options);
```

```
Solving problem using linprog.

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.
```

```
toc
```

Elapsed time is 0.459383 seconds.

How well did the investments do?

```
fprintf('After %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,fval);
```

After 30 years, the return for the initial $1000 is $5167.58

Solution with Limited Holdings

To create constraints that limit the fraction of investments in each asset, set up a matrix that keeps track of the active bonds at each time. To express the constraint that each investment must be less than `Pmax` times the total value, set up a matrix that keeps track of the value of each investment at each time. For this larger problem, set the maximum fraction that can be held to 0.4.

```
Pmax = 0.4;
```

Create an `active` matrix corresponding to times when a bond can be held, and a `cactive` matrix that holds the cumulative duration of each active bond. So the value of bond `j` at time `t` is `x(j)*(rt^cactive)`.

```
active = double(buyindex | sellindex);
for ii = 1:T
    active(:,ii) = double((ii >= PurchaseYears) & (ii <= MaturityYears));
end
cactive = cumsum(active,2);
cactive = cactive.*active;
```

Create the matrix whose entry (j,p) represents the value of bond j at time period p:

```
bondValue = repmat(x, 1, T).*active.*(rt.^(cactive));
```

Determine the total value of the investments at each time interval so you can impose the constraint on limited holdings. `mvalue` is the money invested in all the bonds at the end of each time period, an `nPtotal`-by-T matrix.`moneyavailable` is the sum over the bonds of the money invested at the beginning of the time period, meaning the value of the portfolio at each time.

```
constrlimit = optimconstr(nPtotal,T);
constrlimit(:,1) = xBuy(:,1) <= Pmax*Capital_0;
constrlimit(:,2:T) = xBuy(:,2:T)  <= repmat(Pmax*sum(bondValue(:,1:T-1),1), nPtotal, 1).*double(I
interestprob.Constraints.constrlimit = constrlimit;
```

Solve the problem with limited holdings.

```
tic
[sol,fval,exitflag] = solve(interestprob,'options',options);
```

Solving problem using linprog.

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.

```
toc
```

```
Elapsed time is 2.040657 seconds.
```

```
fprintf('After %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,fval);
```

```
After 30 years, the return for the initial $1000 is $5095.26
```

To speed up the solver, try the dual-simplex algorithm.

```
options = optimoptions('linprog','Algorithm','dual-simplex');
tic
[sol,fval,exitflag] = solve(interestprob,'options',options);
```

```
Solving problem using linprog.
```

```
Optimal solution found.
```

```
toc
```

```
Elapsed time is 0.903047 seconds.
```

```
fprintf('After %d years, the return for the initial $%g is $%g \n',...
    T,Capital_0,fval);
```

```
After 30 years, the return for the initial $1000 is $5095.26
```

In this case, the dual-simplex algorithm took less time to obtain the same solution.

Qualitative Result Analysis

To get a feel for the solution, 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)
```

```
The amount collected is 88.7137% of the maximum amount $5743.49 that you would obtain from invest
Your final wealth corresponds to a 5.57771% interest rate over the 30 year period.
```

```
plotInvestments(N,PurchaseYears,Maturity,InterestRates,sol.x,false)
```

See Also

More About

- "Maximize Long-Term Investments Using Linear Programming: Solver-Based" on page 9-15
- "Problem-Based Optimization Workflow" on page 10-2

Create Multiperiod Inventory Model in Problem-Based Framework

This example shows how to create a multiperiod inventory model in the problem-based framework. The problem is to schedule production of fertilizer blends over a period of time using a variety of ingredients whose costs depend on time in a predictable way. Assume that you know in advance the demand for the fertilizers. The objective is to maximize profits while meeting demand, where the costs are for purchasing raw ingredients and for storing fertilizer over time. You can determine costs in advance by using futures or other contracts.

Fertilizers and Ingredients

Granular fertilizers have nutrients nitrogen (N), phosphorous (P), and potassium (K). You can blend the following raw materials to obtain fertilizer blends with the requisite nutrients.

```
load fertilizer
blends = blendDemand.Properties.VariableNames % Fertilizers to produce

blends = 1x2 cell
    {'Balanced'}    {'HighN'}


nutrients = rawNutrients.Properties.RowNames

nutrients = 3x1 cell
    {'N'}
    {'P'}
    {'K'}


raws = rawNutrients.Properties.VariableNames % Raw materials

raws = 1x6 cell
    {'MAP'}    {'Potash'}    {'AN'}    {'AS'}    {'TSP'}    {'Sand'}
```

The two fertilizer blends have the same nutrient requirements (10% N, 10% P, and 10% K by weight), except the "HighN" blend has an additional 10% N for a total of 20% N.

```
disp(blendNutrients) % Table is in percentage
```

	Balanced	HighN
N	10	20
P	10	10
K	10	10

The raw materials have the following names and nutrient percentages by weight.

```
disp(rawNutrients) % Table is in percentage
```

	MAP	Potash	AN	AS	TSP	Sand
N	11	0	35	21	0	0
P	48	0	0	0	46	0
K	0	60	0	0	0	0

The raw material `Sand` has no nutrient content. Sand dilutes other ingredients, if necessary, to obtain the requisite percentages of nutrients by weight.

Store the numbers of each of these quantities in variables.

```
nBlends = length(blends);
nRaws = length(raws);
nNutrients = length(nutrients);
```

Forecast Demand and Revenue

Assume that you know in advance the demand in weight (tons) for the two fertilizer blends for the time periods in the problem.

```
disp(blendDemand)
```

	Balanced	HighN
January	750	300
February	800	310
March	900	600
April	850	400
May	700	350
June	700	300
July	700	200
August	600	200
September	600	200
October	550	200
November	550	200
December	550	200

You know the prices per ton at which you sell the fertilizer blends. These prices per ton do not depend on time.

```
disp(blendPrice)
```

Balanced	HighN
400	550

Prices of Raw Materials

Assume that you know in advance the prices in tons for the raw materials. These prices per ton depend on time according to the following table.

```
disp(rawCost)
```

	MAP	Potash	AN	AS	TSP	Sand
January	350	610	300	135	250	80
February	360	630	300	140	275	80
March	350	630	300	135	275	80
April	350	610	300	125	250	80
May	320	600	300	125	250	80
June	320	600	300	125	250	80
July	320	600	300	125	250	80

August	320	600	300	125	240	80
September	320	600	300	125	240	80
October	310	600	300	125	240	80
November	310	600	300	125	240	80
December	340	600	300	125	240	80

Storage Cost

The cost for storing blended fertilizer applies per ton and per time period.

```
disp(inventoryCost)

    10
```

Capacity Constraints

You can store no more than `inventoryCapacity` tons of total fertilizer blends at any time period.

```
disp(inventoryCapacity)

      1000
```

You can produce a total of no more than `productionCapacity` tons in any time period.

```
disp(productionCapacity)

      1200
```

Connection Among Production, Sales, and Inventory

You start the schedule with a certain amount, or inventory, of fertilizer blends available. You have a certain target for this inventory at the final period. At each time period, the amount of fertilizer blend is the amount at the end of the previous time period, plus the amount produced, minus the amount sold. In other words, for times greater than 1:

```
inventory(time,product) = inventory(time-1,product) +
production(time,product) - sales(time,product)
```

This equation implies that the inventory is counted at the end of the time period. The time periods in the problem are as follows.

```
months = blendDemand.Properties.RowNames;
nMonths = length(months);
```

The initial inventory affects the inventory at time 1 as follows.

```
inventory(1,product) = initialInventory(product) + production(1,product) -
sales(1,product)
```

The initial inventory is in the data `blendInventory{'Initial',:}`. The final inventory is in the data `blendInventory{'Final',:}`.

Assume that unmet demand is lost. In other words, if you cannot fill all the orders in a time period, the excess orders do not carry over into the next time period.

Optimization Problem Formulation

The objective function for this problem is profit, which you want to maximize. Therefore, create a maximization problem in the problem-based framework.

```
inventoryProblem = optimproblem('ObjectiveSense','maximize');
```

The variables for the problem are the quantities of fertilizer blends that you make and sell each month, and the raw ingredients that you use to make those blends. The upper bound on `sell` is the demand, `blendDemand`, for each time period and each fertilizer blend.

```
make = optimvar('make',months,blends,'LowerBound',0);
sell = optimvar('sell',months,blends,'LowerBound',0,'UpperBound',blendDemand{months,blends});
use  = optimvar('use',months,raws,blends,'LowerBound',0);
```

Additionally, create a variable that represents the inventory at each time.

```
inventory = optimvar('inventory',months,blends,'LowerBound',0,'UpperBound',inventoryCapacity);
```

To calculate the objective function in terms of the problem variables, calculate the revenue and costs. The revenue is the amount you sell of each fertilizer blend times the price, added over all time periods and blends.

```
revenue = sum(blendPrice{1,:}.*sum(sell(months,blends),1));
```

The cost of ingredients is the cost for each ingredient used at each time, added over all time periods. Because the amount used at each time is separated into the amount used for each blend, also add over the blends.

```
blendsUsed = sum(use(months,raws,blends),3);
ingredientCost = sum(sum(rawCost{months,raws}.*blendsUsed));
```

The storage cost is the cost for storing the inventory over each time period, added over time and blends.

```
storageCost = inventoryCost*sum(inventory(:));
```

Now place the objective function into the `Objective` property of the problem by using dot notation.

```
inventoryProblem.Objective = revenue - ingredientCost - storageCost;
```

Problem Constraints

The problem has several constraints. First, express the inventory equation as a set of constraints on the problem variables.

```
materialBalance = optimconstr(months,blends);
timeAbove1 = months(2:end);
previousTime = months(1:end-1);
materialBalance(timeAbove1,:) = inventory(timeAbove1,:) == inventory(previousTime,:) +...
    make(timeAbove1,:) - sell(timeAbove1,:);
materialBalance(1,:) = inventory(1,:) == blendInventory{'Initial',:} +...
    make(1,:) - sell(1,:);
```

Express the constraint that the final inventory is fixed as well.

```
finalC = inventory(end,:) == blendInventory{'Final',:};
```

The total inventory at each time is bounded.

```
boundedInv = sum(inventory,2) <= inventoryCapacity;
```

You can produce a limited amount in each time period.

```
processLimit = sum(make,2) <= productionCapacity;
```

The amount that you produce each month of each blend is the amount of raw materials that you use. The `squeeze` function converts the sum from a `nmonths`-by-1-by- `nblends` array to a `nmonths`-by-`nblends` array.

```
rawMaterialUse = squeeze(sum(use(months,raws,blends),2)) == make(months,blends);
```

The nutrients in each blend must have the requisite values. In the following inner statement, the multiplication `rawNutrients{n,raws}*use(m,raws,b)'` adds the nutrient values at each time over the raw materials used.

```
blendNutrientsQuality = optimconstr(months,nutrients,blends);
for m = 1:nMonths
    for b = 1:nBlends
        for n = 1:nNutrients
            blendNutrientsQuality(m,n,b) = rawNutrients{n,raws}*use(m,raws,b)' == blendNutrients
        end
    end
end
```

Place the constraints into the problem.

```
inventoryProblem.Constraints.materialBalance = materialBalance;
inventoryProblem.Constraints.finalC = finalC;
inventoryProblem.Constraints.boundedInv = boundedInv;
inventoryProblem.Constraints.processLimit = processLimit;
inventoryProblem.Constraints.rawMaterialUse = rawMaterialUse;
inventoryProblem.Constraints.blendNutrientsQuality = blendNutrientsQuality;
```

Solve Problem

The problem formulation is complete. Solve the problem.

```
[sol,fval,exitflag,output] = solve(inventoryProblem)

Solving problem using linprog.

Optimal solution found.

sol = struct with fields:
    inventory: [12x2 double]
         make: [12x2 double]
         sell: [12x2 double]
          use: [12x6x2 double]


fval = 2.2474e+06

exitflag =
    OptimalSolution


output = struct with fields:
          iterations: 162
       constrviolation: 5.4570e-12
             message: 'Optimal solution found.'
           algorithm: 'dual-simplex'
       firstorderopt: 6.5235e-12
```

```
        solver: 'linprog'
```

Display the results in tabular and graphical form.

```matlab
if exitflag > 0
    fprintf('Profit: %g\n',fval);
    makeT = array2table(sol.make,'RowNames',months,'VariableNames',strcat('make',blends));
    sellT = array2table(sol.sell,'RowNames',months,'VariableNames',strcat('sell',blends));
    storeT = array2table(sol.inventory,'RowNames',months,'VariableNames',strcat('store',blends));
    productionPlanT = [makeT sellT storeT]
    figure
    subplot(3,1,1)
    bar(sol.make)
    legend('Balanced','HighN','Location','eastoutside')
    title('Amount Made')
    subplot(3,1,2)
    bar(sol.sell)
    legend('Balanced','HighN','Location','eastoutside')
    title('Amount Sold')
    subplot(3,1,3)
    bar(sol.inventory)
    legend('Balanced','HighN','Location','eastoutside')
    title('Amount Stored')
    xlabel('Time')
end
```

```
Profit: 2.24739e+06

productionPlanT=12×6 table
               makeBalanced    makeHighN    sellBalanced    sellHighN    storeBalanced    store
               _____    _____    _____    _____    _____    _____

    January        1100           100           750            300            550
    February        600           310           800            310            350
    March           550           650           900            600              0                5
    April           850           350           850            400              0
    May             700           350           700            350              0
    June            700           300           700            300              0
    July            700           200           700            200              0
    August          600           200           600            200              0
    September       600           200           600            200              0
    October         550           200           550            200              0
    November        550           200           550            200              0
    December        750           400           550            200            200               20
```
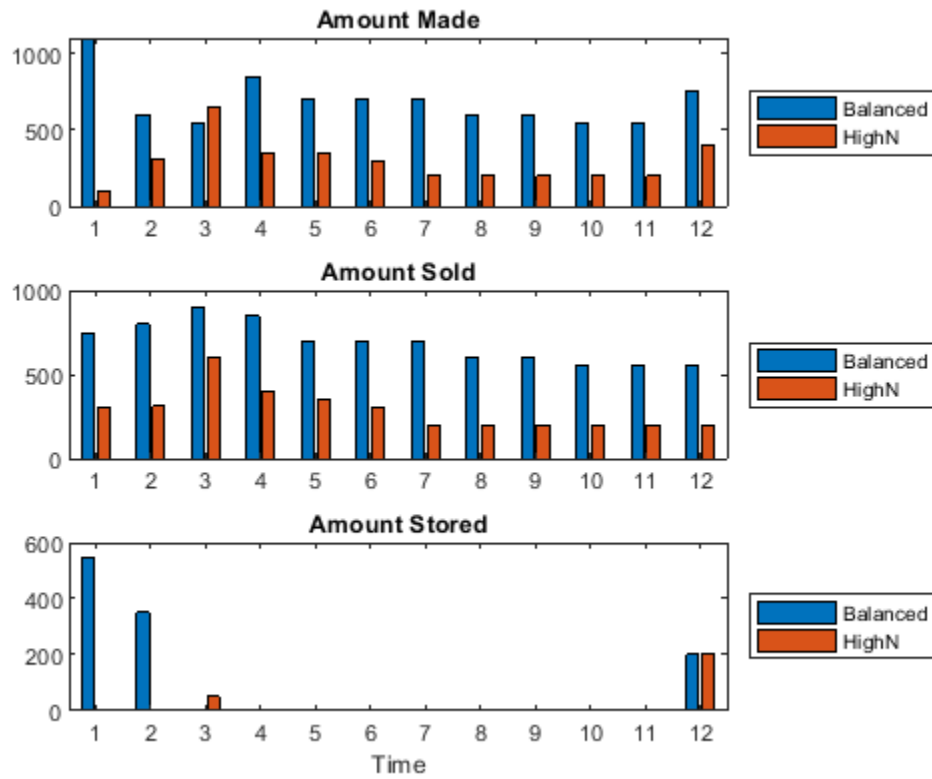
See Also

More About

*   "Problem-Based Optimization Workflow" on page 10-2

Mixed-Integer Linear Programming Algorithms

In this section...
"Mixed-Integer Linear Programming Definition" on page 9-43
"intlinprog Algorithm" on page 9-43

Mixed-Integer Linear Programming Definition

A mixed-integer linear program (MILP) is a problem with

- Linear objective function, f^Tx, 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 "Write Constraints")

- 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 `intcon`, find a vector x to solve

$$\min_{x} f^Tx \text{ subject to } \begin{cases} x(\text{intcon}) \text{ are integers} \\ A \cdot x \le b \\ Aeq \cdot x = beq \\ lb \le x \le ub \,. \end{cases}$$

intlinprog Algorithm

- "Algorithm Overview" on page 9-43
- "Linear Program Preprocessing" on page 9-44
- "Linear Programming" on page 9-44
- "Mixed-Integer Program Preprocessing" on page 9-44
- "Cut Generation" on page 9-45
- "Heuristics for Finding Feasible Solutions" on page 9-46
- "Branch and Bound" on page 9-48

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 9-44.

2 Solve an initial relaxed (noninteger) problem using "Linear Programming" on page 9-44.

3 Perform "Mixed-Integer Program Preprocessing" on page 9-44 to tighten the LP relaxation of the mixed-integer problem.

4 Try "Cut Generation" on page 9-45 to further tighten the LP relaxation of the mixed-integer problem.

5 Try to find integer-feasible solutions using heuristics on page 9-46.

6 Use a "Branch and Bound" on page 9-48 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 9-43, there are matrices A and Aeq and corresponding vectors b and beq that encode a set of linear inequalities and linear equalities

$$A \cdot x \leq b$$
$$Aeq \cdot x = beq.$$

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 [2] and Mészáros and Suhl [8].

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 9-43, 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,

$$f^T x_{LP} \leq f^T x,$$

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 `IntegerPreprocess` option lets you choose whether `intlinprog` takes several steps, takes all of them, or takes almost none of them. If you include an `x0` argument, `intlinprog` uses that value in preprocessing.

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 [10].

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 solutions 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
- Clique cuts
- Cover cuts
- Flow cover cuts

Furthermore, if the problem is purely integer (all variables are integer-valued), then `intlinprog` also uses the following cuts:

- Strong Chvatal-Gomory cuts
- Zero-half 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

For purely integer problems, `'intermediate'` uses the most cut types, because it uses reduce-and-split cuts, while `'advanced'` does not.

Another option, `CutMaxIterations`, 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 [5] and, for clique cuts, Atamtürk, Nemhauser, and Savelsbergh [3].

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. Certain techniques find feasible points faster before or during branch-and-bound. `intlinprog` uses these techniques at the root node and during some branch-and-bound iterations. These techniques are heuristic, meaning they are algorithms that can succeed but can also fail.

Heuristics can be start heuristics, which help the solver find an initial or new integer-feasible solution. Or heuristics can be improvement heuristics, which start at an integer-feasible point and attempt to find a better integer-feasible point, meaning one with lower objective function value. The `intlinprog` improvement heuristics are `'rins'`, `'rss'`, 1-opt, 2-opt, and guided diving.

Set the `intlinprog` heuristics using the `'Heuristics'` option. The options are:

Option	Description
`'basic'` (default)	The solver runs rounding heuristics twice with different parameters, runs diving heuristics twice with different parameters, then runs `'rss'`. The solver does not run later heuristics when earlier heuristics lead to a sufficiently good integer-feasible solution.
`'intermediate'`	The solver runs rounding heuristics twice with different parameters, then runs diving heuristics twice with different parameters. If there is an integer-feasible solution, the solver then runs `'rins'` followed by `'rss'`. If `'rss'` finds a new solution, the solver runs `'rins'` again. The solver does not run later heuristics when earlier heuristics lead to a sufficiently good integer-feasible solution.
`'advanced'`	The solver runs rounding heuristics twice with different parameters, then runs diving heuristics twice with different parameters. If there is an integer-feasible solution, the solver then runs `'rins'` followed by `'rss'`. If `'rss'` finds a new solution, the solver runs `'rins'` again. The solver does not run later heuristics when earlier heuristics lead to a sufficiently good integer-feasible solution.
`'rins'` or the equivalent `'rins-diving'`	`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 [6]. When you select `'rins'`, the solver runs rounding heuristics twice with different parameters, runs diving heuristics twice with different parameters, then runs `'rins'`.
`'rss'` or the equivalent `'rss-diving'`	`intlinprog` applies a hybrid procedure combining ideas from `'rins'` and local branching to search for integer-feasible solutions. When you select `'rss'`, the solver runs rounding heuristics twice with different parameters, runs diving heuristics twice with different parameters, then runs `'rss'`. The solver does not run later heuristics when earlier heuristics lead to a sufficiently good integer-feasible solution. These settings perform the same heuristics as `'basic'`.

Option	Description
'round'	intlinprog takes the LP solution to the relaxed problem at a node, and rounds the integer components in a way that attempts to maintain feasibility. When you select 'round', the solver, at the root node, runs rounding heuristics twice with different parameters, then runs diving heuristics twice with different parameters. Thereafter, the solver runs only rounding heuristics at some branch-and-bound nodes.
'round-diving'	The solver works in a similar way to 'round', but also runs diving heuristics (in addition to rounding heuristics) at some branch-and-bound nodes, not just the root node.
'diving'	intlinprog uses heuristics that are similar to branch-and-bound steps, but follow just one branch of the tree down, without creating the other branches. This single branch leads to a fast "dive" down the tree fragment, thus the name "diving." Currently, intlinprog uses six diving heuristics in this order: • Vector length diving • Coefficient diving • Fractional diving • Pseudo cost diving • Line search diving • Guided diving (applies when the solver already found at least one integer-feasible point) Diving heuristics generally select one variable that should be integer-valued, for which the current solution is fractional. The heuristics then introduce a bound that forces the variable to be integer-valued, and solve the associated relaxed LP again. The method of choosing the variable to bound is the main difference between the diving heuristics. See Berthold [4], Section 3.1.
'none'	intlinprog does not search for a feasible point. The solver simply takes any feasible point it encounters in its branch-and-bound search.

The main difference between 'intermediate' and 'advanced' is that 'advanced' runs heuristics more frequently during branch-and-bound iterations.

In addition to the previous table, the following heuristics run when the Heuristics option is 'basic', 'intermediate', or 'advanced'.

• ZI round — This heuristic runs whenever an algorithm solves a relaxed LP. The heuristic goes through each fractional integer variable to attempt to shift it to a neighboring integer without affecting the feasibility with respect to other constraints. For details, see Hendel [7].

• 1-opt — This heuristic runs whenever an algorithm finds a new integer-feasible solution. The heuristic goes through each integer variable to attempt to shift it to a neighboring integer without affecting the feasibility with respect to other constraints, while lowering the objective function value.

• 2-opt — This heuristic runs whenever an algorithm finds a new integer-feasible solution. 2-opt finds all pairs of integer variables that affect the same constraint, meaning they have nonzero entries in the same row of an A or Aeq constraint matrix. For each pair, 2-opt takes an integer-

feasible solution and moves the values of the variable pairs up or down using all four possible moves (up-up, up-down, down-up, and down-down), looking for a feasible neighboring solution that has a better objective function value. The algorithm tests each integer variable pair by calculating the largest size (same magnitude) of shifts for each variable in the pair that satisfies the constraints and also improves the objective function value.

At the beginning of the heuristics phase, `intlinprog` runs the trivial heuristic unless `Heuristics` is `'none'` or you provide an initial integer-feasible point in the `x0` argument. The trivial heuristic checks the following points for feasibility:

- All zeros
- Upper bound
- Lower bound (if nonzero)
- "Lock" point

The "lock" point is defined only for problems with finite upper and lower bounds for all variables. The "lock" point for each variable is its upper or lower bound, chosen as follows. For each variable j, count the number of corresponding positive entries in the linear constraint matrix `A(:,j)` and subtract the number corresponding negative entries. If the result is positive, use the lower bound for that variable, `lb(j)`. Otherwise, use the upper bound for that variable, `ub(j)`. The "lock" point attempts to satisfy the largest number of linear inequality constraints for each variable, but is not necessarily feasible.

After each heuristic completes with a feasible solution, `intlinprog` calls output functions and plot functions. See "intlinprog Output Function and Plot Function Syntax" on page 15-35.

If you include an `x0` argument, `intlinprog` uses that value in the `'rins'` and guided diving heuristics until it finds a better integer-feasible point. So when you provide `x0`, you can obtain good results by setting the `'Heuristics'` option to `'rins-diving'` or another setting that uses `'rins'`.

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^Tx. 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 9-46.

As explained in "Linear Programming" on page 9-44, 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

$$f^Tx_{feas} \geq f^Tx,$$

because f^Tx 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^Tx_{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 `BranchRule` option:

- `'maxpscost'` — Choose the fractional variable with maximal pseudocost.

    Pseudocost

    The pseudocost 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 $x(i) \leq \lfloor x(i) \rfloor$, and let the change in objective function be denoted

    $\Delta_i^- = f^Tx_i^- - f^Tx$.

    Similarly, Δ_i^+ is the change in objective function when the problem is restricted to have $x(i) \geq \lceil x(i) \rceil$.

    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.

- `'strongpscost'` — Similar to `'maxpscost'`, but instead of the pseudocost being initialized to `1` for each variable, the solver attempts to branch on a variable only after the pseudocost has a more reliable estimate. To obtain a more reliable estimate, the solver does the following (see Achterberg, Koch, and Martin [1]).

    - Order all potential branching variables (those that are currently fractional but should be integer) by their current pseudocost-based scores.
    - Run the two relaxed linear programs based on the current branching variable, starting from the variable with the highest score (if the variable has not yet been used for a branching

calculation). The solver uses these two solutions to update the pseudocosts for the current branching variable. The solver can halt this process early to save time in choosing the branch.

- Continue choosing variables in the list until the current highest pseudocost-based score does not change for k consecutive variables, where k is an internally chosen value, usually between 5 and 10.

- Branch on the variable with the highest pseudocost-based score. The solver might have already computed the relaxed linear programs based on this variable during an earlier pseudocost estimation procedure.

Because of the extra linear program solutions, each iteration of `'strongpscost'` branching takes longer than the default `'maxpscost'`. However, the number of branch-and-bound iterations typically decreases, so the `'strongpscost'` method can save time overall.

- `'reliability'` — Similar to `'strongpscost'`, but instead of running the relaxed linear programs only for uninitialized pseudocost branches, `'reliability'` runs the programs up to k2 times for each variable, where k2 is a small integer such as 4 or 8. Therefore, `'reliability'` has even slower branching, but potentially fewer branch-and-bound iterations, compared to `'strongpscost'`.

- `'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 $f^T x_B = 0$.

`intlinprog` skips the analysis of some subproblems by considering information from the original problem such as the objective function's greatest common divisor (GCD).

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 `AbsoluteGapTolerance` or `RelativeGapTolerance` tolerances.
- The number of explored nodes exceeds the `MaxNodes` option.
- The number of integer feasible points exceeds the `MaxFeasiblePoints` option.

For details about the branch-and-bound procedure, see Nemhauser and Wolsey [9] and Wolsey [11].

References

[1] Achterberg, T., T. Koch and A. Martin. *Branching rules revisited.* Operations Research Letters 33, 2005, pp. 42–54. Available at `https://www-m9.ma.tum.de/downloads/felix-klein/20B/AchterbergKochMartin-BranchingRulesRevisited.pdf`.

[2] Andersen, E. D., and Andersen, K. D. *Presolving in linear programming.* Mathematical Programming 71, pp. 221–245, 1995.

[3] Atamtürk, A., G. L. Nemhauser, M. W. P. Savelsbergh. *Conflict graphs in solving integer programming problems.* European Journal of Operational Research 121, 2000, pp. 40–55.

[4] Berthold, T. *Primal Heuristics for Mixed Integer Programs.* Technischen Universität Berlin, September 2006. Available at `https://www.zib.de/groetschel/students/Diplom-Berthold.pdf`.

[5] Cornuéjols, G. *Valid inequalities for mixed integer linear programs.* Mathematical Programming B, Vol. 112, pp. 3–44, 2008.

[6] 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.

[7] Hendel, G. *New Rounding and Propagation Heuristics for Mixed Integer Programming.* Bachelor's thesis at Technische Universität Berlin, 2011. PDF available at https://opus4.kobv.de/opus4-zib/files/1332/bachelor_thesis_main.pdf.

[8] Mészáros C., and Suhl, U. H. *Advanced preprocessing techniques for linear and quadratic programming.* OR Spectrum, 25(4), pp. 575–595, 2003.

[9] Nemhauser, G. L. and Wolsey, L. A. *Integer and Combinatorial Optimization.* Wiley-Interscience, New York, 1999.

[10] Savelsbergh, M. W. P. *Preprocessing and Probing Techniques for Mixed Integer Programming Problems.* ORSA J. Computing, Vol. 6, No. 4, pp. 445–454, 1994.

[11] 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 9-52
"Some "Integer" Solutions Are Not Integers" on page 9-53
"Large Components Not Integer Valued" on page 9-53
"Large Coefficients Disallowed" on page 9-53

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 `CutMaxIterations` 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 `IntegerPreprocess` 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 `'intermediate'` or `'advanced'`.

7 To try to find more or better feasible points, change the `BranchRule` option to `'strongpscost'` or, if that choice fails to improve the solution, `'maxpscost'`.

8 For a faster solution, increase the `ObjectiveImprovementThreshold` option from its default of zero to a positive value such as `1e-4`. However, this change can cause `intlinprog` to find fewer integer feasible points or a less accurate solution.

9 To attempt to stop the solver more quickly, change the `RelativeGapTolerance` option to a higher value than the default `1e-4`. Similarly, to attempt to obtain a more accurate answer,

change the `RelativeGapTolerance` option to a lower value. These changes do not always improve results.

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 `IntegerTolerance` 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: Solver-Based

This example shows how to solve a mixed-integer linear problem. Although not complex, the example shows the typical steps in formulating a problem using the syntax for `intlinprog`.

For the problem-based approach to this problem, see "Mixed-Integer Linear Programming Basics: Problem-Based" on page 10-47.

Problem Description

You want to blend 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 https://doi.org/10.1287/inte.7.2.39.

Four ingots of steel are available for purchase. Only one of each ingot is available.

Ingot	Weight in Tons	% Carbon	% Molybdenum	$\dfrac{\text{Cost}}{\text{Ton}}$
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 and one grade of scrap steel are available for purchase. Alloy and scrap steels can be purchased in fractional amounts.

Alloy	% Carbon	% Molybdenum	$\dfrac{\text{Cost}}{\text{Ton}}$
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 whether you purchase ingots **2** through **4**.

Variables `x(5)` through `x(7)` are the quantities in tons of alloys **1**, **2**, and **3** that you purchase, and `x(8)` is the quantity of scrap steel that you purchase.

MATLAB® Formulation

Formulate the problem by specifying the inputs for `intlinprog`. The relevant `intlinprog` syntax is:

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

Create the inputs for `intlinprog` from the first (`f`) through the 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`.

The problem has no linear inequality constraints, so `A` and `b` are empty matrices (`[]`).

```
A = [];
b = [];
```

The problem has 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
```

Specify the constraints, which are Aeq*x = beq in matrix form.

```
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 Problem

Now that you have all the inputs, call the solver.

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

LP:               Optimal objective value is 8125.600000.

Cut Generation:   Applied 3 mir cuts.
                  Lower bound is 8495.000000.
                  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.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).
```

View the solution.

`x,fval`

```
x = 8×1

    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 realistic, because you cannot purchase a fraction of an ingot.

See Also

More About

- "Mixed-Integer Linear Programming Basics: Problem-Based" on page 10-47
- "Solver-Based Optimization Problem Setup"

Factory, Warehouse, Sales Allocation Model: Solver-Based

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. For the problem-based approach, see "Factory, Warehouse, Sales Allocation Model: Problem-Based" on page 9-108.

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 $f + w + s \le 1$. 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

$$\sum_w x(p, f, w) \le pcap(f, p) \text{ (capacity of factory).}$$

$$\sum_f x(p, f, w) = \sum_s (d(s, p) \cdot y(s, w)) \text{ (demand is met).}$$

$$\sum_p \sum_s \frac{d(s, p)}{turn(p)} \cdot y(s, w) \le wcap(w) \text{ (capacity of warehouse).}$$

$$\sum_w y(s, w) = 1 \text{ (each sales outlet associates to one warehouse).}$$

$$x(p, f, w) \ge 0 \text{ (nonnegative production).}$$

$$y(s, w) \in \{0, 1\} \text{ (binary } y\text{).}$$

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(1) % 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');
lgnd = legend('Factory','Warehouse','Sales outlet','Location','EastOutside');
lgnd.AutoUpdate = 'off';
xlim([0 N+1]);ylim([0 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,1) + 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)`.

```
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`.

```
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.

```
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.

```
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.

```
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.

```
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','PlotFcn',@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);
```
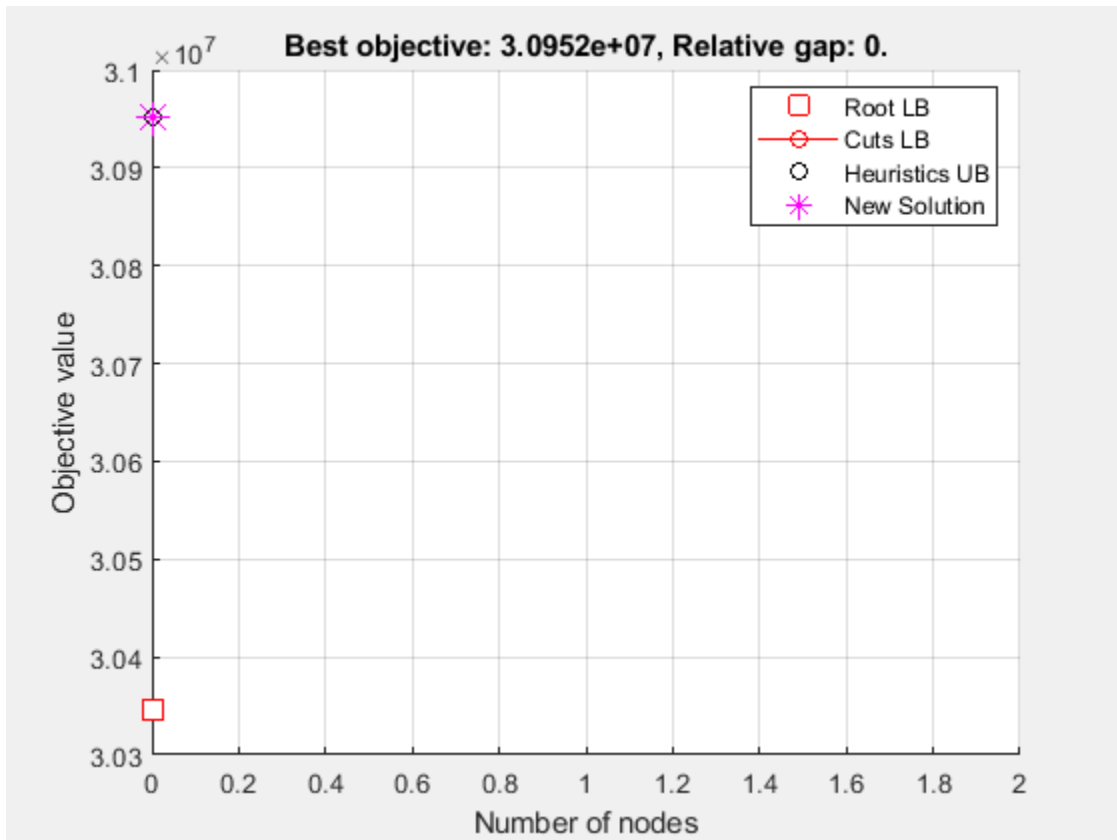


```
if isempty(solution) % If the problem is infeasible or you stopped early with no solution
    disp('intlinprog did not return a solution.')
    return % Stop the script because there is nothing to examine
end
```

Examine the Solution

The solution is feasible, to within the given tolerances.

```
exitflag
```

```
exitflag = 1
```

```
infeas1 = max(Aineq*solution - bineq)
```

```
infeas1 = 8.2991e-12
```

```
infeas2 = norm(Aeq*solution - beq,Inf)
```

```
infeas2 = 1.6428e-11
```

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 "Some "Integer" Solutions Are Not Integers" on page 9-53.

```
diffint = norm(solution(intcon) - round(solution(intcon)),Inf)
```

diffint = 1.1990e-13

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)
```

infeas1 = 8.2991e-12

```
infeas2 = norm(Aeq*solution - beq,Inf)
```

infeas2 = 5.8435e-11

```
diffrounding = norm(fval - obj(:)'*solution,Inf)
```

diffrounding = 1.8626e-08

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 = *1×20*

3	0	3	2	2	2	3	2	3	1	1	0	0	3	4	3

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);
    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')
```


The black * with no green lines represent the unused warehouses.

See Also

More About

- "Factory, Warehouse, Sales Allocation Model: Problem-Based" on page 9-108

Traveling Salesman Problem: Solver-Based

This example shows how to use binary integer programming to solve the classic traveling 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.

For the problem-based approach, see "Traveling Salesman Problem: Problem-Based" on page 9-116.

Problem Formulation

Formulate the traveling 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.

Generate Stops

Generate random stops inside a crude polygonal representation of the continental U.S.

```
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,x,y) % Test if inside the border
        stopsLon(n) = xp;
        stopsLat(n) = yp;
        n = n+1;
    end
end
```

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.

Create Graph and Draw Map

Represent the problem as a graph. Create a graph where the stops are nodes and the trips are edges.

```
G = graph(idxs(:,1),idxs(:,2));
```

Display the stops using a graph plot. Plot the nodes without the graph edges.

```
figure
hGraph = plot(G,'XData',stopsLon,'YData',stopsLat,'LineStyle','none','NodeLabel',{});
hold on
% Draw the outside border
plot(x,y,'r-')
hold off
```

Constraints

Create the linear constraints that each stop has two associated trips, because there must be a trip to each stop and a trip departing each stop.

```
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,:) = whichIdxs'; % Include in the constraint matrix
end
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 for solution. To suppress iterative output, turn off the default display.

```
opts = optimoptions('intlinprog','Display','off');
[x_tsp,costopt,exitflag,output] = intlinprog(dist,intcon,[],[],Aeq,beq,lb,ub,opts);
```

Create a new graph with the solution trips as edges. To do so, round the solution in case some values are not exactly integers, and convert the resulting values to `logical`.

```
x_tsp = logical(round(x_tsp));
Gsol = graph(idxs(x_tsp,1),idxs(x_tsp,2));
```

Visualize Solution

```
hold on
highlight(hGraph,Gsol,'LineStyle','-')
title('Solution with Subtours')
```

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).

Detect the subtours by identifying the connected components in `Gsol`, the graph built with the edges in the current solution. `conncomp` returns a vector with the number of the subtour to which each edge belongs.

```
tourIdxs = conncomp(Gsol);
numtours = max(tourIdxs); % number of subtours
fprintf('# of subtours: %d\n',numtours);
```

```
# of subtours: 27
```

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 allocate
    for ii = 1:numtours
        rowIdx = size(A,1) + 1; % Counter for indexing
        subTourIdx = find(tourIdxs == 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);
    x_tsp = logical(round(x_tsp));
    Gsol = graph(idxs(x_tsp,1),idxs(x_tsp,2));

    % Visualize result
    hGraph.LineStyle = 'none'; % Remove the previous highlighted path
    highlight(hGraph,Gsol,'LineStyle','-')
    drawnow

    % How many subtours this time?
    tourIdxs = conncomp(Gsol);
    numtours = max(tourIdxs); % number of subtours
    fprintf('# of subtours: %d\n',numtours)
end
```

```
# of subtours: 20
```

```
# of subtours: 7
```

```
# of subtours: 9
```

```
# of subtours: 9
```

```
# of subtours: 3
```

```
# of subtours: 2
```

```
# of subtours: 7
```

```
# of subtours: 2
```

```
# of subtours: 1
```

```
title('Solution with Subtours Eliminated');
hold off
```
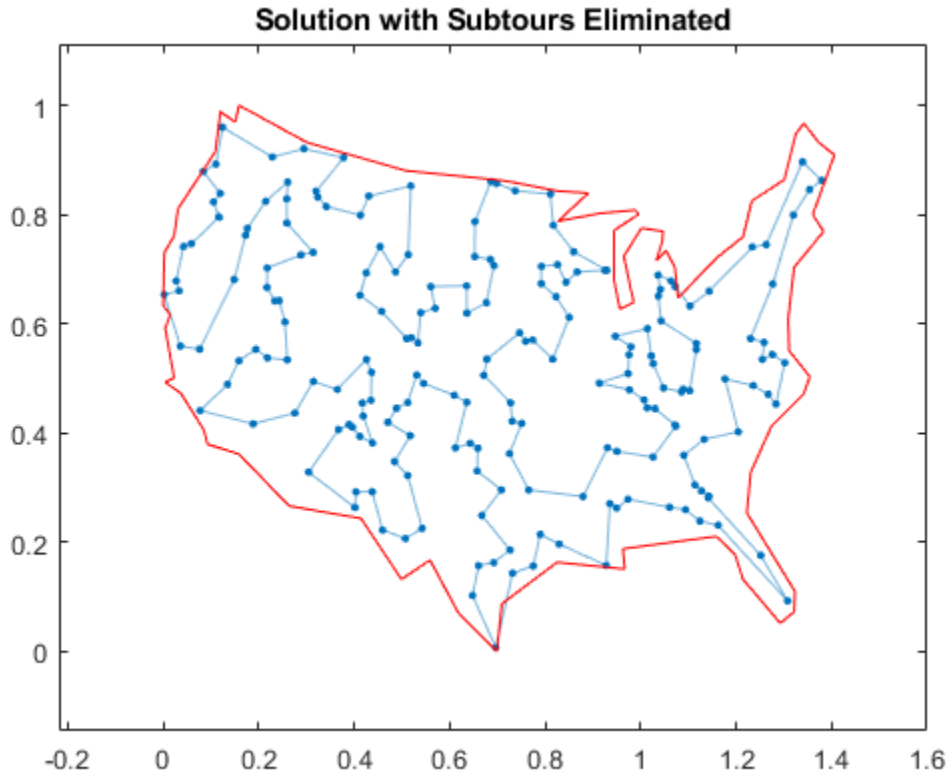

Solution with Subtours Eliminated

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
```

The smallness of the absolute gap implies that the solution is either optimal or has a total length that is close to optimal.

See Also

More About

- "Traveling Salesman Problem: Problem-Based" on page 9-116

Optimal Dispatch of Power Generators: Solver-Based

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.

For the problem-based approach to this problem, see "Optimal Dispatch of Power Generators: Problem-Based" on page 9-122.

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 https://www.nemweb.com.au/REPORTS/CURRENT/ in mid-2013, and is used under their terms https://www.aemo.com.au/privacy-and-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')
```
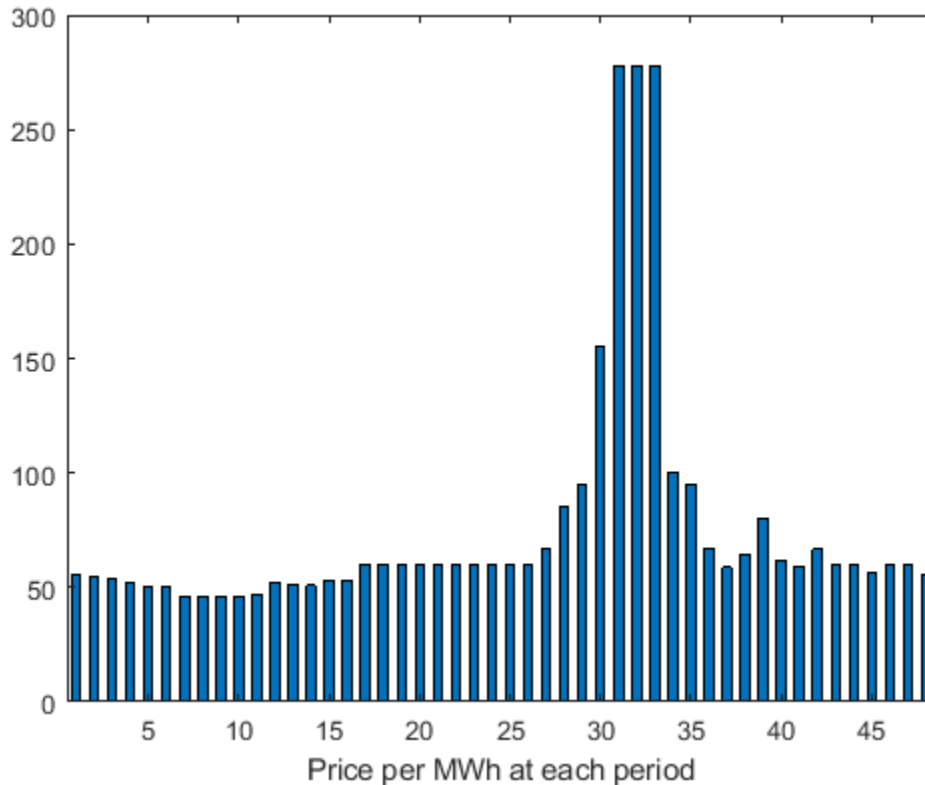
There is a cost to start a generator after it has been off. The other constraint is a maximum fuel usage for the day. The maximum fuel constraint is because you buy your fuel a day ahead of time, so can use only what you just bought.

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 <= i <= 48.
- j = a generator index, 1 <= j <= 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); % 0 for the y variables
lbz = zeros(nPeriods,nGens); % 0 for the z variables
lb = [lby(:);lbz(:)]; % Column vector lower bound
ub = ones(size(lb)); % Binary variables have lower bound 0, 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) are equal
```

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 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.

```matlab
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:

```matlab
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.

```matlab
generatorlevel = lby; % Generation in MW, start with 0s
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

```matlab
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.AbsoluteGapTolerance = 0 (the default value). The intcon
variables are integer within tolerance, options.IntegerTolerance = 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')
```
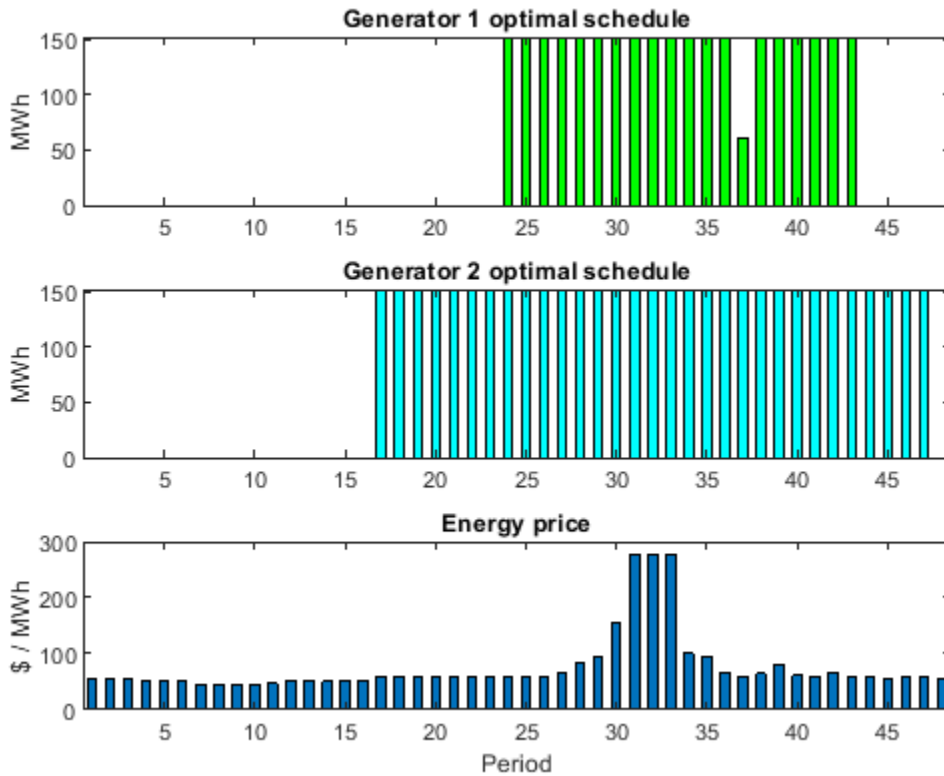

Generator 2 runs longer than generator 1, which you would expect because it is more efficient. Generator 2 runs at its high power level whenever it is on. Generator 1 runs mainly at its high power level, but dips down to low power for one time unit. Each generator runs for one contiguous set of periods 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 = *2×1*

```
    23
    16
```

thegenerator = *2×1*

```
    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);
```

```
Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the
optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon
variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the
default value).
```

```
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')
```
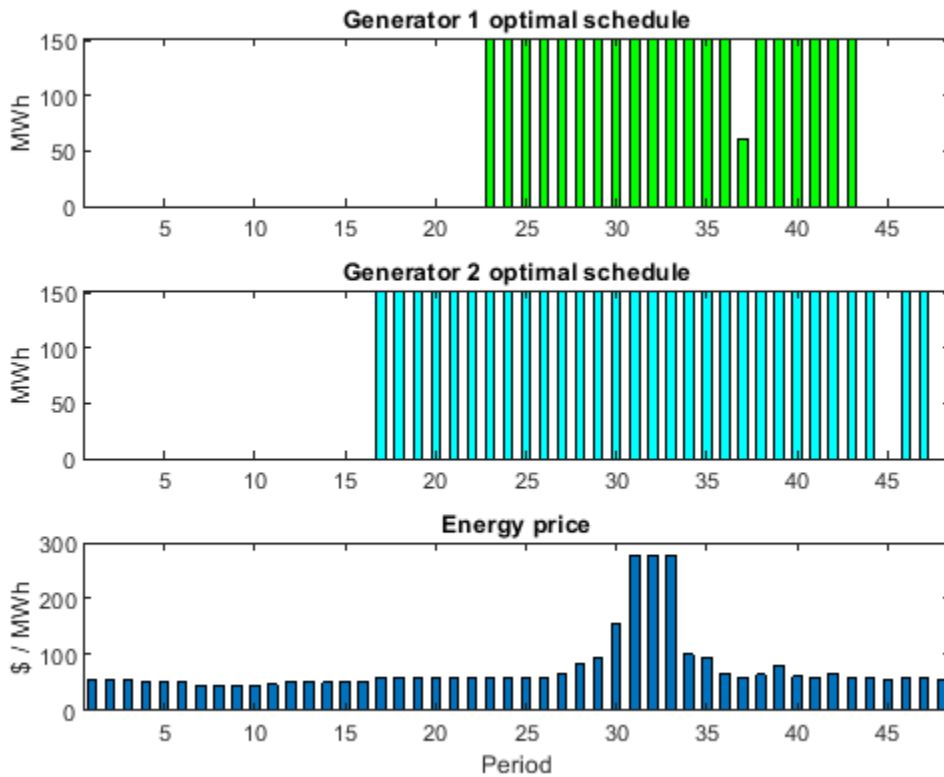
```
starttimes = find(round(zsolutionnew) == 1); % Use round for noninteger results
[theperiod,thegenerator] = ind2sub(size(zsolution),starttimes)
```

theperiod = *3×1*

```
    22
    16
    45
```

thegenerator = *3×1*

```
     1
     2
     2
```

See Also

More About

- "Optimal Dispatch of Power Generators: Problem-Based" on page 9-122

Mixed-Integer Quadratic Programming Portfolio Optimization: Solver-Based

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. For the problem-based approach, see "Mixed-Integer Quadratic Programming Portfolio Optimization: Problem-Based" on page 9-136.

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 $f\text{min} \leq x(i) \leq f\text{max}$ for some fixed fractions $f\text{min} > 0$ and $f\text{max} \geq f\text{min}$.

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

$$v(i)f\text{min} \leq x(i) \leq v(i)f\text{max}.$$

These inequalities both enforce that $x(i)$ and $v(i)$ are zero at exactly the same time, and they also enforce that $f\text{min} \leq x(i) \leq f\text{max}$ 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

$$x^T Q x - z = x_0^T Q x_0 + 2 x_0^T Q \delta - z + O\left(\left|\delta\right|^2\right) \,.$$

Replacing δ by $x - x_0$ gives

$$x^T Q x - z = -x_0^T Q x_0 + 2 x_0^T Q x - z + O\left(\left|x - x_0\right|^2\right) \,.$$

For each intermediate solution x_k you introduce a new linear constraint in x and z as the linear part of the expression above:

$$-x_k^T Q x_k + 2 x_k^T Q x - z \leq 0 \,.$$

This has the form $Ax \leq b$, where $A = 2 x_k^T Q$, 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. Set tighter tolerances than default to help ensure that the problem remains strictly feasible as constraints accumulate.

```
thediff = 1e-4;
iter = 1; % iteration counter
assets = xLinInt(xvars); % the x variables
truequadratic = assets'*Q*assets;
zslack = xLinInt(zvar); % slack variable value
options = optimoptions(options,'LPOptimalityTolerance',1e-10,'RelativeGapTolerance',1e-8,...
                       'ConstraintTolerance',1e-9,'IntegerTolerance',1e-6);
```

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 $Ax \le b$ comes from the linear approximation

$$-x_k^T Q x_k + 2x_k^T Q x - z \le 0 \,.$$

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
    rhs = assets'*Q*assets;                        % right hand side of the linearized constraint
    A = [A;newArow];
    b = [b;rhs];
    % 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 = xLinInt(xvars)'*Q* xLinInt(xvars);
    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)
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.

```
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 $f\mathrm{min} = 0.001$ and $f\mathrm{max} = 0.05$.

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 0.000595107, and the risk-adjusted return is -0.0360382.
```

More elaborate analyses are possible by using features specifically designed for portfolio optimization in Financial Toolbox™. For an example that shows how to use the Portfolio class to directly handle semicontinuous and cardinality constraints, see "Portfolio Optimization with Semicontinuous and Cardinality Constraints" (Financial Toolbox).

See Also

More About

- "Mixed-Integer Quadratic Programming Portfolio Optimization: Problem-Based" on page 9-136

Solve Sudoku Puzzles Via Integer Programming: Solver-Based

This example shows how to solve a Sudoku puzzle using binary integer programming. For the problem-based approach, see "Solve Sudoku Puzzles Via Integer Programming: Problem-Based" on page 9-148.

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 tie breaking 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 $x(i, j, 1), \ldots, x(i, j, 9)$. 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:

$$\sum_{i = 1}^{3} \sum_{j = 1}^{3} x(i + U, j + V, k) = 1, \text{ where } U, V \ \epsilon \ \{0, 3, 6\} .$$

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 solver
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
```

```
LP:                Optimal objective value is 29565.000000.
```

```
Optimal solution found.
```

```
Intlinprog stopped at the root node because the objective value is within a gap
tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).
```

drawSudoku(S)

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
```

See Also

More About

*   "Solve Sudoku Puzzles Via Integer Programming: Problem-Based" on page 9-148

Office Assignments by Binary Integer Programming: Solver-Based

This example shows how to solve an assignment problem by binary integer programming using the `intlinprog` function. For the problem-based approach to this problem, see "Office Assignments by Binary Integer Programming: Problem-Based" on page 9-131.

Office Assignment Problem

You want to assign six people, Marcelo, Rakesh, Peter, Tom, Marjorie, and Mary Ann, to seven offices. Each office can have no more than one person, and each person gets exactly one office. So there will be one empty office. People can give preferences for the offices, and their preferences are considered based on their seniority. The longer they have been at the MathWorks, the higher the seniority. Some offices have windows, some do not, and one window is smaller than others. Additionally, Peter and Tom often work together, so should be in adjacent offices. Marcelo and Rakesh often work together, and should be in adjacent offices.

Office Layout

Offices 1, 2, 3, and 4 are inside offices (no windows). Offices 5, 6, and 7 have windows, but the window in office 5 is smaller than the other two. Here is how the offices are arranged.

```
name = {'Office 1','Office 2','Office 3','Office 4','Office 5','Office 6','Office 7'};
printofficeassign(name)
```


Office layout: windows are red lines
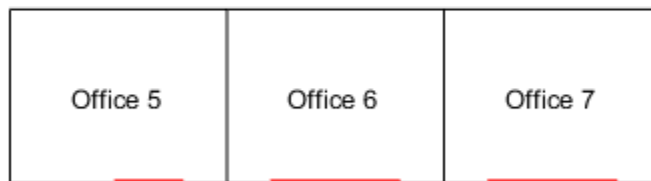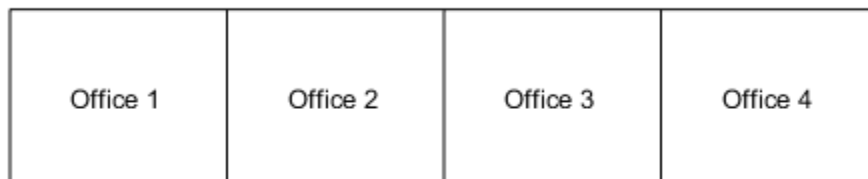
Problem Formulation

You need to formulate the problem mathematically. First, choose what each element of your solution variable x represents in the problem. Since this is a binary integer problem, a good choice is that each element represents a person assigned to an office. If the person is assigned to the office, the variable has value 1. If the person is not assigned to the office, the variable has value 0. Number people as follows:

1 Mary Ann

2 Marjorie

3 Tom

4 Peter

5 Marcelo

6 Rakesh

x is a vector. The elements x(1) to x(7) correspond to Mary Ann being assigned to office 1, office 2, etc., to office 7. The next seven elements correspond to Marjorie being assigned to the seven offices, etc. In all, the x vector has 42 elements, since six people are assigned to seven offices.

Seniority

You want to weight the preferences based on seniority so that the longer you have been at MathWorks, the more your preferences count. The seniority is as follows: Mary Ann 9 years, Marjorie 10 years, Tom 5 years, Peter 3 years, Marcelo 1.5 years, and Rakesh 2 years. Create a normalized weight vector based on seniority.

```
seniority = [9 10 5 3 1.5 2];
weightvector = seniority/sum(seniority);
```

People's Office Preferences

Set up a preference matrix where the rows correspond to offices and the columns correspond to people. Ask each person to give values for each office so that the sum of all their choices, i.e., their column, sums to 100. A higher number means the person prefers the office. Each person's preferences are listed in a column vector.

```
MaryAnn = [0; 0; 0; 0; 10; 40; 50];
Marjorie = [0; 0; 0; 0; 20; 40; 40];
Tom = [0; 0; 0; 0; 30; 40; 30];
Peter = [1; 3; 3; 3; 10; 40; 40];
Marcelo = [3; 4; 1; 2; 10; 40; 40];
Rakesh = [10; 10; 10; 10; 20; 20; 20];
```

The ith element of a person's preference vector is how highly they value the ith office. Thus, the combined preference matrix is as follows.

```
prefmatrix = [MaryAnn Marjorie Tom Peter Marcelo Rakesh];
```

Weight the preferences matrix by `weightvector` to scale the columns by seniority. Also, it's more convenient to reshape this matrix as a vector in column order so that it corresponds to the x vector.

```
PM = prefmatrix * diag(weightvector);
c = PM(:);
```

Objective Function

The objective is to maximize the satisfaction of the preferences weighted by seniority. This is a linear objective function

```
max c'*x
```

or equivalently

```
min -c'*x.
```

Constraints

The first set of constraints requires that each person gets exactly one office, that is for each person, the sum of the x values corresponding to that person is exactly one. For example, since Marjorie is the second person, this means that `sum(x(8:14))=1`. Represent these linear constraints in an equality matrix `Aeq` and vector `beq`, where `Aeq*x = beq`, by building the appropriate matrices. The matrix `Aeq` consists of ones and zeros. For example, the second row of `Aeq` will correspond to Marjorie getting one office, so the row looks like this:

```
0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0
```

There are seven 1s in columns 8 through 14 and 0s elsewhere. Then `Aeq(2,:)*x = 1` is equivalent to `sum(x(8:14)) = 1`.

```
numOffices = 7;
numPeople = 6;
numDim = numOffices * numPeople;
onesvector = ones(1,numOffices);
```

Each row of `Aeq` corresponds to one person.

```
Aeq = blkdiag(onesvector,onesvector,onesvector,onesvector, ...
    onesvector,onesvector);
beq = ones(numPeople,1);
```

The second set of constraints are inequalities. These constraints specify that each office has no more than one person in it, i.e., each office has one person in it, or is empty. Build the matrix `A` and the vector `b` such that `A*x <= b` to capture these constraints. Each row of `A` and `b` corresponds to an office and so row 1 corresponds to people assigned to office 1. This time, the rows have this type of pattern, for row 1:

```
1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 ... 1 0 0 0 0 0 0
```

Each row after this is similar but shifted (circularly) to the right by one spot. For example, row 3 corresponds to office 3 and says that `A(3,:)*x <= 1`, i.e., office 3 cannot have more than one person in it.

```
A = repmat(eye(numOffices),1,numPeople);
b = ones(numOffices,1);
```

The next set of constraints are also inequalities, so add them to the matrix `A` and vector `b`, which already contain the inequalities from above. You want Tom and Peter no more than one office away from each other, and the same with Marcelo and Rakesh. First, build the distance matrix of the offices based on their physical locations and using approximate Manhattan distances. This is a symmetric matrix.

```
D = zeros(numOffices);
```

Set up the top right half of the matrix.

```
D(1,2:end) = [1 2 3 2 3 4];
D(2,3:end) = [1 2 1 2 3];
D(3,4:end) = [1 2 1 2];
D(4,5:end) = [3 2 1];
D(5,6:end) = [1 2];
D(6,end)   = 1;
```

The lower left half is the same as the upper right.

```
D = triu(D)' + D;
```

Find the offices that are more than one distance unit away.

```
[officeA,officeB] = find(D > 1);
numPairs = length(officeA)
```

```
numPairs = 26
```

This finds `numPairs` pairs of offices that are not adjacent. For these pairs, if Tom occupies one office in the pair, then Peter cannot occupy the other office in the pair. If he did, they would not be adjacent. The same is true for Marcelo and Rakesh. This gives `2*numPairs` more inequality constraints that you add to A and b.

Add enough rows to A to accommodate these constraints.

```
numrows = 2*numPairs + numOffices;
A((numOffices+1):numrows, 1:numDim) = zeros(2*numPairs,numDim);
```

For each pair of offices in `numPairs`, for the `x(i)` that corresponds to Tom in `officeA` and for the `x(j)` that corresponds to Peter in `OfficeB`,

```
x(i) + x(j) <= 1.
```

This means that either Tom or Peter can occupy one of these offices, but they both cannot.

Tom is person 3 and Peter is person 4.

```
tom = 3;
peter = 4;
```

Marcelo is person 5 and Rakesh is person 6.

```
marcelo = 5;
rakesh = 6;
```

The following anonymous functions return the index in x corresponding to Tom, Peter, Marcelo and Rakesh respectively in office i.

```
tom_index=@(officenum) (tom-1)*numOffices+officenum;
peter_index=@(officenum) (peter-1)*numOffices+officenum;
marcelo_index=@(officenum) (marcelo-1)*numOffices+officenum;
rakesh_index=@(officenum) (rakesh-1)*numOffices+officenum;

for i = 1:numPairs
```

```
        tomInOfficeA = tom_index(officeA(i));
        peterInOfficeB = peter_index(officeB(i));
        A(i+numOffices, [tomInOfficeA, peterInOfficeB]) = 1;

        % Repeat for Marcelo and Rakesh, adding more rows to A and b.
        marceloInOfficeA = marcelo_index(officeA(i));
        rakeshInOfficeB = rakesh_index(officeB(i));
        A(i+numPairs+numOffices, [marceloInOfficeA, rakeshInOfficeB]) = 1;
    end
    b(numOffices+1:numOffices+2*numPairs) = ones(2*numPairs,1);
```

Solve Using `intlinprog`

The problem formulation consists of a linear objective function

```
min -c'*x
```

subject to the linear constraints

```
Aeq*x = beq
```

```
A*x <= b
```

all variables binary

You already made the `A`, `b`, `Aeq`, and `beq` entries. Now set the objective function.

```
f = -c;
```

To ensure that the variables are binary, put lower bounds of 0, upper bounds of 1, and set `intvars` to represent all variables.

```
lb = zeros(size(f));
ub = lb + 1;
intvars = 1:length(f);
```

Call `intlinprog` to solve the problem.

```
[x,fval,exitflag,output] = intlinprog(f,intvars,A,b,Aeq,beq,lb,ub);
```

```
LP:                Optimal objective value is -33.868852.

Cut Generation:    Applied 1 Gomory cut.
                   Lower bound is -33.836066.
                   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
options.AbsoluteGapTolerance = 0 (the default value). The intcon variables are integer within to
options.IntegerTolerance = 1e-05 (the default value).
```

View the Solution -- Who Got Each Office?

```
numPeople = 7; office = cell(numPeople,1);
for i=1:numPeople
    office{i} = find(x(i:numPeople:end)); % people index in office
end
```

```
people = {'Mary Ann', 'Marjorie',' Tom   ',' Peter  ','Marcelo ',' Rakesh '};
for i=1:numPeople
    if isempty(office{i})
        name{i} = ' empty  ';
    else
        name{i} = people(office{i});
    end
end

printofficeassign(name);
title('Solution of the Office Assignment Problem');
```

Solution of the Office Assignment Problem

Solution Quality

For this problem, the satisfaction of the preferences by seniority is maximized to the value of -fval. exitflag = 1 tells you that intlinprog converged to an optimal solution. The output structure gives information about the solution process, such as how many nodes were explored, and the gap between the lower and upper bounds in the branching calculation. In this case, no branch-and-bound nodes were generated, meaning the problem was solved without a branch-and-bound step. The gap is 0, meaning the solution is optimal, with no difference between the internally calculated lower and upper bounds on the objective function.

```
fval,exitflag,output

fval = -33.8361

exitflag = 1
```

```
output = struct with fields:
          relativegap: 0
          absolutegap: 0
        numfeaspoints: 1
             numnodes: 0
        constrviolation: 0
              message: 'Optimal solution found.↵Intlinprog stopped at the root node because the ol
```

See Also

More About

- "Office Assignments by Binary Integer Programming: Problem-Based" on page 9-131

Cutting Stock Problem: Solver-Based

This example shows how to solve a cutting stock problem using linear programming with an integer linear programming subroutine. The example uses the "Solver-Based Optimization Problem Setup" approach. For the problem-based approach, see "Cutting Stock Problem: Problem-Based" on page 9-143.

Problem Overview

A lumber mill starts with trees that have been trimmed to fixed-length logs. Specify the fixed log length.

```
logLength = 40;
```

The mill then cuts the logs into fixed lengths suitable for further processing. The problem is how to make the cuts so that the mill satisfies a set of orders with the fewest logs.

Specify these fixed lengths and the order quantities for the lengths.

```
lengthlist = [8; 12; 16; 20];
quantity = [90; 111; 55; 30];
nLengths = length(lengthlist);
```

Assume that there is no material loss in making cuts, and no cost for cutting.

Linear Programming Formulation

Several authors, including Ford and Fulkerson [1] and Gilmore and Gomory [2], suggest the following procedure, which you implement in the next section. A cutting pattern is a set of lengths to which a single log can be cut.

Log   Log with cutting pattern

Instead of generating every possible cutting pattern, it is more efficient to generate cutting patterns as the solution of a subproblem. Starting from a base set of cutting patterns, solve the linear programming problem of minimizing the number of logs used subject to the constraint that the cuts, using the existing patterns, satisfy the demands.

After solving that problem, generate a new pattern by solving an integer linear programming subproblem. The subproblem is to find the best new pattern, meaning the number of cuts from each length in `lengthlist` that add up to no more than the total possible length `logLength`. The quantity to optimize is the reduced cost of the new pattern, which is one minus the sum of the Lagrange multipliers for the current solution times the new cutting pattern. If this quantity is negative, then bringing that pattern into the linear program will improve its objective. If not, then no better cutting pattern exists, and the patterns used so far give the optimal linear programming solution. The reason for this conclusion is exactly parallel to the reason for when to stop the primal simplex method: the method terminates when there is no variable with a negative reduced cost. The problem in this example terminates when there is no pattern with negative reduced cost. For details and an example, see Column generation algorithms and its references.

After solving the linear programming problem in this way, you can have noninteger solutions. Therefore, solve the problem once more, using the generated patterns and constraining the variables to have integer values.

MATLAB Solver-Based Formulation

A pattern, in this formulation, is a vector of integers containing the number of cuts of each length in `lengthlist`. Arrange a matrix named `patterns` to store the patterns, where each column in the matrix gives a pattern. For example,

$$\text{patterns} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

The first pattern (column) represents two cuts of length 8 and one cut of length 20. The second pattern represents two cuts of length 12 and one cut of length 16. Each is a feasible pattern because the total of the cuts is no more than `logLength = 40`.

In this formulation, if `x` is a column vector of integers containing the number of times each pattern is used, then `patterns*x` is a column vector giving the number of cuts of each type. The constraint of meeting demand is `patterns*x >= quantity`. For example, using the previous `patterns` matrix, suppose that $x = \begin{bmatrix} 45 \\ 56 \end{bmatrix}$. (This x uses 101 logs.) Then

$$\text{patterns} * x = \begin{bmatrix} 90 \\ 112 \\ 56 \\ 45 \end{bmatrix},$$

which represents a feasible solution because the result exceeds the demand

$$\text{quantity} = \begin{bmatrix} 90 \\ 111 \\ 55 \\ 30 \end{bmatrix}.$$

To have an initial feasible cutting pattern, use the simplest patterns, which have just one length of cut. Use as many cuts of that length as feasible for the log.

```
patterns = diag(floor(logLength./lengthlist));
nPatterns = size(patterns,2);
```

To generate new patterns from the existing ones based on the current Lagrange multipliers, solve a subproblem. Call the subproblem in a loop to generate patterns until no further improvement is found. The subproblem objective depends only on the current Lagrange multipliers. The variables are nonnegative integers representing the number of cuts of each length. The only constraint is that the sum of the lengths of the cuts in a pattern is no more than the log length. Create a lower bound vector `lb2` and matrices `A2` and `b2` that represent these bounds and linear constraints.

```
lb2 = zeros(nLengths,1);
A2 = lengthlist';
b2 = logLength;
```

To avoid unnecessary feedback from the solvers, set the `Display` option to `'off'` for both the outer loop and the inner subproblem solution.

```
lpopts = optimoptions('linprog','Display','off');
ipopts = optimoptions('intlinprog',lpopts);
```

Initialize the variables for the loop.

```
reducedCost = -Inf;
reducedCostTolerance = -0.0001;
exitflag = 1;
```

Call the loop.

```
while reducedCost < reducedCostTolerance && exitflag > 0
    lb = zeros(nPatterns,1);
    f = lb + 1;
    A = -patterns;
    b = -quantity;

    [values,nLogs,exitflag,~,lambda] = linprog(f,A,b,[],[],lb,[],lpopts);
    if exitflag > 0
        fprintf('Using %g logs\n',nLogs);
        % Now generate a new pattern, if possible
        f2 = -lambda.ineqlin;
        [values,reducedCost,pexitflag] = intlinprog(f2,1:nLengths,A2,b2,[],[],lb2,[],ipopts);
        reducedCost = 1 + reducedCost; % continue if this reducedCost is negative
        newpattern = round(values);
        if pexitflag > 0 && reducedCost < reducedCostTolerance
            patterns = [patterns newpattern];
            nPatterns = nPatterns + 1;
        end
    end
end
```

```
Using 97.5 logs
Using 92 logs
Using 89.9167 logs
Using 88.3 logs
```

You now have the solution of the linear programming problem. To complete the solution, solve the problem again with the final patterns, using `intlinprog` with all variables being integers. Also, compute the waste, which is the quantity of unused logs (in feet) for each pattern and for the problem as a whole.

```
if exitflag <= 0
    disp('Error in column generation phase')
else
    [values,logsUsed,exitflag] = intlinprog(f,1:length(lb),A,b,[],[],lb,[],[],ipopts);
    if exitflag > 0
        values = round(values);
        logsUsed = round(logsUsed);
        fprintf('Optimal solution uses %g logs\n', logsUsed);
        totalwaste = sum((patterns*values - quantity).*lengthlist); % waste due to overproduction
        for j = 1:size(values)
            if values(j) > 0
                fprintf('Cut %g logs with pattern\n',values(j));
                for w = 1:size(patterns,1)
                    if patterns(w,j) > 0
                        fprintf('    %d cut(s) of length %d\n', patterns(w,j),lengthlist(w));
                    end
```

```
                end
                wastej = logLength - dot(patterns(:,j),lengthlist); % waste due to pattern ineff:
                totalwaste = totalwaste + wastej;
            fprintf('    Waste of this pattern is %g\n', wastej);
            end
        end
        fprintf('Total waste in this problem is %g.\n',totalwaste);
    else
        disp('Error in final optimization')
    end
end
```

```
Optimal solution uses 89 logs

Cut 15 logs with pattern

    2 cut(s) of length 20

    Waste of this pattern is 0

Cut 18 logs with pattern

    1 cut(s) of length 8
    2 cut(s) of length 16

    Waste of this pattern is 0

Cut 37 logs with pattern

    2 cut(s) of length 8
    2 cut(s) of length 12

    Waste of this pattern is 0

Cut 19 logs with pattern

    2 cut(s) of length 12
    1 cut(s) of length 16

    Waste of this pattern is 0

Total waste in this problem is 28.
```

Part of the waste is due to overproduction, because the mill cuts one log into three 12-foot pieces, but uses only one. Part of the waste is due to pattern inefficiency, because the three 12-foot pieces are 4 feet short of the total length of 40 feet.

References

[1] Ford, L. R., Jr. and D. R. Fulkerson. *A Suggested Computation for Maximal Multi-Commodity Network Flows.* Management Science 5, 1958, pp. 97-101.

[2] Gilmore, P. C., and R. E. Gomory. *A Linear Programming Approach to the Cutting Stock Problem--Part II.* Operations Research 11, No. 6, 1963, pp. 863-888.

See Also

More About

- "Cutting Stock Problem: Problem-Based" on page 9-143

Factory, Warehouse, Sales Allocation Model: Problem-Based

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. For the solver-based approach, see "Factory, Warehouse, Sales Allocation Model: Solver-Based" on page 9-57.

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

$$\sum_w x(p, f, w) \le pcap(f, p) \text{ (capacity of factory).}$$

$$\sum_f x(p, f, w) = \sum_s (d(s, p) \cdot y(s, w)) \text{ (demand is met).}$$

$$\sum_p \sum_s \frac{d(s, p)}{turn(p)} \cdot y(s, w) \le wcap(w) \text{ (capacity of warehouse).}$$

$$\sum_w y(s, w) = 1 \text{ (each sales outlet associates to one warehouse).}$$

$$x(p, f, w) \ge 0 \text{ (nonnegative production).}$$

$$y(s, w) \; \epsilon \; \{0, 1\} \text{ (binary } y\text{).}$$

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(1) % 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');
lgnd = legend('Factory','Warehouse','Sales outlet','Location','EastOutside');
lgnd.AutoUpdate = 'off';
xlim([0 N+1]);ylim([0 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,1) + 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 Variables and Constraints

To begin specifying the problem, generate the distance arrays `distfw(i,j)` and `distsw(i,j)`.

```
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
```

Create variables for the optimization problem. `x` represents the production, a continuous variable, with dimension P-by-F-by-W. `y` represents the binary allocation of sales outlet to warehouse, an S-by-W variable.

```
x = optimvar('x',P,F,W,'LowerBound',0);
y = optimvar('y',S,W,'Type','integer','LowerBound',0,'UpperBound',1);
```

Now create the constraints. The first constraint is a capacity constraint on production.

```
capconstr = sum(x,3) <= pcap';
```

The next constraint is that the demand is met at each sales outlet.

```
demconstr = squeeze(sum(x,2)) == d'*y;
```

9-111

There is a capacity constraint at each warehouse.

```
warecap = sum(diag(1./turn)*(d'*y),1) <= wcap';
```

Finally, there is a requirement that each sales outlet connects to exactly one warehouse.

```
salesware = sum(y,2) == ones(S,1);
```

Create Problem and Objective

Create an optimization problem.

```
factoryprob = optimproblem;
```

The objective function has three parts. The first part is the sum of the production costs.

```
objfun1 = sum(sum(sum(x,3).*(pcost'),2),1);
```

The second part is the sum of the transportation costs from factories to warehouses.

```
objfun2 = 0;
for p = 1:P
    objfun2 = objfun2 + tcost(p)*sum(sum(squeeze(x(p,:,:)).*distfw));
end
```

The third part is the sum of the transportation costs from warehouses to sales outlets.

```
r = sum(distsw.*y,2); % r is a length s vector
v = d*(tcost(:));
objfun3 = sum(v.*r);
```

The objective function to minimize is the sum of the three parts.

```
factoryprob.Objective = objfun1 + objfun2 + objfun3;
```

Include the constraints in the problem.

```
factoryprob.Constraints.capconstr = capconstr;
factoryprob.Constraints.demconstr = demconstr;
factoryprob.Constraints.warecap = warecap;
factoryprob.Constraints.salesware = salesware;
```

Solve the Problem

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','PlotFcn',@optimplotmilp);
```

Call the solver to find the solution.

```
[sol,fval,exitflag,output] = solve(factoryprob,'options',opts);
```
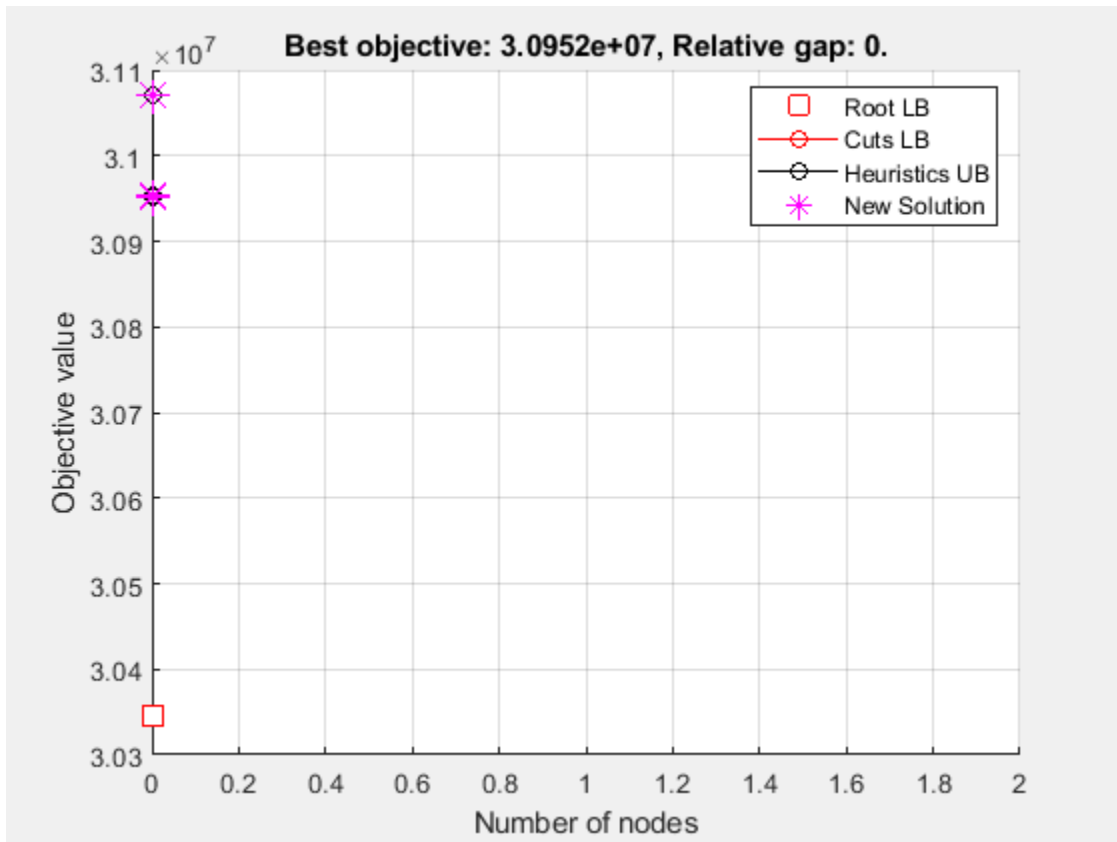
```
if isempty(sol) % If the problem is infeasible or you stopped early with no solution
    disp('The solver did not return a solution.')
    return % Stop the script because there is nothing to examine
end
```

Examine the Solution

Examine the exit flag and the infeasibility of the solution.

```
exitflag
```

```
exitflag =
    OptimalSolution
```

```
infeas1 = max(max(infeasibility(capconstr,sol)))
```

```
infeas1 = 9.0949e-13
```

```
infeas2 = max(max(infeasibility(demconstr,sol)))
```

```
infeas2 = 8.0718e-12
```

```
infeas3 = max(infeasibility(warecap,sol))
```

```
infeas3 = 0
```

```
infeas4 = max(infeasibility(salesware,sol))
```

```
infeas4 = 2.4425e-15
```

Round the y portion of the solution to be exactly integer-valued. To understand why these variables might not be exactly integers, see "Some "Integer" Solutions Are Not Integers" on page 9-53.

```
sol.y = round(sol.y); % get integer solutions
```

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(sol.y,1)
```

```
outlets = 1×20
```

| 2 | 0 | 3 | 2 | 2 | 2 | 3 | 2 | 3 | 2 | 1 | 0 | 0 | 3 | 4 | 3 |

Plot the connection between each sales outlet and its warehouse.

```
figure(h);
hold on
for ii = 1:S
    jj = find(sol.y(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);
    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.

See Also

More About

- "Factory, Warehouse, Sales Allocation Model: Solver-Based" on page 9-57
- "Problem-Based Optimization Workflow" on page 10-2

Traveling Salesman Problem: Problem-Based

This example shows how to use binary integer programming to solve the classic traveling 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.

For the solver-based approach to this problem, see "Traveling Salesman Problem: Solver-Based" on page 9-66.

Problem Formulation

Formulate the traveling 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.

Generate Stops

Generate random stops inside a crude polygonal representation of the continental U.S.

```
load('usborder.mat','x','y','xx','yy');
rng(3,'twister') % Makes 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,x,y) % tTest if inside the border
        stopsLon(n) = xp;
        stopsLat(n) = yp;
        n = n+1;
    end
end
```

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'*trips
```

where `trips` is the binary vector representing the trips that the solution takes. This is the distance of a tour that you try to minimize.

Create Graph and Draw Map

Represent the problem as a graph. Create a graph where the stops are nodes and the trips are edges.

```
G = graph(idxs(:,1),idxs(:,2));
```

Display the stops using a graph plot. Plot the nodes without the graph edges.

```
figure
hGraph = plot(G,'XData',stopsLon,'YData',stopsLat,'LineStyle','none','NodeLabel',{});
hold on
% Draw the outside border
plot(x,y,'r-')
hold off
```
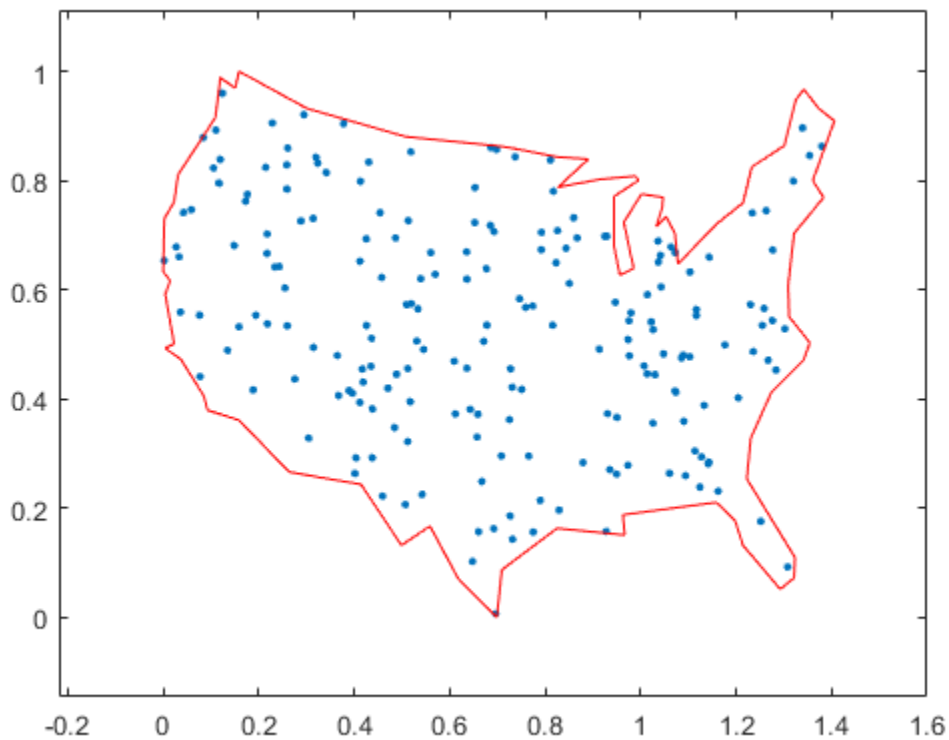
Create Variables and Problem

Create an optimization problem with binary optimization variables representing the potential trips.

```
tsp = optimproblem;
trips = optimvar('trips',lendist,1,'Type','integer','LowerBound',0,'UpperBound',1);
```

Include the objective function in the problem.

```
tsp.Objective = dist'*trips;
```

Constraints

Create the linear constraints that each stop has two associated trips, because there must be a trip to each stop and a trip departing each stop.

Use the graph representation to identify all trips starting or ending at a stop by finding all edges connecting to that stop. For each stop, create the constraint that the sum of trips for that stop equals two.

```
constr2trips = optimconstr(nStops,1);
for stop = 1:nStops
    whichIdxs = outedges(G,stop); % Identify trips associated with the stop
    constr2trips(stop) = sum(trips(whichIdxs)) == 2;
end
tsp.Constraints.constr2trips = constr2trips;
```

Solve Initial Problem

The problem is ready to be solved. To suppress iterative output, turn off the default display.

```
opts = optimoptions('intlinprog','Display','off');
tspsol = solve(tsp,'options',opts)
```

```
tspsol = struct with fields:
    trips: [19900×1 double]
```

Visualize Solution

Create a new graph with the solution trips as edges. To do so, round the solution in case some values are not exactly integers, and convert the resulting values to `logical`.

```
tspsol.trips = logical(round(tspsol.trips));
Gsol = graph(idxs(tspsol.trips,1),idxs(tspsol.trips,2));
```

Overlay the new graph on the existing plot and highlight its edges.

```
hold on
highlight(hGraph,Gsol,'LineStyle','-')
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).

Detect the subtours by identifying the connected components in `Gsol`, the graph built with the edges in the current solution. `conncomp` returns a vector with the number of the subtour to which each edge belongs.

```
tourIdxs = conncomp(Gsol);
numtours = max(tourIdxs); % Number of subtours
fprintf('# of subtours: %d\n',numtours);
```

```
# of subtours: 27
```

Include the linear inequality constraints to eliminate subtours, and repeatedly call the solver, until just one subtour remains.

```matlab
% Index of added constraints for subtours
k = 1;
while numtours > 1 % Repeat until there is just one subtour
    % Add the subtour constraints
    for ii = 1:numtours
        inSubTour = (tourIdxs == ii); % Edges in current subtour
        a = all(inSubTour(idxs),2); % Complete graph indices with both ends in subtour
        constrname = "subtourconstr" + num2str(k);
        tsp.Constraints.(constrname) = sum(trips(a)) <= (nnz(inSubTour) - 1);
        k = k + 1;
    end

    % Try to optimize again
    [tspsol,fval,exitflag,output] = solve(tsp,'options',opts);
    tspsol.trips = logical(round(tspsol.trips));
    Gsol = graph(idxs(tspsol.trips,1),idxs(tspsol.trips,2));

    % Plot new solution
    hGraph.LineStyle = 'none'; % Remove the previous highlighted path
    highlight(hGraph,Gsol,'LineStyle','-')
    drawnow

    % How many subtours this time?
    tourIdxs = conncomp(Gsol);
    numtours = max(tourIdxs); % Number of subtours
    fprintf('# of subtours: %d\n',numtours)
end
```

```
# of subtours: 20

# of subtours: 7

# of subtours: 9

# of subtours: 9

# of subtours: 3

# of subtours: 2

# of subtours: 7

# of subtours: 2

# of subtours: 1
```

```matlab
title('Solution with Subtours Eliminated');
hold off
```

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
```

The smallness of the absolute gap implies that the solution is either optimal or has a total length that is close to optimal.

See Also

More About

*   "Traveling Salesman Problem: Solver-Based" on page 9-66
*   "Problem-Based Optimization Workflow" on page 10-2

Optimal Dispatch of Power Generators: Problem-Based

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.

For the solver-based approach to this problem, see "Optimal Dispatch of Power Generators: Solver-Based" on page 9-72.

Problem Definition

The electricity market has different prices at different times of day. If you have generators supplying electricity, you can take advantage of this variable pricing by scheduling your generators to operate when prices are high. Suppose that you control two generators. 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. Fuel consumption is 0 when the generator is off.

You can assign a power level to each generator for each half-hour time interval during a day (24 hours, so 48 intervals). Based on historical records, assume that you know the revenue per megawatt-hour (MWh) that you receive in each time interval. The data for this example is from the Australian Energy Market Operator `https://www.nemweb.com.au/REPORTS/CURRENT/` in mid-2013, and is used under their terms https://www.aemo.com.au/privacy-and-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
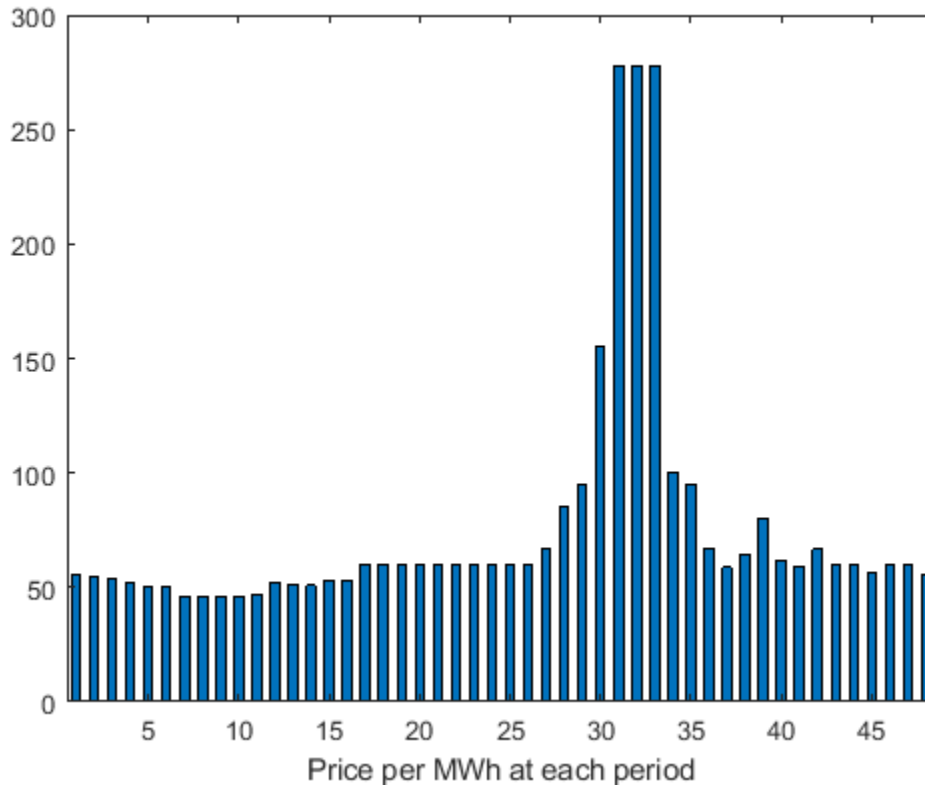
There is a cost to start a generator after it has been off. Also, there is a constraint on the maximum fuel usage for the day. This constraint exists because you buy your fuel a day ahead of time, so you can use only what you just bought.

Problem Notation and Parameters

You can formulate the scheduling problem as a binary integer programming problem. Define indexes `i`, `j`, and `k`, and a binary scheduling vector `y`, as follows:

- `nPeriods` = the number of time periods, 48 in this case.

- `i` = a time period, 1 <= i <= 48.

- `j` = a generator index, 1 <= j <= 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`.

Determine when a generator starts after being off. To do so, define the auxiliary binary variable `z(i,j)` that indicates whether to charge for turning on generator `j` at period `i`.

- `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`.

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 and 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 problem form. The variables `y(i,j,k)` represent the solution of the problem, and the auxiliary variables `z(i,j)` indicate whether to charge for turning on a generator. `y` is an `nPeriods-by-nGens-by-2` array, and `z` is an `nPeriods-by-nGens` array. All variables are binary.

```
y = optimvar('y',nPeriods,nGens,{'Low','High'},'Type','integer','LowerBound',0,...
    'UpperBound',1);
z = optimvar('z',nPeriods,nGens,'Type','integer','LowerBound',0,...
    'UpperBound',1);
```

Linear Constraints

To ensure that the power level has no more than one component equal to 1, set a linear inequality constraint.

```
powercons = y(:,:,'Low') + y(:,:,'High') <= 1;
```

The running cost per period is the cost of fuel for that period. For generator `j` operating at level `k`, the cost is `fuelPrice * fuel(j,k)`.

Create an expression `fuelUsed` that accounts for all the fuel used.

```
yFuel = zeros(nPeriods,nGens,2);
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

fuelUsed = sum(sum(sum(y.*yFuel)));
```

The constraint is that the fuel used is no more than the fuel available.

```
fuelcons = fuelUsed <= 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 of the y variables? 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 these linear inequality constraints in the problem formulation.

$sum\_k\ (\ -\ y(i,j,k)\ +\ y(i+1,j,k)\ )\ -\ z(i,j)\ <\ =\ 0$.

Also, include the z variables in the objective function cost. With the z variables in the objective function, the solver attempts to lower their values, meaning it tries to set them all equal to 0. But for those intervals when a generator turns on, the linear inequality forces $z(i,j)$ to equal 1.

Create an auxiliary variable w that represents $y(i+1,j,k)\ -\ y(i,j,k)$. Represent the generator startup inequality in terms of w.

```
w = optimexpr(nPeriods,nGens); % Allocate w
idx = 1:(nPeriods-1);
w(idx,:) = y(idx+1,:,'Low') - y(idx,:,'Low') + y(idx+1,:,'High') - y(idx,:,'High');
w(nPeriods,:) = y(1,:,'Low') - y(nPeriods,:,'Low') + y(1,:,'High') - y(nPeriods,:,'High');
switchcons = w - z <= 0;
```

Define Objective

The objective function includes fuel costs for running the generators, revenue from running the generators, and costs for starting the generators.

```
generatorlevel  = zeros(size(yFuel));
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 = `y.*generatorlevel.*poolPrice`.

```
revenue = optimexpr(size(y));
for ii = 1:nPeriods
    revenue(ii,:,:) = poolPrice(ii)*y(ii,:,:).*generatorlevel(ii,:,:);
end
```

The total fuel cost = `fuelUsed*fuelPrice`.

```
fuelCost = fuelUsed*fuelPrice;
```

The generator startup cost = z*`startCost`.

```
startingCost = z*startCost;
```

The profit = incoming revenue - total fuel cost - startup cost.

```
profit = sum(sum(sum(revenue))) - fuelCost - sum(sum(startingCost));
```

Solve the Problem

Create an optimization problem and include the objective and constraints.

```
dispatch = optimproblem('ObjectiveSense','maximize');
dispatch.Objective = profit;
dispatch.Constraints.switchcons = switchcons;
dispatch.Constraints.fuelcons = fuelcons;
dispatch.Constraints.powercons = powercons;
```

To save space, suppress iterative display.

```
options = optimoptions('intlinprog','Display','final');
```

Solve the problem.

```
[dispatchsol,fval,exitflag,output] = solve(dispatch,'options',options);
```

```
Solving problem using intlinprog.

Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the
optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon
variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the
default value).
```

Examine the Solution

Plot the solution as a function of time.

```
subplot(3,1,1)
bar(dispatchsol.y(:,1,1)*gen(1,1)+dispatchsol.y(:,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(dispatchsol.y(:,2,1)*gen(1,1)+dispatchsol.y(:,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')
```
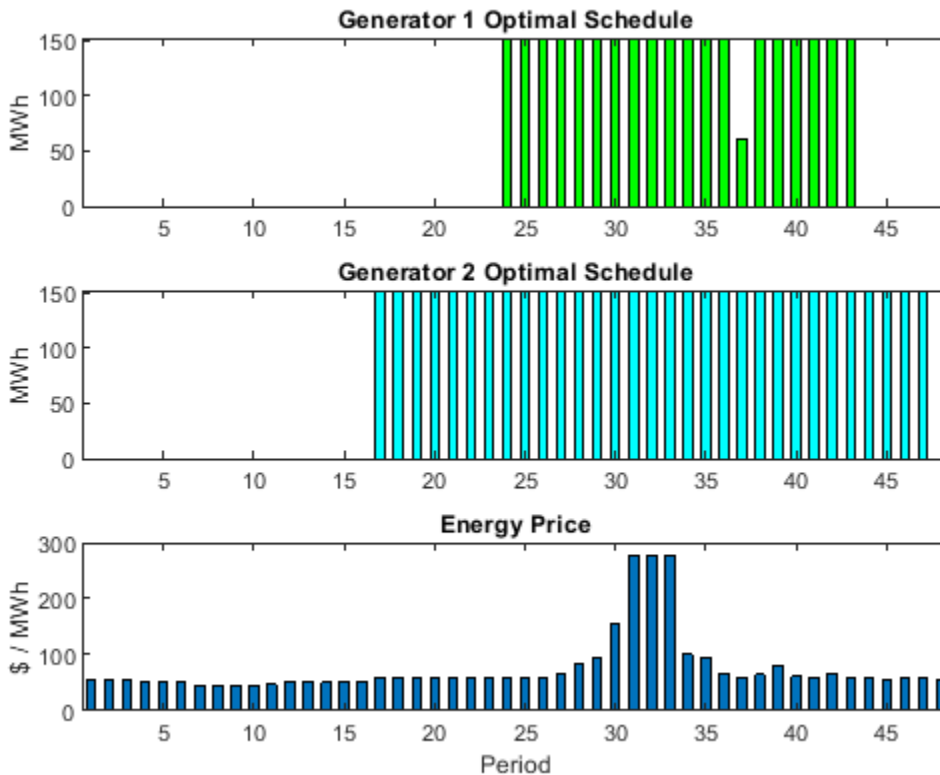
Generator 2 runs longer than generator 1, which you would expect because it is more efficient. Generator 2 runs at its high power level whenever it is on. Generator 1 runs mainly at its high power level, but dips down to low power for one time unit. Each generator runs for one contiguous set of periods daily, and, therefore, incurs only one startup cost each day.

Check that the z variable is 1 for the periods when the generators start.

```
starttimes = find(round(dispatchsol.z) == 1); % Use round for noninteger results
[theperiod,thegenerator] = ind2sub(size(dispatchsol.z),starttimes)
```

theperiod = *2×1*

        23
        16

thegenerator = *2×1*

        1
        2

The periods when the generators start match the plots.

Compare to Lower Penalty for Startup

If you specify a lower value for startCost, the solution involves multiple generation periods.

```
startCost = 500; % Choose a lower penalty for starting the generators
startingCost = z*startCost;
profit = sum(sum(sum(revenue))) - fuelCost - sum(sum(startingCost));
dispatch.Objective = profit;
[dispatchsolnew,fvalnew,exitflagnew,outputnew] = solve(dispatch,'options',options);
```

Solving problem using intlinprog.

Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the
optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon
variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the
default value).

```
subplot(3,1,1)
bar(dispatchsolnew.y(:,1,1)*gen(1,1)+dispatchsolnew.y(:,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(dispatchsolnew.y(:,2,1)*gen(1,1)+dispatchsolnew.y(:,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 1 Optimal Schedule, Generator 2 Optimal Schedule, Energy Price
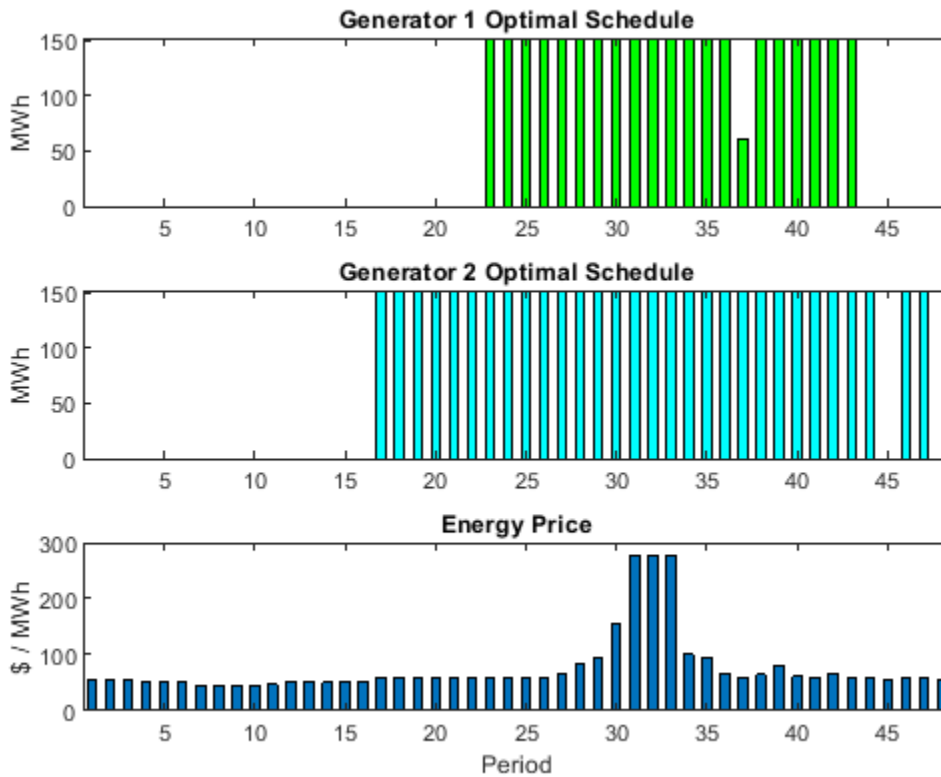
```
starttimes = find(round(dispatchsolnew.z) == 1); % Use round for noninteger results
[theperiod,thegenerator] = ind2sub(size(dispatchsolnew.z),starttimes)
```

theperiod = *3×1*

```
    22
    16
    45
```

thegenerator = *3×1*

```
     1
     2
     2
```

See Also

More About

*   "Optimal Dispatch of Power Generators: Solver-Based" on page 9-72
*   "Problem-Based Optimization Workflow" on page 10-2

Office Assignments by Binary Integer Programming: Problem-Based

This example shows how to solve an assignment problem by binary integer programming using the optimization problem approach. For the solver-based approach, see "Office Assignments by Binary Integer Programming: Solver-Based" on page 9-96.

Office Assignment Problem

You want to assign six people, Marcelo, Rakesh, Peter, Tom, Marjorie, and Mary Ann, to seven offices. Each office can have no more than one person, and each person gets exactly one office. So there will be one empty office. People can give preferences for the offices, and their preferences are considered based on their seniority. The longer they have been at MathWorks, the higher the seniority. Some offices have windows, some do not, and one window is smaller than others. Additionally, Peter and Tom often work together, so should be in adjacent offices. Marcelo and Rakesh often work together, and should be in adjacent offices.

Office Layout

Offices 1, 2, 3, and 4 are inside offices (no windows). Offices 5, 6, and 7 have windows, but the window in office 5 is smaller than the other two. Here is how the offices are arranged.

```
officelist = {'Office 1','Office 2','Office 3','Office 4','Office 5','Office 6','Office 7'};
printofficeassign(officelist)
```


Office layout: windows are red lines
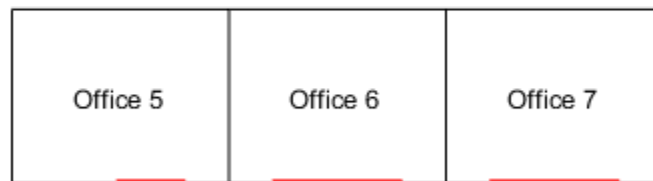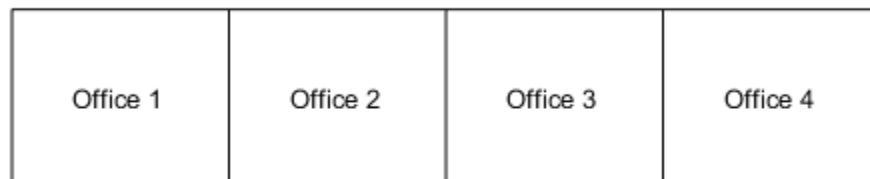
Problem Formulation

You need to formulate the problem mathematically. Create binary variables that indicate whether a person occupies an office. The list of people's names is

```
namelist = {'Mary Ann','Marjorie','Tom','Peter','Marcelo','Rakesh'};
```

Create binary variables indexed by office number and name.

```
occupy = optimvar('occupy',namelist,officelist,...
    'Type','integer','LowerBound',0,'Upperbound',1);
```

Seniority

You want to weight the preferences based on seniority so that the longer you have been at MathWorks, the more your preferences count. The seniority is as follows: Mary Ann 9 years, Marjorie 10 years, Tom 5 years, Peter 3 years, Marcelo 1.5 years, and Rakesh 2 years. Create a normalized weight vector based on seniority.

```
seniority = [9 10 5 3 1.5 2];
weightvector = seniority/sum(seniority);
```

People's Office Preferences

Set up a preference matrix where the rows correspond to offices and the columns correspond to people. Ask each person to give values for each office so that the sum of all their choices, i.e., their column, sums to 100. A higher number means the person prefers the office. Each person's preferences are listed in a column vector.

```
MaryAnn = [0, 0, 0, 0, 10, 40, 50];
Marjorie = [0, 0, 0, 0, 20, 40, 40];
Tom = [0, 0, 0, 0, 30, 40, 30];
Peter = [1, 3, 3, 3, 10, 40, 40];
Marcelo = [3, 4, 1, 2, 10, 40, 40];
Rakesh = [10, 10, 10, 10, 20, 20, 20];
```

The ith element of a person's preference vector is how highly they value the ith office. Thus, the combined preference matrix is as follows.

```
prefmatrix = [MaryAnn;Marjorie;Tom;Peter;Marcelo;Rakesh];
```

Weight the preferences matrix by `weightvector` to scale the columns by seniority.

```
PM = diag(weightvector) * prefmatrix;
```

Objective Function

The objective is to maximize the satisfaction of the preferences weighted by seniority. This is the linear objective function `sum(sum(occupy.*PM))`.

Create an optimization problem and include the objective function.

```
peopleprob = optimproblem('ObjectiveSense','maximize','Objective',sum(sum(occupy.*PM)));
```

Constraints

The first set of constraints requires that each person gets exactly one office, that is for each person, the sum of the `occupy` values corresponding to that person is exactly one.

```
peopleprob.Constraints.constr1 = sum(occupy,2) == 1;
```

The second set of constraints are inequalities. These constraints specify that each office has no more than one person in it.

```
peopleprob.Constraints.constr2 = sum(occupy,1) <= 1;
```

You want Tom and Peter no more than one office away from each other, and the same with Marcelo and Rakesh.

Set constraints that Tom and Peter are not more than 1 away from each other.

```
peopleprob.Constraints.constrpt1 = occupy('Tom','Office 1') + sum(occupy('Peter',:)) - occupy('Pe
peopleprob.Constraints.constrpt2 = occupy('Tom','Office 2') + sum(occupy('Peter',:)) - occupy('Pe
    - occupy('Peter','Office 3') - occupy('Peter','Office 5') <= 1;
peopleprob.Constraints.constrpt3 = occupy('Tom','Office 3') + sum(occupy('Peter',:)) - occupy('Pe
    - occupy('Peter','Office 4') - occupy('Peter','Office 6') <= 1;
peopleprob.Constraints.constrpt4 = occupy('Tom','Office 4') + sum(occupy('Peter',:)) - occupy('Pe
    - occupy('Peter','Office 7') <= 1;
peopleprob.Constraints.constrpt5 = occupy('Tom','Office 5') + sum(occupy('Peter',:)) - occupy('Pe
    - occupy('Peter','Office 6') <= 1;
peopleprob.Constraints.constrpt6 = occupy('Tom','Office 6') + sum(occupy('Peter',:)) - occupy('Pe
    - occupy('Peter','Office 5') - occupy('Peter','Office 7') <= 1;
peopleprob.Constraints.constrpt7 = occupy('Tom','Office 7') + sum(occupy('Peter',:)) - occupy('Pe
    - occupy('Peter','Office 6') <= 1;
```

Now create constraints that Marcelo and Rakesh are not more than 1 away from each other.

```
peopleprob.Constraints.constmr1 = occupy('Marcelo','Office 1') + sum(occupy('Rakesh',:)) - occupy
peopleprob.Constraints.constmr2 = occupy('Marcelo','Office 2') + sum(occupy('Rakesh',:)) - occupy
    - occupy('Rakesh','Office 3') - occupy('Rakesh','Office 5') <= 1;
peopleprob.Constraints.constmr3 = occupy('Marcelo','Office 3') + sum(occupy('Rakesh',:)) - occupy
    - occupy('Rakesh','Office 4') - occupy('Rakesh','Office 6') <= 1;
peopleprob.Constraints.constmr4 = occupy('Marcelo','Office 4') + sum(occupy('Rakesh',:)) - occupy
    - occupy('Rakesh','Office 7') <= 1;
peopleprob.Constraints.constmr5 = occupy('Marcelo','Office 5') + sum(occupy('Rakesh',:)) - occupy
    - occupy('Rakesh','Office 6') <= 1;
peopleprob.Constraints.constmr6 = occupy('Marcelo','Office 6') + sum(occupy('Rakesh',:)) - occupy
    - occupy('Rakesh','Office 5') - occupy('Rakesh','Office 7') <= 1;
peopleprob.Constraints.constmr7 = occupy('Marcelo','Office 7') + sum(occupy('Rakesh',:)) - occupy
    - occupy('Rakesh','Office 6') <= 1;
```

Solve Assignment Problem

Call `solve` to solve the problem.

```
[soln,fval,exitflag,output] = solve(peopleprob);
```

```
LP:               Optimal objective value is -33.836066.


Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap tolerance of the
options.AbsoluteGapTolerance = 0 (the default value). The intcon variables are integer within tol
options.IntegerTolerance = 1e-05 (the default value).
```

View the Solution -- Who Got Each Office?

```
numOffices = length(officelist);
office = cell(numOffices,1);
for i=1:numOffices
    office{i} = find(soln.occupy(:,i)); % people index in office
end

whoinoffice = officelist; % allocate
for i=1:numOffices
    if isempty(office{i})
        whoinoffice{i} = ' empty  ';
    else
        whoinoffice{i} = namelist(office{i});
    end
end

printofficeassign(whoinoffice);
title('Solution of the Office Assignment Problem');
```

Solution of the Office Assignment Problem

empty	Peter	Rakesh	Marcelo

Tom	Marjorie	Mary Ann

Solution Quality

For this problem, the satisfaction of the preferences by seniority is maximized to the value of `fval`. The value of `exitflag` indicates that `solve` converged to an optimal solution. The output structure gives information about the solution process, such as how many nodes were explored, and the gap between the lower and upper bounds in the branching calculation. In this case, no branch-and-bound nodes were generated, meaning the problem was solved without a branch-and-bound step. The

absolute gap is 0, meaning the solution is optimal, with no difference between the internally calculated lower and upper bounds on the objective function.

```
fval,exitflag,output
```

```
fval = 33.8361
```

```
exitflag = 1
```

```
output = struct with fields:
        relativegap: 0
        absolutegap: 0
      numfeaspoints: 1
           numnodes: 0
      constrviolation: 0
            message: 'Optimal solution found.↵Intlinprog stopped at the root node because the ob
             solver: 'intlinprog'
```

See Also

More About

- "Office Assignments by Binary Integer Programming: Solver-Based" on page 9-96
- "Problem-Based Optimization Workflow" on page 10-2

Mixed-Integer Quadratic Programming Portfolio Optimization: Problem-Based

This example shows how to solve a Mixed-Integer Quadratic Programming (MIQP) portfolio optimization problem using the problem-based approach. The idea is to iteratively solve a sequence of mixed-integer linear programming (MILP) problems that locally approximate the MIQP problem. For the solver-based approach, see "Mixed-Integer Quadratic Programming Portfolio Optimization: Solver-Based" on page 9-82.

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 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 $0 \le x(i) \le 1$ 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

$$v(i)f\min \le x(i) \le v(i)f\max.$$

These inequalities both enforce that $x(i)$ and $v(i)$ are zero at exactly the same time, and they also enforce that $f\min \le x(i) \le f\max$ 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 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

$$x^T Q x - z = x_0^T Q x_0 + 2 x_0^T Q \delta - z + O(|\delta|^2) \, .$$

Replacing δ by $x - x_0$ gives

$$x^T Q x - z = -x_0^T Q x_0 + 2 x_0^T Q x - z + O(|x - x_0|^2) \, .$$

For each intermediate solution x_k you introduce a new linear constraint in x and z as the linear part of the expression above:

$$-x_k^T Q x_k + 2 x_k^T Q x - z \leq 0 \, .$$

This has the form $Ax \leq b$, where $A = 2 x_k^T Q$, 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 optimization problems:

- Decide what your variables represent
- Express lower and upper bounds in these variables
- Give linear equality and inequality expressions

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);
```

Create Problem Variables, Constraints, and Objective

Create continuous variables `xvars` representing the asset allocation fraction, binary variables `vvars` representing whether or not the associated `xvars` is zero or strictly positive, and `zvar` representing the *z* variable, a positive scalar.

```
xvars = optimvar('xvars',N,1,'LowerBound',0,'UpperBound',1);
vvars = optimvar('vvars',N,1,'Type','integer','LowerBound',0,'UpperBound',1);
zvar = optimvar('zvar',1,'LowerBound',0);
```

The lower bounds of all the 2N+1 variables in the problem are zero. The upper bounds of the `xvars` and `yvars` variables are one, and `zvar` has no upper bound.

Set the number of assets in the solution to be between 100 and 150. Incorporate this constraint into the problem in the form, namely

$$m \le \sum_i v(i) \le M,$$

by writing two linear constraints:

$$\sum_i v(i) \le M$$

$$\sum_i v(i) \ge m .$$

```
M = 150;
m = 100;
qpprob = optimproblem('ObjectiveSense','maximize');
qpprob.Constraints.mconstr = sum(vvars) <= M;
qpprob.Constraints.mconstr2 = sum(vvars) >= 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 $x(i) \le f\max(i) * v(i)$ and $f\min(i) * v(i) \le x(i)$.

```
qpprob.Constraints.fmaxconstr = xvars <= fmax*vvars;
qpprob.Constraints.fminconstr = fmin*vvars <= xvars;
```

Include the constraint that the portfolio is 100% invested, meaning $\sum x_i = 1$.

```
qpprob.Constraints.allin = sum(xvars) == 1;
```

Set the risk-aversion coefficient λ to `100`.

```matlab
lambda = 100;
```

Define the objective function $r^T x - \lambda z$ and include it in the problem.

```matlab
qpprob.Objective = r'*xvars - lambda*zvar;
```

Solve the Problem

To solve the problem iteratively, begin by solving the problem with the current constraints, which do not yet reflect any linearization.

```matlab
options = optimoptions(@intlinprog,'Display','off'); % Suppress iterative display
[xLinInt,fval,exitFlagInt,output] = solve(qpprob,'options',options);
```

Prepare a stopping condition for the iterations: stop when the slack variable z is within 0.01% of the true quadratic value.

```matlab
thediff = 1e-4;
iter = 1; % iteration counter
assets = xLinInt.xvars;
truequadratic = assets'*Q*assets;
zslack = xLinInt.zvar;
```

Keep a history of the computed true quadratic and slack variables for plotting. Set tighter tolerances than default to help the iterations converge to a correct solution.

```matlab
history = [truequadratic,zslack];

options = optimoptions(options,'LPOptimalityTolerance',1e-10,'RelativeGapTolerance',1e-8,...
                       'ConstraintTolerance',1e-9,'IntegerTolerance',1e-6);
```

Compute the quadratic and slack values. If they differ, then add another linear constraint and solve again.

Each new linear constraint $Ax \le b$ comes from the linear approximation

$$-x_k^T Q x_k + 2x_k^T Q x - z \le 0 \, .$$

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.

```matlab
while abs((zslack - truequadratic)/truequadratic) > thediff % relative error
    constr = 2*assets'*Q*xvars - zvar <= assets'*Q*assets;
    newname = ['iteration',num2str(iter)];
    qpprob.Constraints.(newname) = constr;
    % Solve the problem with the new constraints
    [xLinInt,fval,exitFlagInt,output] = solve(qpprob,'options',options);
    assets = (assets+xLinInt.xvars)/2; % Midway from the previous to the current
%     assets = xLinInt(xvars); % Use the previous line or this one
    truequadratic = xLinInt.xvars'*Q*xLinInt.xvars;
    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)
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.

```
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')
```

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 0.000595107, and the risk-adjusted return is -0.0360382.
```

More elaborate analyses are possible by using features specifically designed for portfolio optimization in Financial Toolbox®. For an example that shows how to use the Portfolio class to directly handle

semicontinuous and cardinality constraints, see "Portfolio Optimization with Semicontinuous and Cardinality Constraints" (Financial Toolbox).

See Also

More About

- "Mixed-Integer Quadratic Programming Portfolio Optimization: Solver-Based" on page 9-82
- "Problem-Based Optimization Workflow" on page 10-2

Cutting Stock Problem: Problem-Based

This example shows how to solve a cutting stock problem using linear programming with an integer linear programming subroutine. The example uses the "Problem-Based Optimization Setup" approach. For the solver-based approach, see "Cutting Stock Problem: Solver-Based" on page 9-103.

Problem Overview

A lumber mill starts with trees that have been trimmed to fixed-length logs. Specify the fixed log length.

```
logLength = 40;
```

The mill then cuts the logs into fixed lengths suitable for further processing. The problem is how to make the cuts so that the mill satisfies a set of orders with the fewest logs.

Specify these fixed lengths and the order quantities for the lengths.

```
lengthlist = [8; 12; 16; 20];
quantity = [90; 111; 55; 30];
nLengths = length(lengthlist);
```

Assume that there is no material loss in making cuts, and no cost for cutting.

Linear Programming Formulation

Several authors, including Ford and Fulkerson [1] and Gilmore and Gomory [2], suggest the following procedure, which you implement in the next section. A cutting pattern is a set of lengths to which a single log can be cut.

Log   Log with cutting pattern

Instead of generating every possible cutting pattern, it is more efficient to generate cutting patterns as the solution of a subproblem. Starting from a base set of cutting patterns, solve the linear programming problem of minimizing the number of logs used subject to the constraint that the cuts, using the existing patterns, satisfy the demands.

After solving that problem, generate a new pattern by solving an integer linear programming subproblem. The subproblem is to find the best new pattern, meaning the number of cuts from each length in `lengthlist` that add up to no more than the total possible length `logLength`. The quantity to optimize is the reduced cost of the new pattern, which is one minus the sum of the Lagrange multipliers for the current solution times the new cutting pattern. If this quantity is negative, then bringing that pattern into the linear program will improve its objective. If not, then no better cutting pattern exists, and the patterns used so far give the optimal linear programming solution. The reason for this conclusion is exactly parallel to the reason for when to stop the primal simplex method: the method terminates when there is no variable with a negative reduced cost. The problem in this example terminates when there is no pattern with negative reduced cost. For details and an example, see Column generation algorithms and its references.

After solving the linear programming problem in this way, you can have noninteger solutions. Therefore, solve the problem once more, using the generated patterns and constraining the variables to have integer values.

MATLAB Problem-Based Formulation

A pattern, in this formulation, is a vector of integers containing the number of cuts of each length in `lengthlist`. Arrange a matrix named `patterns` to store the patterns, where each column in the matrix gives a pattern. For example,

$$\text{patterns} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

The first pattern (column) represents two cuts of length 8 and one cut of length 20. The second pattern represents two cuts of length 12 and one cut of length 16. Each is a feasible pattern because the total of the cuts is no more than `logLength = 40`.

In this formulation, if `x` is a column vector of integers containing the number of times each pattern is used, then `patterns*x` is a column vector giving the number of cuts of each type. The constraint of meeting demand is `patterns*x >= quantity`. For example, using the previous `patterns` matrix, suppose that $x = \begin{bmatrix} 45 \\ 56 \end{bmatrix}$. (This x uses 101 logs.) Then

$$\text{patterns}*x = \begin{bmatrix} 90 \\ 112 \\ 56 \\ 45 \end{bmatrix},$$

which represents a feasible solution because the result exceeds the demand

$$\text{quantity} = \begin{bmatrix} 90 \\ 111 \\ 55 \\ 30 \end{bmatrix}.$$

To have an initial feasible cutting pattern, use the simplest patterns, which have just one length of cut. Use as many cuts of that length as feasible for the log.

```
patterns = diag(floor(logLength./lengthlist));
nPatterns = size(patterns,2);
```

To generate new patterns from the existing ones based on the current Lagrange multipliers, solve a subproblem. Call the subproblem in a loop to generate patterns until no further improvement is found. The subproblem objective depends only on the current Lagrange multipliers. The variables are nonnegative integers representing the number of cuts of each length. The only constraint is that the sum of the lengths of the cuts in a pattern is no more than the log length.

```
subproblem = optimproblem();
cuts = optimvar('cuts', nLengths, 1, 'Type','integer','LowerBound',zeros(nLengths,1));
subproblem.Constraints = dot(lengthlist,cuts) <= logLength;
```

To avoid unnecessary feedback from the solvers, set the `Display` option to `'off'` for both the outer loop and the inner subproblem solution.

```matlab
lpopts = optimoptions('linprog','Display','off');
ipopts = optimoptions('intlinprog',lpopts);
```

Initialize the variables for the loop.

```matlab
reducedCost = -inf;
reducedCostTolerance = -0.0001;
exitflag = 1;
```

Call the loop.

```matlab
while reducedCost < reducedCostTolerance && exitflag > 0
    logprob = optimproblem('Description','Cut Logs');
    % Create variables representing the number of each pattern used
    x = optimvar('x', nPatterns, 1, 'LowerBound', 0);
    % The objective is the number of logs used
    logprob.Objective.logsUsed = sum(x);
    % The constraint is that the cuts satisfy the demand
    logprob.Constraints.Demand = patterns*x >= quantity;

    [values,nLogs,exitflag,~,lambda] = solve(logprob,'options',lpopts);

    if exitflag > 0
        fprintf('Using %g logs\n',nLogs);
        % Now generate a new pattern, if possible
        subproblem.Objective = 1.0 - dot(lambda.Constraints.Demand,cuts);
        [values,reducedCost,pexitflag] = solve(subproblem,'options',ipopts);
        newpattern = round(values.cuts);
        if double(pexitflag) > 0 && reducedCost < reducedCostTolerance
            patterns = [patterns newpattern];
            nPatterns = nPatterns + 1;
        end
    end
end
```

```
Using 97.5 logs
Using 92 logs
Using 89.9167 logs
Using 88.3 logs
```

You now have the solution of the linear programming problem. To complete the solution, solve the problem again with the final patterns, changing the solution variable x to the integer type. Also, compute the waste, which is the quantity of unused logs (in feet) for each pattern and for the problem as a whole.

```matlab
if exitflag <= 0
    disp('Error in column generation phase')
else
    x.Type = 'integer';
    [values,logsUsed,exitflag] = solve(logprob,'options',ipopts);
    if double(exitflag) > 0
        values.x = round(values.x); % in case some values were not exactly integers
        logsUsed = sum(values.x);
        fprintf('Optimal solution uses %g logs\n', logsUsed);
        totalwaste = sum((patterns*values.x - quantity).*lengthlist); % waste due to overproduct
        for j = 1:size(values.x)
            if values.x(j) > 0
                fprintf('Cut %g logs with pattern\n',values.x(j));
```

```
                    for w = 1:size(patterns,1)
                        if patterns(w,j) > 0
                            fprintf('    %g cut(s) of length %d\n', patterns(w,j),lengthlist(w));
                        end
                    end
                    wastej = logLength - dot(patterns(:,j),lengthlist); % waste due to pattern ineff
                    totalwaste = totalwaste + wastej;
                fprintf('    Waste of this pattern is %g\n',wastej);
                    end
                end
            fprintf('Total waste in this problem is %g.\n',totalwaste);
        else
            disp('Error in final optimization')
        end
    end
end
```

```
Optimal solution uses 89 logs

Cut 15 logs with pattern

    2 cut(s) of length 20

    Waste of this pattern is 0

Cut 18 logs with pattern

    1 cut(s) of length 8
    2 cut(s) of length 16

    Waste of this pattern is 0

Cut 37 logs with pattern

    2 cut(s) of length 8
    2 cut(s) of length 12

    Waste of this pattern is 0

Cut 19 logs with pattern

    2 cut(s) of length 12
    1 cut(s) of length 16

    Waste of this pattern is 0

Total waste in this problem is 28.
```

Part of the waste is due to overproduction, because the mill cuts one log into three 12-foot pieces, but uses only one. Part of the waste is due to pattern inefficiency, because the three 12-foot pieces are 4 feet short of the total length of 40 feet.

References

[1] Ford, L. R., Jr. and D. R. Fulkerson. *A Suggested Computation for Maximal Multi-Commodity Network Flows.* Management Science 5, 1958, pp. 97-101.

[2] Gilmore, P. C., and R. E. Gomory. *A Linear Programming Approach to the Cutting Stock Problem--Part II.* Operations Research 11, No. 6, 1963, pp. 863-888.

See Also

More About

- "Cutting Stock Problem: Solver-Based" on page 9-103
- "Problem-Based Optimization Workflow" on page 10-2

Solve Sudoku Puzzles Via Integer Programming: Problem-Based

This example shows how to solve a Sudoku puzzle using binary integer programming. For the solver-based approach, see "Solve Sudoku Puzzles Via Integer Programming: Solver-Based" on page 9-89.

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 MATLAB 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, where each layer corresponds to an integer from 1 through 9. 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 a constant term 0. The problem is really just to find a feasible solution, meaning one that satisfies all the constraints. However, for tie breaking 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 $x(i, j, 1), \ldots, x(i, j, 9)$. 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:

$$\sum_{i=1}^{3} \sum_{j=1}^{3} x(i + U, j + V, k) = 1, \text{ where } U, V \in \{0, 3, 6\} \,.$$

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 $k \neq m$.

Sudoku in Optimization Problem Form

Create an optimization variable x that is binary and of size 9-by-9-by-9.

```
x = optimvar('x',9,9,9,'Type','integer','LowerBound',0,'UpperBound',1);
```

Create an optimization problem with a rather arbitrary objective function. The objective function can help the solver by destroying the inherent symmetry of the problem.

```
sudpuzzle = optimproblem;
mul = ones(1,1,9);
mul = cumsum(mul,3);
sudpuzzle.Objective = sum(sum(sum(x,1),2).*mul);
```

Represent the constraints that the sums of x in each coordinate direction are one.

```
sudpuzzle.Constraints.consx = sum(x,1) == 1;
sudpuzzle.Constraints.consy = sum(x,2) == 1;
sudpuzzle.Constraints.consz = sum(x,3) == 1;
```

Create the constraints that the sums of the major grids sum to one as well.

```
majorg = optimconstr(3,3,9);

for u = 1:3
    for v = 1:3
        arr = x(3*(u-1)+1:3*(u-1)+3,3*(v-1)+1:3*(v-1)+3,:);
        majorg(u,v,:) = sum(sum(arr,1),2) == ones(1,1,9);
    end
end
sudpuzzle.Constraints.majorg = majorg;
```

Include the initial clues by setting lower bounds of 1 at the clue entries. This setting fixes the value of the corresponding entry to be 1, and so sets the solution at each clued value to be the clue entry.

```
for u = 1:size(B,1)
    x.LowerBound(B(u,1),B(u,2),B(u,3)) = 1;
end
```

Solve the Sudoku puzzle.

```
sudsoln = solve(sudpuzzle)
```

```
Solving problem using intlinprog.
LP:                Optimal objective value is 405.000000.


Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap
tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).

sudsoln = struct with fields:
    x: [9x9x9 double]
```

Round the solution to ensure that all entries are integers, and display the solution.

```
sudsoln.x = round(sudsoln.x);

y = ones(size(sudsoln.x));
for k = 2:9
    y(:,:,k) = k; % multiplier for each depth k
end
S = sudsoln.x.*y; % multiply each entry by its depth
S = sum(S,3); % S is 9-by-9 and holds the solved puzzle
drawSudoku(S)
```

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
```

See Also

More About

- "Solve Sudoku Puzzles Via Integer Programming: Solver-Based" on page 9-89
- "Problem-Based Optimization Workflow" on page 10-2

Minimize Makespan in Parallel Processing

This example involves a set of tasks to be processed in parallel. Each task has a known processing time. The makespan is the amount of time to process all of the tasks. This figure shows two processors; the height of each colored box represents the length of time to process a task. Each task can have a different run time on each processor.
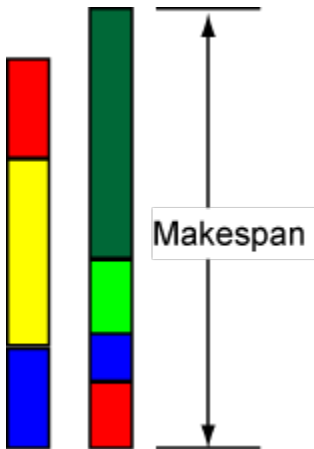
Your goal is to schedule tasks on processors so as to minimize the makespan.

Problem Setup

This example has 11 processors and 40 tasks. The time for each processor to process each task is given in the array `processingTime`. For this example, generate random processing times.

```
rng default % for reproducibility
numberOfProcessors = 11;
numberOfTasks = 40;
processingTime = [10;7;2;5;3;4;7;6;4;3;1] .* rand(numberOfProcessors,numberOfTasks);
```

`processingTime(i,j)` represents the amount of time that processor `i` takes to process task `j`.

To solve the problem using binary integer programming, create `process` as a binary optimization variable array, where `process(i,j) = 1` means processor `i` processes task `j`.

```
process = optimvar('process',size(processingTime),'Type','integer','LowerBound',0,'UpperBound',1)
```

Each task must be assigned to exactly one processor.

```
assigneachtask = sum(process,1) == 1;
```

To represent the objective, define a nonnegative optimization variable named `makespan`.

```
makespan = optimvar('makespan','LowerBound',0);
```

Compute the time that each processor requires to process its tasks.

```
computetime = sum(process.*processingTime,2);
```

Relate the compute times to the makespan. The makespan is greater than or equal to each compute time.

```
makespanbound = makespan >= computetime;
```

Create an optimization problem whose objective is to minimize the makespan, and include the problem constraints.

```
scheduleproblem = optimproblem('Objective',makespan);
scheduleproblem.Constraints.assigneachtask = assigneachtask;
scheduleproblem.Constraints.makespanbound = makespanbound;
```

Solve Problem and View Solution

Solve the problem, suppressing the usual display.

```
options = optimoptions(scheduleproblem,'Display',"off");
[sol,fval,exitflag] = solve(scheduleproblem,'Options',options)
```

```
sol = struct with fields:
    makespan: 1.3634
     process: [11x40 double]


fval = 1.3634

exitflag =
    OptimalSolution
```

The returned `exitflag` indicates that the solver found an optimal solution, meaning the returned solution has minimal makespan.

The returned makespan is 1.3634. Is this an efficient schedule? To find out, view the resulting schedule as a stacked bar chart. First, create a schedule matrix where row `i` represents the tasks done by processor `i`. Then, find the processing time for each entry in the schedule.

```
processval = round(sol.process);
maxlen = max(sum(processval,2)); % Required width of the matrix
% Now calculate the schedule matrix
optimalSchedule = zeros(numberOfProcessors,maxlen);
ptime = optimalSchedule;
for i = 1:numberOfProcessors
    schedi = find(processval(i,:));
    optimalSchedule(i,1:length(schedi)) = schedi;
    ptime(i,1:length(schedi)) = processingTime(i,schedi);
end
optimalSchedule
```

optimalSchedule = *11×10*

25	38	0	0	0	0	0	0	0	0
4	12	23	32	0	0	0	0	0	0
7	13	14	18	31	37	0	0	0	0
35	0	0	0	0	0	0	0	0	0
6	22	39	0	0	0	0	0	0	0
10	26	28	30	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0
21	24	27	0	0	0	0	0	0	0
8	16	33	0	0	0	0	0	0	0
3	17	34	0	0	0	0	0	0	0

9-155

⋮

Display the schedule matrix as a stacked bar chart. Label the top of each bar with the task number.

```
figure
bar(ptime,'stacked')
xlabel('Processor Number')
ylabel('Processing Time')
title('Task Assignments to Processors')
for i=1:size(optimalSchedule,1)
    for j=1:size(optimalSchedule,2)
        if optimalSchedule(i,j) > 0
            processText = num2str(optimalSchedule(i,j),"%d");
            hText = text(i,sum(ptime(i,1:j),2),processText);
            set(hText,"VerticalAlignment","top","HorizontalAlignment","center","FontSize",10,"Co
        end
    end
end
```
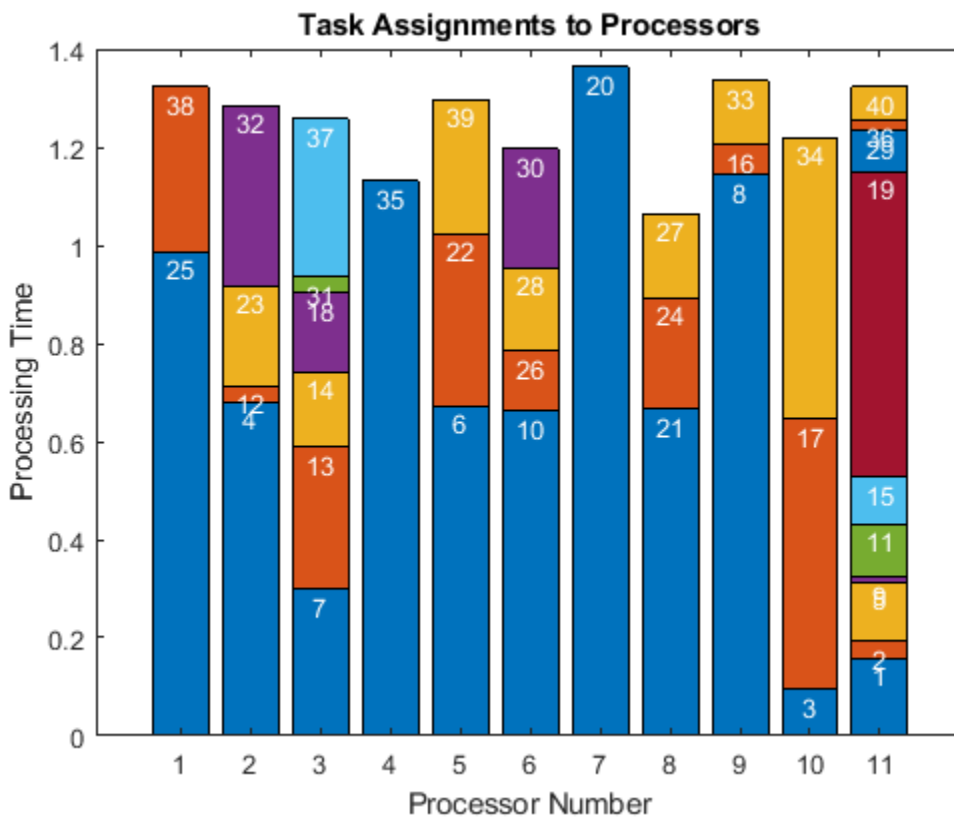

Find the minimum height of the stacked bars, which represents the earliest time a processor stops working. Then, find the processor corresponding to the maximum height.

```
minlength = min(sum(ptime,2))
```

```
minlength = 1.0652
```

```
[~,processormaxlength] = max(sum(ptime,2))
```

```
processormaxlength = 7
```

All processors are busy until time `minlength` = 1.0652. From the stacked bar chart, you can see that processor 8 stops working at that time. Processor `processormaxlength` = 7 is the last processor to stop working, which occurs at time `makespan` = 1.3634.

See Also
`solve`

More About
- "Problem-Based Optimization Workflow" on page 10-2
- "Linear Programming and Mixed-Integer Linear Programming"

Investigate Linear Infeasibilities

This example shows how to investigate the linear constraints that cause a problem to be infeasible. For further details about these techniques, see Chinneck [1] on page 9-0   and [2] on page 9-0   .

If linear constraints cause a problem to be infeasible, you might want to find a subset of the constraints that is infeasible, but removing any member of the subset makes the rest of the subset feasible. The name for such a subset is *Irreducible Infeasible Subset of Constraints,* abbreviated IIS. Conversely, you might want to find a maximum cardinality subset of constraints that is feasible. This subset is called a *Maximum Feasible Subset,* abbreviated MaxFS. The two concepts are related, but not identical. A problem can have many different IISs, some with different cardinality.

This example shows two ways of finding an IIS, and two ways of obtaining a feasible set of constraints. The example assumes that all given bounds are correct, meaning the lb and ub arguments have no errors.

Infeasible Example

Create a random matrix A representing linear inequalities of size 150-by-15. Set the corresponding vector b to a vector with entries of 10, and change 5% of those values to –10.

```
N = 15;
rng default
A = randn([10*N,N]);
b = 10*ones(size(A,1),1);
Aeq = [];
beq = [];
b(rand(size(b)) <= 0.05) = -10;
f = ones(N,1);
lb = -f;
ub = f;
```

Check that problem is infeasible.

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

No feasible solution found.

Linprog stopped because no point satisfies the constraints.

Deletion Filter

To identify an IIS, perform the following steps. Given a set of linear constraints numbered 1 through N, where all problem constraints are infeasible:

For each i from 1 to N:

- Temporarily remove constraint i from the problem.
- Test the resulting problem for feasibility.
- If the problem is feasible without constraint i, return constraint i to the problem.
- If the problem is not feasible without constraint i, do not return constraint i to the problem

Continue to the next i (up to value N).

At the end of this procedure, the constraints that remain in the problem form an IIS.

For MATLAB® code that implements this procedure, see the `deletionfilter` helper function at the end of this example on page 9-0   .

Note: If you use the live script file for this example, the `deletionfilter` function is already included at the end of the file. Otherwise, you need to create this function at the end of your .m file or add it as a file on the MATLAB path. The same is true for the other helper functions used later in this example.

See the effect of `deletionfilter` on the example data.

```
[ineqs,eqs,ncall] = deletionfilter(A,b,Aeq,beq,lb,ub);
```

The problem has no equality constraints. Find the indices for the inequality constraints and the value of `b(iis)`.

```
iis = find(ineqs)
```

```
iis = 114
```

```
b(iis)
```

```
ans = -10
```

Only one inequality constraint causes the problem to be infeasible, along with the bound constraints. The constraint is

`A(iis,:)*x <= b(iis)`.

Why is this constraint infeasible together with the bounds? Find the sum of the absolute values of that row of `A`.

```
disp(sum(abs(A(iis,:))))
```

```
    8.4864
```

Due to the bounds, the `x` vector has values between –1 and 1, and so `A(iis,:)*x` cannot be less than `b(iis)` = –10.

How many `linprog` calls did `deletionfilter` perform?

```
disp(ncall)
```

```
   150
```

The problem has 150 linear constraints, so the function called `linprog` 150 times.

Elastic Filter

As an alternative to the deletion filter, which examines every constraint, try the elastic filter. This filter works as follows.

First, allow each constraint `i` to be violated by a positive amount `e(i)`, where equality constraints have both additive and subtractive positive elastic values.

$$A_{\mathrm{ineq}}x \le b_{\mathrm{ineq}} + e$$
$$A_{\mathrm{eq}}x = b_{\mathrm{eq}} + e_1 - e_2$$

Next, solve the associated linear programming problem (LP)

$$\min_{x,\,e} \sum e_i$$

subject to the listed constraints and with $e_i \geq 0$.

- If the associated LP has a solution, remove all constraints that have a strictly positive associated e_i, and record those constraints in a list of indices (potential IIS members). Return to the previous step to solve the new, reduced associated LP.

- If the associated LP has no solution (is infeasible) or has no strictly positive associated e_i, exit the filter.

The elastic filter can exit in many fewer iterations than the deletion filter, because it can bring many indices at once into the IIS, and can halt without going through the entire list of indices. However, the problem has more variables than the original problem, and its resulting list of indices can be larger than an IIS. To find an IIS after running an elastic filter, run the deletion filter on the result.

For MATLAB® code that implements this filter, see the `elasticfilter` helper function at the end of this example on page 9-0 .

See the effect of `elasticfilter` on the example data.

```
[ineqselastic,eqselastic,ncall] = ...
    elasticfilter(A,b,Aeq,beq,lb,ub);
```

The problem has no equality constraints. Find the indices for the inequality constraints.

```
iiselastic = find(ineqselastic)
```

```
iiselastic = 5×1

     2
    60
    82
    97
   114
```

The elastic IIS lists five constraints, whereas the deletion filter found only one. Run the deletion filter on the returned set to find a genuine IIS.

```
A_1 = A(ineqselastic > 0,:);
b_1 = b(ineqselastic > 0);
[dineq_iis,deq_iis,ncall2] = deletionfilter(A_1,b_1,Aeq,beq,lb,ub);
iiselasticdeletion = find(dineq_iis)
```

```
iiselasticdeletion = 5
```

The fifth constraint in the elastic filter result, inequality 114, is the IIS. This result agrees with the answer from the deletion filter. The difference between the approaches is that the combined elastic and deletion filter approach uses many fewer `linprog` calls. Display the total number of `linprog` calls used by the elastic filter followed by the deletion filter.

```
disp(ncall + ncall2)
```

```
     7
```

Remove IIS in a Loop

Generally, obtaining a single IIS does not enable you to find all the reasons that your optimization problem fails. To correct an infeasible problem, you can repeatedly find an IIS and remove it from the problem until the problem becomes feasible.

The following code shows how to remove one IIS at a time from a problem until the problem becomes feasible. The code uses an indexing technique to keep track of constraints in terms of their positions in the original problem, before the algorithm removes any constraints.

The code keeps track of the original variables in the problem by using a Boolean vector `activeA` to represent the current constraints (rows) of the `A` matrix, and a Boolean vector `activeAeq` to represent the current constraints of the `Aeq` matrix. When adding or removing constraints, the code indexes into `A` or `Aeq` so that the row numbers do not change, even though the number of constraints changes.

Running this code returns `idx2`, a vector of the indices of the nonzero elements in `activeA`:

```
idx2 = find(activeA)
```

Suppose that `var` is a Boolean vector that has the same length as `idx2`. Then

```
idx2(find(var))
```

expresses `var` as indices into the original problem variables. In this way, the indexing can take a subset of a subset of constraints, work with only the smaller subset, and still unambiguously refer to the original problem variables.

```
opts = optimoptions('linprog','Display',"none");
activeA = true(size(b));
activeAeq = true(size(beq));
[~,~,exitflag] = linprog(f,A,b,Aeq,beq,lb,ub,opts);
ncl = 1;
while exitflag < 0
    [ineqselastic,eqselastic,ncall] = ...
        elasticfilter(A(activeA,:),b(activeA),Aeq(activeAeq,:),beq(activeAeq),lb,ub);
    ncl = ncl + ncall;
    idxaa = find(activeA);
    idxae = find(activeAeq);
    tmpa = idxaa(find(ineqselastic));
    tmpae = idxae(find(eqselastic));
    AA = A(tmpa,:);
    bb = b(tmpa);
    AE = Aeq(tmpae,:);
    be = beq(tmpae);
    [ineqs,eqs,ncall] = ...
        deletionfilter(AA,bb,AE,be,lb,ub);
    ncl = ncl + ncall;
    activeA(tmpa(ineqs)) = false;
    activeAeq(tmpae(eqs)) = false;
    disp(['Removed inequalities ',int2str((tmpa(ineqs))'),' and equalities ',int2str((tmpae(eqs)
    [~,~,exitflag] = ...
        linprog(f,A(activeA,:),b(activeA),Aeq(activeAeq,:),beq(activeAeq),lb,ub,opts);
    ncl = ncl + 1;
end
```

```
Removed inequalities 114 and equalities
Removed inequalities 97 and equalities
```

```
Removed inequalities 64  82 and equalities
Removed inequalities 60 and equalities
```

```
fprintf('Number of linprog calls: %d\n',ncl)
```

```
Number of linprog calls: 28
```

Notice that the loop removes inequalities 64 and 82 simultaneously, which indicates that these two constraints form an IIS.

Find MaxFS

Another approach for obtaining a feasible set of constraints is to find a MaxFS directly. As Chinneck [1] explains, finding a MaxFS is an NP-complete problem, meaning the problem does not necessarily have efficient algorithms for finding a MaxFS. However, Chinneck proposes some algorithms that can work efficiently.

Use Chinneck's Algorithm 7.3 to find a *cover set* of constraints that, when removed, gives a feasible set. The algorithm is implemented in the `generatecover` helper function at the end of this example on page 9-0 .

```
[coversetineq,coverseteq,nlp] = generatecover(A,b,Aeq,beq,lb,ub)
```

```
coversetineq = 5×1

   114
    97
    60
    82
     2
```

```
coverseteq =

     []
```

```
nlp = 40
```

Remove these constraints and solve the LP.

```
usemeineq = true(size(b));
usemeineq(coversetineq) = false; % Remove inequality constraints
usemeeq = true(size(beq));
usemeeq(coverseteq) = false; % Remove equality constraints
[xs,fvals,exitflags] = ...
    linprog(f,A(usemeineq,:),b(usemeineq),Aeq(usemeeq),beq(usemeeq),lb,ub);
```

```
Optimal solution found.
```

Notice that the cover set is exactly the same as the `iiselastic` set from Elastic Filter on page 9-0 . In general, the elastic filter finds too large a cover set. Chinneck's Algorithm 7.3 starts with the elastic filter result and then retains only the constraints that are necessary.

Chinneck's Algorithm 7.3 takes 40 calls to `linprog` to complete the calculation of a MaxFS. This number is a bit more than 28 calls used earlier in the process of deleting IIS in a loop.

Also, notice that the inequalities removed in the loop are not exactly the same as the inequalities removed by Algorithm 7.3. The loop removes inequalities 114, 97, 82, 60, and **64**, while Algorithm 7.3

removes inequalities 114, 97, 82, 60, and **2**. Check that inequalities 82 and 64 form an IIS (as indicated in Remove IIS in a Loop on page 9-0   ), and that inequalities 82 and 2 also form an IIS.

```
usemeineq = false(size(b));
usemeineq([82,64]) = true;
ineqs = deletionfilter(A(usemeineq,:),b(usemeineq),Aeq,beq,lb,ub);
disp(ineqs)

    1
    1

usemeineq = false(size(b));
usemeineq([82,2]) = true;
ineqs = deletionfilter(A(usemeineq,:),b(usemeineq),Aeq,beq,lb,ub);
disp(ineqs)

    1
    1
```

References

[1] Chinneck, J. W. *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods.* Springer, 2008.

[2] Chinneck, J. W. "Feasibility and Infeasibility in Optimization." Tutorial for CP-AI-OR-07, Brussels, Belgium. Available at `https://www.sce.carleton.ca/faculty/chinneck/docs/CPAIOR07InfeasibilityTutorial.pdf`.

Helper Functions

This code creates the `deletionfilter` helper function.

```
function [ineq_iis,eq_iis,ncalls] = deletionfilter(Aineq,bineq,Aeq,beq,lb,ub)
ncalls = 0;
[mi,n] = size(Aineq); % Number of variables and linear inequality constraints
f = zeros(1,n);
me = size(Aeq,1); % Number of linear equality constraints
opts = optimoptions("linprog","Algorithm","dual-simplex","Display","none");

ineq_iis = true(mi,1); % Start with all inequalities in the problem
eq_iis = true(me,1); % Start with all equalities in the problem

for i=1:mi
    ineq_iis(i) = 0; % Remove inequality i
    [~,~,exitflag] = linprog(f,Aineq(ineq_iis,:),bineq(ineq_iis),...
                             Aeq,beq,lb,ub,[],opts);
    ncalls = ncalls + 1;
    if exitflag == 1 % If now feasible
        ineq_iis(i) = 1; % Return i to the problem
    end
end
for i=1:me
    eq_iis(i) = 0; % Remove equality i
    [~,~,exitflag] = linprog(f,Aineq,bineq,...
                             Aeq(eq_iis,:),beq(eq_iis),lb,ub,[],opts);
    ncalls = ncalls + 1;
    if exitflag == 1 % If now feasible
        eq_iis(i) = 1; % Return i to the problem
```

This code creates the `elasticfilter` helper function.

```
function [ineq_iis,eq_iis,ncalls,fval0] = elasticfilter(Aineq,bineq,Aeq,beq,lb,ub)
ncalls = 0;
[mi,n] = size(Aineq); % Number of variables and linear inequality constraints
me = size(Aeq,1);
Aineq_r = [Aineq -1.0*eye(mi) zeros(mi,2*me)];
Aeq_r = [Aeq zeros(me,mi) eye(me) -1.0*eye(me)]; % Two slacks for each equality constraint
lb_r = [lb(:); zeros(mi+2*me,1)];
ub_r = [ub(:); inf(mi+2*me,1)];
ineq_slack_offset = n;
eq_pos_slack_offset = n + mi;
eq_neg_slack_offset = n + mi + me;
f = [zeros(1,n) ones(1,mi+2*me)];
opts = optimoptions("linprog","Algorithm","dual-simplex","Display","none");
tol = 1e-10;

ineq_iis = false(mi,1);
eq_iis = false(me,1);
[x,fval,exitflag] = linprog(f,Aineq_r,bineq,Aeq_r,beq,lb_r,ub_r,[],opts);
fval0 = fval;
ncalls = ncalls + 1;
while exitflag == 1 && fval > tol % Feasible and some slacks are nonzero
    c = 0;
    for i = 1:mi
        j = ineq_slack_offset+i;
        if x(j) > tol
            ub_r(j) = 0.0;
            ineq_iis(i) = true;
            c = c+1;
        end
    end
    for i = 1:me
        j = eq_pos_slack_offset+i;
        if x(j) > tol
            ub_r(j) = 0.0;
            eq_iis(i) = true;
            c = c+1;
        end
    end
    for i = 1:me
        j = eq_neg_slack_offset+i;
        if x(j) > tol
            ub_r(j) = 0.0;
            eq_iis(i) = true;
            c = c+1;
        end
    end
    [x,fval,exitflag] = linprog(f,Aineq_r,bineq,Aeq_r,beq,lb_r,ub_r,[],opts);
    if fval > 0
        fval0 = fval;
    end
    ncalls = ncalls + 1;
```

```
        end
    end
```

This code creates the `generatecover` helper function. The code uses the same indexing technique for keeping track of constraints as the Remove IIS in a Loop on page 9-0   code.

```
function [coversetineq,coverseteq,nlp] = generatecover(Aineq,bineq,Aeq,beq,lb,ub)
% Returns the cover set of linear inequalities, the cover set of linear
% equalities, and the total number of calls to linprog.
% Adapted from Chinneck [1] Algorithm 7.3. Step numbers are from this book.
coversetineq = [];
coverseteq = [];
activeA = true(size(bineq));
activeAeq = true(size(beq));
% Step 1 of Algorithm 7.3
[ineq_iis,eq_iis,ncalls] = elasticfilter(Aineq,bineq,Aeq,beq,lb,ub);
nlp = ncalls;
ninf = sum(ineq_iis(:)) + sum(eq_iis(:));
if ninf == 1
    coversetineq = ineq_iis;
    coverseteq = eq_iis;
    return
end
holdsetineq = find(ineq_iis);
holdseteq = find(eq_iis);
candidateineq = holdsetineq;
candidateeq = holdseteq;
% Step 2 of Algorithm 7.3
while sum(candidateineq(:)) + sum(candidateeq(:)) > 0
    minsinf = inf;
    ineqflag = 0;
    for i = 1:length(candidateineq(:))
        activeA(candidateineq(i)) = false;
        idx2 = find(activeA);
        idx2eq = find(activeAeq);
        [ineq_iis,eq_iis,ncalls,fval] = elasticfilter(Aineq(activeA,:),bineq(activeA),Aeq(activeA
        nlp = nlp + ncalls;
        ineq_iis = idx2(find(ineq_iis));
        eq_iis = idx2eq(find(eq_iis));
        if fval == 0
            coversetineq = [coversetineq;candidateineq(i)];
            return
        end
        if fval < minsinf
            ineqflag = 1;
            winner = candidateineq(i);
            minsinf = fval;
            holdsetineq = ineq_iis;
            if numel(ineq_iis(:)) + numel(eq_iis(:)) == 1
                nextwinner = ineq_iis;
                nextwinner2 = eq_iis;
                nextwinner = [nextwinnner,nextwinner2];
            else
                nextwinner = [];
            end
        end
        activeA(candidateineq(i)) = true;
    end
```

```
    for i = 1:length(candidateeq(:))
        activeAeq(candidateeq(i)) = false;
        idx2 = find(activeA);
        idx2eq = find(activeAeq);
        [ineq_iis,eq_iis,ncalls,fval] = elasticfilter(Aineq(activeA),bineq(activeA),Aeq(activeAec
        nlp = nlp + ncalls;
        ineq_iis = idx2(find(ineq_iis));
        eq_iis = idx2eq(find(eq_iis));
        if fval == 0
            coverseteq = [coverseteq;candidateeq(i)];
            return
        end
        if fval < minsinf
            ineqflag = -1;
            winner = candidateeq(i);
            minsinf = fval;
            holdseteq = eq_iis;
            if numel(ineq_iis(:)) + numel(eq_iis(:)) == 1
                nextwinner = ineq_iis;
                nextwinner2 = eq_iis;
                nextwinner = [nextwinnner,nextwinner2];
            else
                nextwinner = [];
            end
        end
        activeAeq(candidateeq(i)) = true;
    end
% Step 3 of Algorithm 7.3
    if ineqflag == 1
        coversetineq = [coversetineq;winner];
        activeA(winner) = false;
        if nextwinner
            coversetineq = [coversetineq;nextwinner];
            return
        end
    end
    if ineqflag == -1
        coverseteq = [coverseteq;winner];
        activeAeq(winner) = false;
        if nextwinner
            coverseteq = [coverseteq;nextwinner];
            return
        end
    end
    candidateineq = holdsetineq;
    candidateeq = holdseteq;
end
end
```

See Also

```
linprog
```

More About

- "Solve Nonlinear Feasibility Problem, Problem-Based" on page 7-43
- "Converged to an Infeasible Point" on page 4-6

- "Solve Feasibility Problem" (Global Optimization Toolbox)

10

Problem-Based Optimization

Problem-Based Optimization Workflow

> **Note** Optimization Toolbox provides two approaches for solving single-objective optimization problems. This topic describes the problem-based approach. "Solver-Based Optimization Problem Setup" describes the solver-based approach.

To solve an optimization problem, perform the following steps.

- Create an optimization problem object by using `optimproblem`. A problem object is a container in which you define an objective expression and constraints. The optimization problem object defines the problem and any bounds that exist in the problem variables.

  For example, create a maximization problem.

  ```
  prob = optimproblem('ObjectiveSense','maximize');
  ```

- Create named variables by using `optimvar`. An optimization variable is a symbolic variable that you use to describe the problem objective and constraints. Include any bounds in the variable definitions.

  For example, create a 15-by-3 array of binary variables named `'x'`.

  ```
  x = optimvar('x',15,3,'Type','integer','LowerBound',0,'UpperBound',1);
  ```

- Define the objective function in the problem object as an expression in the named variables.

  > **Note** If you have a nonlinear function that is not composed of polynomials, rational expressions, and elementary functions such as `exp`, then convert the function to an optimization expression by using `fcn2optimexpr`. See "Convert Nonlinear Function to Optimization Expression" on page 7-8 and "Supported Operations on Optimization Variables and Expressions" on page 10-43.

  If necessary, include extra parameters in your expression as workspace variables; see "Pass Extra Parameters in Problem-Based Approach" on page 10-11.

  For example, assume that you have a real matrix `f` of the same size as a matrix of variables `x`, and the objective is the sum of the entries in `f` times the corresponding variables `x`.

  ```
  prob.Objective = sum(sum(f.*x));
  ```

- Define constraints for optimization problems as either comparisons in the named variables or as comparisons of expressions.

  > **Note** If you have a nonlinear function that is not composed of polynomials, rational expressions, and elementary functions such as `exp`, then convert the function to an optimization expression by using `fcn2optimexpr`. See "Convert Nonlinear Function to Optimization Expression" on page 7-8 and "Supported Operations on Optimization Variables and Expressions" on page 10-43.

  For example, assume that the sum of the variables in each row of `x` must be one, and the sum of the variables in each column must be no more than one.

  ```
  onesum = sum(x,2) == 1;
  vertsum = sum(x,1) <= 1;
  ```

```
prob.Constraints.onesum = onesum;
prob.Constraints.vertsum = vertsum;
```

- For nonlinear problems, set an initial point as a structure whose fields are the optimization variable names. For example:

```
x0.x = randn(size(x));
x0.y = eye(4); % Assumes y is a 4-by-4 variable
```

- Solve the problem by using `solve`.

```
sol = solve(prob);
% Or, for nonlinear problems,
sol = solve(prob,x0)
```

In addition to these basic steps, you can review the problem definition before solving the problem by using `show` or `write`. Set options for `solve` by using `optimoptions`, as explained in "Change Default Solver or Options" on page 10-14.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

For a basic mixed-integer linear programming example, see "Mixed-Integer Linear Programming Basics: Problem-Based" on page 10-47 or the video version Solve a Mixed-Integer Linear Programming Problem Using Optimization Modeling. For a nonlinear example, see "Solve a Constrained Nonlinear Problem, Problem-Based" on page 1-5. For more extensive examples, see "Problem-Based Nonlinear Optimization", "Linear Programming and Mixed-Integer Linear Programming", or "Quadratic Programming and Cone Programming".

See Also

fcn2optimexpr | optimoptions | optimproblem | optimvar | show | solve | write

More About

- "Mixed-Integer Linear Programming Basics: Problem-Based" on page 10-47
- Solve a Mixed-Integer Linear Programming Problem Using Optimization Modeling
- "Optimization Expressions" on page 10-6
- "Review or Modify Optimization Problems" on page 10-14
- "Examine Optimization Solution" on page 10-25

Problem-Based Workflow for Solving Equations

> **Note** Optimization Toolbox provides two approaches for solving equations. This topic describes the problem-based approach. "Solver-Based Optimization Problem Setup" describes the solver-based approach.

To solve a system of equations, perform the following steps.

- Create an equation problem object by using `eqnproblem`. A problem object is a container in which you define equations. The equation problem object defines the problem and any bounds that exist in the problem variables.

  For example, create an equation problem.

  ```
  prob = eqnproblem;
  ```
- Create named variables by using `optimvar`. An optimization variable is a symbolic variable that you use to describe the equations. Include any bounds in the variable definitions.

  For example, create a 15-by-3 array of variables named `'x'` with lower bounds of `0` and upper bounds of `1`.

  ```
  x = optimvar('x',15,3,'LowerBound',0,'UpperBound',1);
  ```
- Define equations in the problem variables. For example:

  ```
  sumeq = sum(x,2) == 1;
  prob.Equations.sumeq = sumeq;
  ```

  > **Note** If you have a nonlinear function that is not composed of polynomials, rational expressions, and elementary functions such as `exp`, then convert the function to an optimization expression by using `fcn2optimexpr`. See "Convert Nonlinear Function to Optimization Expression" on page 7-8 and "Supported Operations on Optimization Variables and Expressions" on page 10-43.

  If necessary, include extra parameters in your equations as workspace variables; see "Pass Extra Parameters in Problem-Based Approach" on page 10-11.
- For nonlinear problems, set an initial point as a structure whose fields are the optimization variable names. For example:

  ```
  x0.x = randn(size(x));
  x0.y = eye(4); % Assumes y is a 4-by-4 variable
  ```
- Solve the problem by using `solve`.

  ```
  sol = solve(prob);
  % Or, for nonlinear problems,
  sol = solve(prob,x0)
  ```

In addition to these basic steps, you can review the problem definition before solving the problem by using `show` or `write`. Set options for `solve` by using `optimoptions`, as explained in "Change Default Solver or Options" on page 10-14.

> **Warning** The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

For a basic equation-solving example with polynomials, see "Solve Nonlinear System of Polynomials, Problem-Based" on page 13-23. For a general nonlinear example, see "Solve Nonlinear System of Equations, Problem-Based" on page 13-21. For more extensive examples, see "Systems of Nonlinear Equations".

See Also
eqnproblem | fcn2optimexpr | optimoptions | optimvar | show | solve | write

More About
- "Systems of Nonlinear Equations"
- "Optimization Expressions" on page 10-6
- "Review or Modify Optimization Problems" on page 10-14
- "Examine Optimization Solution" on page 10-25

Optimization Expressions

In this section...

What Are Optimization Expressions?

Optimization expressions are polynomial or rational combinations of optimization variables.

```
x = optimvar('x',3,3); % a 3-by-3 variable named 'x'
expr1 = sum(x,1) % add the columns of x, get a row vector
expr2 = sum(x,2) % add the rows of x, get a column vector
expr3 = sum(sum(x.*randn(3))) % add the elements of x multiplied by random numbers
expr4 = randn(3)*x % multiply a random matrix times x
expr5 = sum(sum(x*diag(1:3))) % multiply the columns of x by their column number and sum the resu
expr6 = sum(sum(x.*x)) % sum of the squares of all the variables
```

Optimization expressions also result from many MATLAB operations on optimization variables, such as transpose or concatenation of variables. For the list of supported operations on optimization expressions, see "Supported Operations on Optimization Variables and Expressions" on page 10-43.

Finally, optimization expressions can be the result of applying `fcn2optimexpr` to a MATLAB function acting on optimization variables. For details, see "Convert Nonlinear Function to Optimization Expression" on page 7-8.

Optimization modeling functions do not allow you to specify complex, `Inf`, or `NaN` values. If you obtain such an expression through operations, the expression cannot be displayed. See "Expression Contains Inf or NaN" on page 10-36.

Expressions for Objective Functions

An objective function is an expression of size 1-by-1.

```
y = optimvar('y',5,3);
expr = sum(y,2); % a 5-by-1 vector
expr2 = [1:5]*expr;
```

The expression `expr` is not suitable for an objective function because it is a vector. The expression `expr2` is suitable for an objective function.

Note If you have a nonlinear function that is not composed of polynomials, rational expressions, and elementary functions such as `exp`, then convert the function to an optimization expression by using `fcn2optimexpr`. See "Convert Nonlinear Function to Optimization Expression" on page 7-8 and "Supported Operations on Optimization Variables and Expressions" on page 10-43.

To include an expression as an objective function in a problem, use dot notation, or include the objective when you create the problem.

```
prob = optimproblem;
prob.Objective = expr2;
% or equivalently
prob = optimproblem('Objective',expr2);
```

To create an expression in a loop, start with an empty expression as returned by `optimexpr`.

```
x = optimvar('x',3,3,'Type','integer','LowerBound',0,'UpperBound',1);
y = optimvar('y',3,3);
expr = optimexpr;
for i = 1:3
    for j = 1:3
        expr = expr + y(j,i) - x(i,j);
    end
end
show(expr)
```

```
  y(1, 1) + y(2, 1) + y(3, 1) + y(1, 2) + y(2, 2) + y(3, 2) + y(1, 3) + y(2, 3) + y(3, 3)
- x(1, 1) - x(2, 1) - x(3, 1) - x(1, 2) - x(2, 2) - x(3, 2) - x(1, 3) - x(2, 3) - x(3, 3)
```

You can create `expr` without any loops:

```
x = optimvar('x',3,3,'Type','integer','LowerBound',0,'UpperBound',1);
y = optimvar('y',3,3);
expr = sum(sum(y' - x));
show(expr)
```

```
  y(1, 1) + y(2, 1) + y(3, 1) + y(1, 2) + y(2, 2) + y(3, 2) + y(1, 3) + y(2, 3) + y(3, 3)
- x(1, 1) - x(2, 1) - x(3, 1) - x(1, 2) - x(2, 2) - x(3, 2) - x(1, 3) - x(2, 3) - x(3, 3)
```

Note If your objective function is a sum of squares, and you want `solve` to recognize it as such, write it as `sum(expr.^2)`, and not as `expr'*expr`. The internal parser recognizes only explicit sums of squares. For an example, see "Nonnegative Linear Least Squares, Problem-Based" on page 12-40.

Expressions for Constraints and Equations

Constraints are any two comparable expressions that include one of these comparison operators: `==`, `<=`, or `>=`. Equations are two comparable expressions that use the comparison operator `==`. Comparable expressions have the same size, or one of the expressions must be scalar, meaning of size 1-by-1.

```
x = optimvar('x',3,2,'Type','integer','LowerBound',0,'UpperBound',1);
y = optimvar('y',2,4);
z = optimvar('z');
```

```
constr1 = sum(x,2) >= z;
```

`x` is of size 3-by-2, so `sum(x,2)` is of size 3-by-1. This expression is comparable to `z` because `z` is a scalar variable.

```
constr2 = y <= z;
```

`y` is of size 2-by-4. Again, `y` is comparable to `z` because `z` is a scalar variable.

```
constr3 = (sum(x,1))' <= sum(y,2);
```

`sum(x,1)` is of size 1-by-2, so `(sum(x,1))'` is of size 2-by-1. `sum(y,2)` is of size 2-by-1, so the two expressions are comparable.

Note If you have a nonlinear function that is not composed of polynomials, rational expressions, and elementary functions such as `exp`, then convert the function to an optimization expression by using `fcn2optimexpr`. See "Convert Nonlinear Function to Optimization Expression" on page 7-8 and "Supported Operations on Optimization Variables and Expressions" on page 10-43.

To include constraints in a problem, use dot notation and give each constraint a different name.

```
prob = optimproblem;
prob.Constraints.constr1 = constr1;
prob.Constraints.constr2 = constr2;
prob.Constraints.constr3 = constr3;
```

Similarly, to include equations in a problem, use dot notation and give each equation a different name.

```
prob = eqnproblem;
prob.Equations.eq1 = eq1;
prob.Equations.eq2 = eq12
```

You can also include constraints or equations when you create a problem. For example, suppose that you have 10 pairs of positive variables whose sums are no more than one.

```
x = optimvar('x',10,2,'LowerBound',0);
prob = optimproblem('Constraints',sum(x,2) <= 1);
```

To create constraint or equation expressions in a loop, start with an empty constraint expression as returned by `optimconstr`, `optimeq`, or `optimineq`.

```
x = optimvar('x',3,2,'Type','integer','LowerBound',0,'UpperBound',1);
y = optimvar('y',2,4);
z = optimvar('z');
const1 = optimconstr(2);
for i = 1:2
    const1(i) = x(1,i) - x(3,i) + 2*z >= 4*(y(i,2) + y(i,3) + 2*y(i,4));
end
show(const1)
```

```
(1, 1)

  x(1, 1) - x(3, 1) + 2*z - 4*y(1, 2) - 4*y(1, 3) - 8*y(1, 4) >= 0

(2, 1)

  x(1, 2) - x(3, 2) + 2*z - 4*y(2, 2) - 4*y(2, 3) - 8*y(2, 4) >= 0
```

You can create `const1` without any loops.

```
x = optimvar('x',3,2,'Type','integer','LowerBound',0,'UpperBound',1);
y = optimvar('y',2,4);
z = optimvar('z');
const1 = x(1,:) - x(3,:) + 2*z >= 4*(y(:,1) + y(:,3) + 2*y(:,4))';
show(const1)
```

```
(1, 1)

  x(1, 1) - x(3, 1) + 2*z - 4*y(1, 1) - 4*y(1, 3) - 8*y(1, 4) >= 0

(1, 2)

  x(1, 2) - x(3, 2) + 2*z - 4*y(2, 1) - 4*y(2, 3) - 8*y(2, 4) >= 0
```

Tip For best performance, include variable bounds in the variable definitions, not in constraint expressions. Also, performance generally improves when you create constraints without using loops. See "Create Efficient Optimization Problems" on page 10-28.

Caution Each constraint expression in a problem must use the same comparison. For example, the following code leads to an error, because `cons1` uses the `<=` comparison, `cons2` uses the `>=` comparison, and `cons1` and `cons2` are in the same expression.

```matlab
prob = optimproblem;
x = optimvar('x',2,'LowerBound',0);
cons1 = x(1) + x(2) <= 10;
cons2 = 3*x(1) + 4*x(2) >= 2;
prob.Constraints = [cons1;cons2]; % This line throws an error
```

You can avoid this error by using separate expressions for the constraints.

```matlab
prob.Constraints.cons1 = cons1;
prob.Constraints.cons2 = cons2;
```

Optimization Variables Have Handle Behavior

- `OptimizationVariable` objects have *handle* copy behavior. See "Handle Object Behavior" and "Comparison of Handle and Value Classes". Handle copy behavior means that a copy of an `OptimizationVariable` points to the original and does not have an independent existence. For example, create a variable `x`, copy it to `y`, then set a property of `y`. Note that `x` takes on the new property value.

  ```matlab
  x = optimvar('x','LowerBound',1);
  y = x;
  y.LowerBound = 0;
  showbounds(x)
  ```

  ```
      0 <= x
  ```

See Also

`OptimizationConstraint` | `OptimizationExpression` | `optimvar` | `show`

More About

- "Mixed-Integer Linear Programming Basics: Problem-Based" on page 10-47
- "Problem-Based Optimization Workflow" on page 10-2
- "Review or Modify Optimization Problems" on page 10-14

Pass Extra Parameters in Problem-Based Approach

In an optimization problem, the objective or constraint functions sometimes 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.

To include these parameters in the problem-based approach, simply refer to workspace variables in your objective or constraint functions.

Least-Squares Problem with Passed Data

For example, suppose that you have matrices C and d in the `particle.mat` file, and these matrices represent data for your problem. Load the data into your workspace.

```
load particle
```

View the sizes of the matrices.

```
disp(size(C))
```

```
      2000         400
```

```
disp(size(d))
```

```
      2000           1
```

Create an optimization variable x of a size that is suitable for forming the vector C*x.

```
x = optimvar('x',size(C,2));
```

Create an optimization problem to minimize the sum of squares of the terms in C*x − d subject to the constraint that x is nonnegative.

```
x.LowerBound = 0;
prob = optimproblem;
expr = sum((C*x - d).^2);
prob.Objective = expr;
```

You include the data C and d into the problem simply by referring to them in the objective function expression. Solve the problem.

```
[sol,fval,exitflag,output] = solve(prob)
```

```
Solving problem using lsqlin.

Minimum found that satisfies the constraints.

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

sol = struct with fields:
    x: [400x1 double]


fval = 22.5795
```

```
exitflag =
    OptimalSolution


output = struct with fields:
            message: '...'
          algorithm: 'interior-point'
       firstorderopt: 9.9673e-07
      constrviolation: 0
           iterations: 9
         linearsolver: 'sparse'
         cgiterations: []
               solver: 'lsqlin'
```

Nonlinear Problem with Extra Parameters

Use the same approach for nonlinear problems. For example, suppose that you have an objective function of several variables, some of which are fixed data for the optimization.

```
type parameterfun
```

```
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;
```

For this objective function, x is a 2-element vector, and a, b, and c are scalar parameters. Create the optimization variable and assign the parameter values in your workspace.

```
a = 4;
b = 2.1;
c = 4;
x = optimvar('x',2);
```

Create an optimization problem. Because this objective function is a rational function of x, you can specify the objective in terms of the optimization variable. Solve the problem starting from the point x0.x = [1/2;1/2].

```
prob = optimproblem;
prob.Objective = parameterfun(x,a,b,c);
x0.x = [1/2;1/2];
[sol,fval] = solve(prob,x0)
```

```
Solving problem using fminunc.

Local minimum found.

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

sol = struct with fields:
    x: [2x1 double]


fval = -1.0316
```

If parameterfun were not composed of supported functions, you would convert parameterfun to an optimization expression and set the converted expression as the objective. See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8.

```
expr = fcn2optimexpr(@parameterfun,x,a,b,c);
prob.Objective = expr;
[sol,fval] = solve(prob,x0)
```

Solving problem using fminunc.

Local minimum found.

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

sol = *struct with fields:*
    x: [2x1 double]

fval = -1.0316

Copyright 2018–2020 The MathWorks, Inc.

See Also

fcn2optimexpr

More About

*   "Passing Extra Parameters" on page 2-57

Review or Modify Optimization Problems

In this section...
"Review Problem Using show or write" on page 10-14
"Change Default Solver or Options" on page 10-14
"Correct a Misspecified Problem" on page 10-16
"Duplicate Variable Name" on page 10-19

Review Problem Using show or write

After you create an optimization problem, you can review its formulation by using `show`. For large problems, use `write` instead. For example,

```
prob = optimproblem;
x = optimvar('x',2,'LowerBound',0);
prob.Objective = x(1) - 2*x(2);
prob.Constraints.cons1 = x(1) + 2*x(2) <= 4;
prob.Constraints.cons2 = -x(1) + x(2) <= 1;
```

```
show(prob)
```

```
  OptimizationProblem :

    Solve for:
       x

    minimize :
       x(1) - 2*x(2)


    subject to cons1:
       x(1) + 2*x(2) <= 4

    subject to cons2:
       -x(1) + x(2) <= 1

    variable bounds:
       0 <= x(1)
       0 <= x(2)
```

This review shows the basic elements of the problem, such as whether the problem is to minimize or maximize, and the variable bounds. The review shows the index names, if any, used in the variables. The review does not show whether the variables are integer valued.

Change Default Solver or Options

To attempt to improve a solution or speed of solution, examine and change the default solver or options.

To see the default solver and options, use `optimoptions(prob)`. For example,

```
rng default
x = optimvar('x',3,'LowerBound',0);
```

```
expr = sum((rand(3,1).*x).^2);
prob = optimproblem('Objective',expr);
prob.Constraints.lincon = sum(sum(randn(size(x)).*x)) <= randn;
options = optimoptions(prob)

options =

  lsqlin options:

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

   Set properties:
     No options set.

   Default properties:
               Algorithm: 'interior-point'
     ConstraintTolerance: 1.0000e-08
                 Display: 'final'
            LinearSolver: 'auto'
           MaxIterations: 200
     OptimalityTolerance: 1.0000e-08
           StepTolerance: 1.0000e-12

   Show options not used by current Algorithm ('interior-point')
```

The default solver for this problem is `lsqlin`, and you can see the default options.

To change the solver, set the `'Solver'` name-value pair in `solve`. To see the applicable options for a different solver, use `optimoptions` to pass the current options to the different solver. For example, continuing the problem,

```
options = optimoptions('quadprog',options)

options =

  quadprog options:

   Options used by current Algorithm ('interior-point-convex'):
   (Other available algorithms: 'trust-region-reflective')

   Set properties:
     ConstraintTolerance: 1.0000e-08
           MaxIterations: 200
     OptimalityTolerance: 1.0000e-08
           StepTolerance: 1.0000e-12

   Default properties:
               Algorithm: 'interior-point-convex'
                 Display: 'final'
            LinearSolver: 'auto'

   Show options not used by current Algorithm ('interior-point-convex')
```

To change the options, use `optimoptions` or dot notation to set options, and pass the options to `solve` in the `'Options'` name-value pair. See "Options in Common Use: Tuning and Troubleshooting" on page 2-61. Continuing the example,

```
options.Display = 'iter';
sol = solve(prob,'Options',options,'Solver','quadprog');

 Iter            Fval    Primal Infeas     Dual Infeas    Complementarity
    0    1.500359e+00    3.068423e-01    2.275437e+00       2.500000e-01
    1    1.728717e-01    0.000000e+00    7.719860e-03       3.637874e-02
    2    2.604108e-02    0.000000e+00    0.000000e+00       5.245260e-03
    3    7.822161e-03    0.000000e+00    2.775558e-17       1.407915e-03
    4    2.909218e-03    0.000000e+00    6.938894e-18       2.070784e-04
    5    1.931264e-03    0.000000e+00    1.734723e-18       2.907724e-05
    6    1.797508e-03    0.000000e+00    2.602085e-18       4.083167e-06
    7    1.775398e-03    0.000000e+00    4.336809e-19       5.102453e-07
    8    1.772971e-03    0.000000e+00    2.632684e-19       3.064243e-08
    9    1.772848e-03    0.000000e+00    5.228973e-19       4.371356e-11

Minimum found that satisfies the constraints.

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

Correct a Misspecified Problem

To check that your problem is correct, review all its aspects. For example, create an optimization problem to solve a Sudoku problem by running this script.

```
x = optimvar('x',9,9,9,'LowerBound',0,'UpperBound',1);
cons1 = sum(x,1) == 1;
cons2 = sum(x,2) == 1;
cons3 = sum(x,3) == 1;
prob = optimproblem;
prob.Constraints.cons1 = cons1;
prob.Constraints.cons2 = cons2;
prob.Constraints.cons3 = cons3;
mul = ones(1,1,9);
mul = cumsum(mul,3);
prob.Objective = sum(sum(sum(x,1),2).*mul);
cons4 = optimconstr(3,3,9);

for u = 1:3
    for v = 1:3
        arr = x(3*(u-1)+1:3*(u-1)+3,3*(v-1)+1:3*(v-1)+3,:);
        cons4(u,v,:) = sum(sum(arr,1),2) <= ones(1,1,9);
    end
end
prob.Constraints.cons4 = cons4;

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];

for u = 1:size(B,1)
    x.LowerBound(B(u,1),B(u,1),B(u,1)) = 1;
end
```

This script has some errors that you can find by examining the variables, objective, and constraints. First, examine the variable x.

```
x
```

```
x =

  9×9×9 OptimizationVariable array with properties:

 Array-wide properties:
          Name: 'x'
          Type: 'continuous'
     IndexNames: {{}  {}  {}}

 Elementwise properties:
     LowerBound: [9×9×9 double]
     UpperBound: [9×9×9 double]

See variables with show.
See bounds with showbounds.
```

This display shows that the type of the variable is continuous. The variable should be integer valued. Change the type.

```
x.Type = 'integer'
```

```
x =

  9×9×9 OptimizationVariable array with properties:

 Array-wide properties:
          Name: 'x'
          Type: 'integer'
     IndexNames: {{}  {}  {}}

 Elementwise properties:
     LowerBound: [9×9×9 double]
     UpperBound: [9×9×9 double]

See variables with show.
See bounds with showbounds.
```

10-17

Check the bounds. There should be 21 lower bounds with the value 1, one for each row of B. Because x is a large array, write the bounds to a file instead of displaying them at the command line.

```
writebounds(x,'xbounds.txt')
```

Search the file xbounds.txt for all instances of 1 <=. Only nine lower bounds having the value 1, in the variables x(1,1,1), x(2,2,2), ..., x(9,9,9). To investigate this discrepancy, examine the code where you set the lower bounds:

```
for u = 1:size(B,1)
    x.LowerBound(B(u,1),B(u,1),B(u,1)) = 1;
end
```

The line inside the loop should say x.LowerBound(B(u,1),B(u,2),B(u,3)) = 1;. Reset all lower bounds to zero, then run the corrected code.

```
x.LowerBound = 0;
for u = 1:size(B,1)
    x.LowerBound(B(u,1),B(u,2),B(u,3)) = 1;
end
writebounds(x,'xbounds.txt')
```

xbounds.txt now has the correct number of lower bound entries that are 1.

Examine the objective function. The objective function expression is large, so write the expression to a file.

```
write(prob.Objective,'objectivedescription.txt')
```

```
    x(1, 1, 1) + x(2, 1, 1) + x(3, 1, 1) + x(4, 1, 1) + x(5, 1, 1) + x(6, 1, 1) + x(7, 1, 1) + x
    1, 1) + x(9, 1, 1) + x(1, 2, 1) + x(2, 2, 1) + x(3, 2, 1) + x(4, 2, 1) + x(5, 2, 1) + x(6, 2
    ...
    9*x(7, 8, 9) + 9*x(8, 8, 9) + 9*x(9, 8, 9) + 9*x(1, 9, 9) + 9*x(2, 9, 9) + 9*x(3, 9, 9) +
    9*x(4, 9, 9) + 9*x(5, 9, 9) + 9*x(6, 9, 9) + 9*x(7, 9, 9) + 9*x(8, 9, 9) + 9*x(9, 9, 9)
```

The objective function looks reasonable, because it is a sum of scalar expressions.

Write the constraints to files for examination.

```
write(prob.Constraints.cons1,'cons1.txt')
write(prob.Constraints.cons2,'cons2.txt')
write(prob.Constraints.cons3,'cons3.txt')
write(prob.Constraints.cons4,'cons4.txt')
```

Review cons4.txt and you see a mistake. All the constraints are inequalities rather than equalities. Correct the lines of code that create this constraint and put the corrected constraint in the problem.

```
cons4 = optimconstr(3,3,9);

for u = 1:3
    for v = 1:3
        arr = x(3*(u-1)+1:3*(u-1)+3,3*(v-1)+1:3*(v-1)+3,:);
        cons4(u,v,:) = sum(sum(arr,1),2) == ones(1,1,9);
    end
end
prob.Constraints.cons4 = cons4;
```

After these changes, you can successfully solve the problem.

```
sol = solve(prob);
x = round(sol.x);
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

drawSudoku(S)
```

Duplicate Variable Name

If you recreate a variable, but already have an expression that uses the old variable, then you can get errors when incorporating the expressions into a single problem. See "Variables with Duplicate Names Disallowed" on page 10-35.

See Also

OptimizationConstraint | OptimizationExpression | OptimizationProblem | OptimizationVariable | show | showbounds | write | writebounds

More About

- "Problem-Based Optimization Workflow" on page 10-2
- "Mixed-Integer Linear Programming Basics: Problem-Based" on page 10-47

Named Index for Optimization Variables

In this section...
"Create Named Indices" on page 10-20
"Use Named Indices" on page 10-21
"View Solution with Index Variables" on page 10-22

Create Named Indices

Optimization variables can use names for indexing elements. You can give the names when you create a variable or afterward. For example, give the names while creating the variable.

```
x = optimvar('x',["United","Lufthansa","Virgin Air"])

x =
  1x3 OptimizationVariable array with properties:

  Array-wide properties:
        Name: 'x'
        Type: 'continuous'
   IndexNames: {{}  {1x3 cell}}

  Elementwise properties:
    LowerBound: [-Inf -Inf -Inf]
    UpperBound: [Inf Inf Inf]

  See variables with show.
  See bounds with showbounds.
```

`optimvar` automatically maps the names you specify to index numbers in the order of your variables. For example, `"United"` corresponds to index 1, `"Lufthansa"` corresponds to index 2, and `"Virgin Air"` corresponds to index 3. Display this last variable for confirmation.

```
show(x(3))

    [ x('Virgin Air') ]
```

Index names enable you to address elements of `x` by the index names. For example:

```
route = 2*x("United") + 3*x("Virgin Air")

route =
  Linear OptimizationExpression

    2*x('United') + 3*x('Virgin Air')
```

You can create or change the index names after you create a variable. However, you cannot change the size of an optimization variable after construction. So you can change index names only by setting new names that index the same size as the original variable. For example:

```
x = optimvar('x',3,2);
x.IndexNames = { {'row1','row2','row3'}, {'col1','col2'} };
```

You can also set the index names for each dimension individually:

```
x.IndexNames{1} = {'row1', 'row2', 'row3'};
x.IndexNames{2} = {'col1', 'col2'};
```

You can set an index name for a particular element:

```
x.IndexNames{1}{2} = 'importantRow';
```

Examine the index names for the variable.

```
x.IndexNames{1}
```

```
ans = 1x3 cell
    {'row1'}    {'importantRow'}    {'row3'}
```

```
x.IndexNames{2}
```

```
ans = 1x2 cell
    {'col1'}    {'col2'}
```

Use Named Indices

You can create and debug some problems easily by using named index variables. For example, consider the variable x that is indexed by the names in `vars`:

```
vars = {'P1','P2','I1','I2','C','LE1','LE2','HE1','HE2',...
    'HPS','MPS','LPS','BF1','BF2','EP','PP'};
x = optimvar('x',vars,'LowerBound',0);
```

Create bounds, an objective function, and linear constraints for x by using the named indices.

```
x('P1').LowerBound = 2500;
x('I2').UpperBound = 244000;
linprob = optimproblem;
linprob.Objective = 0.002614*x('HPS') + 0.0239*x('PP') + 0.009825*x('EP');
linprob.Constraints.cons1 = x('I1') - x('HE1') <= 132000;
```

You can use strings (" ") or character vectors (' ') in index variables indiscriminately. For example:

```
x("P2").LowerBound = 3000;
x('MPS').LowerBound = 271536;
showbounds(x)
```

```
      2500 <= x('P1')
      3000 <= x('P2')
         0 <= x('I1')
         0 <= x('I2')   <= 244000
         0 <= x('C')
         0 <= x('LE1')
         0 <= x('LE2')
         0 <= x('HE1')
         0 <= x('HE2')
         0 <= x('HPS')
    271536 <= x('MPS')
         0 <= x('LPS')
```

```
            0 <= x('BF1')
            0 <= x('BF2')
            0 <= x('EP')
            0 <= x('PP')
```

There is no distinction between variables you specified with a string, such as x("P2"), and variables you specified with a character vector, such as x('MPS').

Because named index variables have numeric equivalents, you can use ordinary summation and colon operators even when you have named index variable. For example, you can have constraints of these forms:

```
constr = sum(x) <= 100;
show(constr)

  x('P1') + x('P2') + x('I1') + x('I2') + x('C') + x('LE1') + x('LE2')
+ x('HE1') + x('HE2') + x('HPS') + x('MPS') + x('LPS') + x('BF1') + x('BF2')
+ x('EP') + x('PP') <= 100

y = optimvar('y',{'red','green','blue'},{'plastic','wood','metal'},...
    'Type','integer','LowerBound',0);
constr2 = y("red",:) == [5,7,3];
show(constr2)

(1, 1)

  y('red', 'plastic') == 5

(1, 2)

  y('red', 'wood') == 7

(1, 3)

  y('red', 'metal') == 3
```

View Solution with Index Variables

Create and solve an optimization problem using named index variables. The problem is to maximize the profit-weighted flow of fruit to various airports, subject to constraints on the weighted flows.

```
rng(0) % For reproducibility
p = optimproblem('ObjectiveSense', 'maximize');
flow = optimvar('flow', ...
    {'apples', 'oranges', 'bananas', 'berries'}, {'NYC', 'BOS', 'LAX'}, ...
    'LowerBound',0,'Type','integer');
p.Objective = sum(sum(rand(4,3).*flow));
p.Constraints.NYC = rand(1,4)*flow(:,'NYC') <= 10;
p.Constraints.BOS = rand(1,4)*flow(:,'BOS') <= 12;
p.Constraints.LAX = rand(1,4)*flow(:,'LAX') <= 35;
sol = solve(p);

Solving problem using intlinprog.
LP:               Optimal objective value is -1027.472366.

Heuristics:       Found 1 solution using ZI round.
```

```
                    Upper bound is -1027.233133.
                    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.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).
```

Find the optimal flow of oranges and berries to New York and Los Angeles.

```
[idxFruit,idxAirports] = findindex(flow, {'oranges','berries'}, {'NYC', 'LAX'})

idxFruit = 1×2

     2     4


idxAirports = 1×2

     1     3


orangeBerries = sol.flow(idxFruit, idxAirports)

orangeBerries = 2×2

         0   980.0000
   70.0000         0
```

This display means that no oranges are going to NYC, 70 berries are going to NYC, 980 oranges are going to LAX, and no berries are going to LAX.

List the optimal flow of the following:

```
Fruit Airports

----- --------

Berries NYC

Apples BOS

Oranges LAX

idx = findindex(flow, {'berries', 'apples', 'oranges'}, {'NYC', 'BOS', 'LAX'})

idx = 1×3

     4     5    10


optimalFlow = sol.flow(idx)

optimalFlow = 1×3
```

```
    70.0000   28.0000   980.0000
```

This display means that 70 berries are going to NYC, 28 apples are going to BOS, and 980 oranges are going to LAX.

See Also

findindex | optimvar

More About

*   "Problem-Based Optimization Workflow" on page 10-2
*   "Create Initial Point for Optimization with Named Index Variables" on page 10-50

Examine Optimization Solution

In this section...
"Obtain Numeric Solution" on page 10-25
"Examine Solution Quality" on page 10-26
"Infeasible Solution" on page 10-26
"Solution Takes Too Long" on page 10-27

Obtain Numeric Solution

The `solve` function returns a solution as a structure, with each variable in the problem having a field in the structure. To obtain numerical values of expressions in the problem from this structure easily, use the `evaluate` function.

For example, solve a linear programming problem in two variables.

```
x = optimvar('x');
y = optimvar('y');
prob = optimproblem;
prob.Objective = -x -y/3;
prob.Constraints.cons1 = x + y <= 2;
prob.Constraints.cons2 = x + y/4 <= 1;
prob.Constraints.cons3 = x - y <= 2;
prob.Constraints.cons4 = x/4 + y >= -1;
prob.Constraints.cons5 = x + y >= 1;
prob.Constraints.cons6 = -x + y <= 2;
```

```
sol = solve(prob)
```

```
Solving problem using linprog.

Optimal solution found.

sol =

  struct with fields:

    x: 0.6667
    y: 1.3333
```

Suppose that you want the objective function value at the solution. You can rerun the problem, this time asking for the objective function value and the solution.

```
[sol,fval] = solve(prob)
```

```
Solving problem using linprog.

Optimal solution found.

sol =

  struct with fields:

    x: 0.6667
```

```
    y: 1.3333


fval =

   -1.1111
```

Alternatively, for a time-consuming problem, save time by evaluating the objective function at the solution using `evaluate`.

```
fval = evaluate(prob.Objective,sol)

fval =

   -1.1111
```

Examine Solution Quality

To check whether the reported solution is accurate, you can review outputs from `solve`. Return all `solve` outputs

```
[sol,fval,exitflag,output,lambda] = solve(prob);
```

- Check the exit flag. `exitflag = OptimalSolution` generally means that `solve` converged to the solution. For an explanation of the other `exitflag` values, see `exitflag`.
- Check the exit message at the command line or in the output structure. When the exit message states that the solver converged to a solution, then generally the solution is reliable. This message corresponds to `exitflag = OptimalSolution`.
- When you have integer constraints, check the absolute gap and the relative gap in the exit message or in the output structure. When these gaps are zero or nearly zero, the solution is reliable.

Infeasible Solution

If `solve` reports that your problem is infeasible (the exit flag is `NoFeasiblePointFound`), examine the problem infeasibility at a variety of points to see which constraints might be overly restrictive. Suppose that you have a single continuous optimization variable named `x` that has finite bounds on all components, and you have constraints `constr1` through `constr20`.

```
N = 100; % check 100 points
infeas = zeros(N,20); % allocate
L = x.LowerBound;
U = x.UpperBound;
S = numel(L);
pthist = cell(N);
for k = 1:N
    pt = L + rand(size(L)).*(U-L);
    pthist{k} = pt;
    for j = 1:20
        infeas(k,j) = infeasibility(['constr',num2str(j)],pt);
    end
end
```

The result `infeas(a,b)` has nonzero values wherever the associated point `pt{a}` is infeasible for constraint `b`.

Solution Takes Too Long

If `solve` takes a long time, there are a few possible causes and remedies.

* *Problem formulation is slow*. If you have defined objective or constraint expressions in nested loops, then `solve` can take a long time to convert the problem internally to a matrix form. To speed the solution, try to formulate your expressions in a vectorized fashion. See "Create Efficient Optimization Problems" on page 10-28.

* *Mixed-integer linear programming solution is slow*. Sometimes you can speed up an integer problem by setting options. You can also reformulate the problem to make it faster to solve. See "Tuning Integer Linear Programming" on page 9-52.

* *Nonlinear programming solution is slow*. For suggestions, see "Solver Takes Too Long" on page 4-9. For further suggestions, see "When the Solver Fails" on page 4-3.

* *Solver Limit Exceeded*. To solve some problems, `solve` can take more than the default number of solution steps. For problems with integer constraints, increase the number of allowed steps by increasing the `LPMaxIterations`, `MaxNodes`,`MaxTime`, or `RootLPMaxIterations` options to higher-than-default values. To set these options, use `optimoptions('intlinprog',...)`. For non-integer problems, increase the `MaxIterations` option using `optimoptions('linprog','MaxIterations',...)`. See `options`.

See Also
`evaluate` | `infeasibility` | `solve`

More About
* "Tuning Integer Linear Programming" on page 9-52
* "Exit Flags and Exit Messages" on page 3-3
* "Output Structures" on page 3-21
* "Lagrange Multiplier Structures" on page 3-22
* "Mixed-Integer Linear Programming Basics: Problem-Based" on page 10-47

Create Efficient Optimization Problems

When a problem has integer constraints, `solve` calls `intlinprog` to obtain the solution. For suggestions on obtaining a faster solution or more integer-feasible points, see "Tuning Integer Linear Programming" on page 9-52.

Before you start solving the problem, sometimes you can improve the formulation of your problem constraints or objective. Usually, the software can create expressions for the objective function or constraints more quickly in a vectorized fashion rather than in a loop. This speed difference is especially large when an optimization expression is subject to automatic differentiation; see "Automatic Differentiation in Optimization Toolbox" on page 10-41.

Suppose that your objective function is

$$\sum_{i=1}^{30}\sum_{j=1}^{30}\sum_{k=1}^{10} x_{i,j,k} b_k c_{i,j},$$

where `x` is an optimization variable, and `b` and `c` are constants. Two general ways to formulate this objective function are as follows:

- Use a `for` loop.

```
x = optimvar('x',30,30,10);
b = optimvar('b',10);
c = optimvar('c',30,30);
tic
expr = optimexpr;
for i = 1:30
    for j = 1:30
        for k = 1:10
            expr = expr + x(i,j,k)*b(k)*c(i,j);
        end
    end
end
toc
```

```
Elapsed time is 307.459465 seconds.
```

    Here, `expr` contains the objective function expression. Although this method is straightforward, it can require excessive time to loop through many levels of `for` loops.

- Use a vectorized statement. Vectorized statements generally run faster than a `for` loop. You can create a vectorized statement in several ways.

  - Expand `b` and `c`. To enable term-wise multiplication, create constants that are the same size as `x`.

```
tic
bigb = reshape(b,1,1,10);
bigb = repmat(bigb,30,30,1);
bigc = repmat(c,1,1,10);
expr = sum(sum(sum(x.*bigb.*bigc)));
toc
```

```
Elapsed time is 0.013631 seconds.
```
  - Loop once over `b`.

```
tic
expr = optimexpr;
for k = 1:10
    expr = expr + sum(sum(x(:,:,k).*c))*b(k);
end
toc
```

Elapsed time is 0.044985 seconds.

- Create an expression by looping over b and then summing terms after the loop.

```
tic
expr = optimexpr(30,30,10);
for k = 1:10
    expr(:,:,k) = x(:,:,k).*c*b(k);
end
expr = sum(expr(:));
toc
```

Elapsed time is 0.039518 seconds.

Observe the speed difference between a vectorized and nonvectorized implementation of the example "Constrained Electrostatic Nonlinear Optimization, Problem-Based" on page 7-14. This example was timed using automatic differentiation in R2020b.

```
N = 30;
x = optimvar('x',N,'LowerBound',-1,'UpperBound',1);
y = optimvar('y',N,'LowerBound',-1,'UpperBound',1);
z = optimvar('z',N,'LowerBound',-2,'UpperBound',0);
elecprob = optimproblem;
elecprob.Constraints.spherec = (x.^2 + y.^2 + (z+1).^2) <= 1;
elecprob.Constraints.plane1 = z <= -x-y;
elecprob.Constraints.plane2 = z <= -x+y;
elecprob.Constraints.plane3 = z <= x-y;
elecprob.Constraints.plane4 = z <= x+y;

rng default % For reproducibility
x0 = randn(N,3);
for ii=1:N
    x0(ii,:) = x0(ii,:)/norm(x0(ii,:))/2;
    x0(ii,3) = x0(ii,3) - 1;
end
init.x = x0(:,1);
init.y = x0(:,2);
init.z = x0(:,3);
opts = optimoptions('fmincon','Display','off');

tic
energy = optimexpr(1);
for ii = 1:(N-1)
    jj = (ii+1):N; % Vectorized
    tempe = (x(ii) - x(jj)).^2 + (y(ii) - y(jj)).^2 + (z(ii) - z(jj)).^2;
    energy = energy + sum(tempe.^(-1/2));
end
elecprob.Objective = energy;
disp('Vectorized computation time:')
[sol,fval,exitflag,output] = solve(elecprob,init,'Options',opts);
toc
```

```
Vectorized computation time:
Elapsed time is 1.838136 seconds.

tic
energy2 = optimexpr(1); % For nonvectorized comparison
for ii = 1:(N-1)
    for jjj = (ii+1):N; % Not vectorized
        energy2 = energy2 + ((x(ii) - x(jjj))^2 + (y(ii) - y(jjj))^2 + (z(ii) - z(jjj))^2)^(-1/2)
    end
end
elecprob.Objective = energy2;
disp('Non-vectorized computation time:')
[sol,fval,exitflag,output] = solve(elecprob,init,'Options',opts);
toc

Non-vectorized computation time:
Elapsed time is 204.615210 seconds.
```

The vectorized version is about 100 times faster than the nonvectorized version.

See Also

More About

- "Tuning Integer Linear Programming" on page 9-52
- "Separate Optimization Model from Data" on page 10-31

Separate Optimization Model from Data

To obtain a scalable, reusable optimization problem, create the problem in a way that separates the problem data from the model structure.

Suppose that you have a multiperiod scheduling problem with several products. The time periods are in a vector, `periods`, and the products are in a string vector, `products`.

```
periods = 1:10;
products = ["strawberry","cherry","red grape",...
    "green grape","nectarine","apricot"];
```

To create variables that represent the number of products used in each period, use statements that take sizes from the data. For example:

```
usage = optimvar('usage',length(periods),products,...
    'Type','integer','LowerBound',0);
```

To later change the time periods or products, you need to change the data only in `periods` and `products`. You can then run the same code to create `usage`.

In other words, to maintain flexibility and allow for reuse, do not use a statement that has hard-coded data sizes. For example:

```
usage = optimvar('usage',10,6,... % DO NOT DO THIS
    'Type','Integer','LowerBound',0);
```

The same consideration holds for expressions as well as variables. Suppose that the costs for the products are in a data matrix, `costs`, of size `length(periods)`-by-`length(products)`. To simulate valid data, create a random integer matrix of the appropriate size.

```
rng default % For reproducibility
costs = randi(8,length(periods),length(products));
```

The best practice is to create cost expressions that take sizes from the data.

```
costPerYear = sum(costs.*usage,2);
totalCost = sum(costPerYear);
```

In this way, if you ever change the data sizes, the statements that create `costPerYear` and `totalCost` do not change. In other words, to maintain flexibility and allow for reuse, do not use a statement that has hard-coded data sizes. For example:

```
costPerYear = optimexpr(10,1); % DO NOT DO THIS
totalcost = 0;
for yr = 1:10 % DO NOT DO THIS
    costPerYear(i) = sum(costs(i,:).*usage(i,:));
    totalcost = totalcost + costPerYear(i);
end
```

See Also

More About

- "Problem-Based Optimization Workflow" on page 10-2

Problem-Based Optimization Algorithms

Internally, the `solve` function solves optimization problems by calling a solver:

- `linprog` for linear objective and linear constraints
- `intlinprog` for linear objective and linear constraints and integer constraints
- `quadprog` for quadratic objective and linear constraints
- `lsqlin` or `lsqnonneg` for linear least-squares with linear constraints
- `lsqcurvefit` or `lsqnonlin` for nonlinear least-squares with bound constraints
- `fminunc` for problems without any constraints (not even variable bounds) and with a general nonlinear objective function
- `fmincon` for problems with a nonlinear constraint, or with a general nonlinear objective and at least one constraint
- `fzero` for a scalar nonlinear equation
- `lsqlin` for systems of linear equations, with or without bounds
- `fsolve` for systems of nonlinear equations without constraints
- `lsqnonlin` for systems of nonlinear equations with bounds

Before `solve` can call these functions, the problems must be converted to solver form, either by `solve` or some other associated functions or objects. This conversion entails, for example, linear constraints having a matrix representation rather than an optimization variable expression.

The first step in the algorithm occurs as you place optimization expressions into the problem. An `OptimizationProblem` object has an internal list of the variables used in its expressions. Each variable has a linear index in the expression, and a size. Therefore, the problem variables have an implied matrix form. The `prob2struct` function performs the conversion from problem form to solver form. For an example, see "Convert Problem to Structure" on page 16-398.

For nonlinear optimization problems that use the `fmincon` or `fminunc` solvers, `solve` uses automatic differentiation to compute the gradients of the objective function and nonlinear constraint functions. These derivatives apply when the objective and constraint functions are composed of "Supported Operations on Optimization Variables and Expressions" on page 10-43 and do not use the `fcn2optimexpr` function. When automatic differentiation does not apply, solvers estimate derivatives using finite differences. For details of automatic differentiation, see "Automatic Differentiation Background" on page 10-37.

For the default and allowed solvers that `solve` calls, depending on the problem objective and constraints, see `'solver'`. You can override the default by using the `'solver'` name-value pair argument when calling `solve`.

For the algorithm that `intlinprog` uses to solve MILP problems, see "intlinprog Algorithm" on page 9-43. For the algorithms that `linprog` uses to solve linear programming problems, see "Linear Programming Algorithms" on page 9-2. For the algorithms that `quadprog` uses to solve quadratic programming problems, see "Quadratic Programming Algorithms" on page 11-2. For linear or nonlinear least-squares solver algorithms, see "Least-Squares (Model Fitting) Algorithms" on page 12-2. For nonlinear solver algorithms, see "Unconstrained Nonlinear Optimization Algorithms" on page 6-2 and "Constrained Nonlinear Optimization Algorithms" on page 6-19.

For nonlinear equation solving, `solve` internally represents each equation as the difference between the left and right sides. Then `solve` attempts to minimize the sum of squares of the equation

components. For the algorithms for solving nonlinear systems of equations, see "Equation Solving Algorithms" on page 13-2. When the problem also has bounds, `solve` calls `lsqnonlin` to minimize the sum of squares of equation components. See "Least-Squares (Model Fitting) Algorithms" on page 12-2.

Note If your objective function is a sum of squares, and you want `solve` to recognize it as such, write it as `sum(expr.^2)`, and not as `expr'*expr` or any other form. The internal parser recognizes only explicit sums of squares. For details, see "Write Objective Function for Problem-Based Least Squares" on page 12-85. For an example, see "Nonnegative Linear Least Squares, Problem-Based" on page 12-40.

See Also
`intlinprog` | `linprog` | `prob2struct`

More About
*   "intlinprog Algorithm" on page 9-43
*   "Linear Programming Algorithms" on page 9-2
*   "Automatic Differentiation Background" on page 10-37
*   "Create Efficient Optimization Problems" on page 10-28

Variables with Duplicate Names Disallowed

If you use two different variables that have the same name, then optimization expressions, constraints, or problems can throw an error. This error is troublesome when you create a variable, then create an expression using that variable, then recreate the variable. Suppose that you create the following variable and constraint expression:

```
x = optimvar('x',10,2);
cons = sum(x,2) == 1;
```

At this point, you realize that you intended to create integer variables. So you recreate the variable, changing its type.

```
x = optimvar('x',10,2,'Type','integer');
```

Create an objective and problem.

```
obj = sum(x*[2;3]);
prob = optimproblem('Objective',obj);
```

Now try to put the constraint into the problem.

```
prob.Constraints = cons
```

At this point, you get an error message stating that `OptimizationVariables` appearing in the same problem must have distinct "Name" properties. The issue is that when you recreated the x variable, it is a new variable, unrelated to the constraint expression.

You can correct this issue in two ways.

- Create a new constraint expression using the current x.

  ```
  cons = sum(x,2) == 1;
  prob.Constraints = cons;
  ```

- Retrieve the original x variable by creating a problem using the old expression. Update the retrieved variable to have the correct `Type` property. Use the retrieved variable for the problem and objective.

  ```
  oprob = optimproblem('Constraints',cons);
  x = oprob.Variables.x;
  x.Type = 'integer';
  oprob.Objective = sum(x*[2;3]);
  ```

  This method can be useful if you have created more expressions using the old variable than expressions using the new variable.

See Also

More About

- "Problem-Based Optimization Workflow" on page 10-2

Expression Contains Inf or NaN

Optimization modeling functions do not allow you to specify complex, `Inf`, or `NaN` values. However, `Inf` or `NaN` expressions can arise during ordinary operations. Often, these expressions lead to erroneous solutions.

Optimization expressions containing `Inf` or `NaN` cannot be displayed. For example, the largest real number in double precision arithmetic is about `1.8e308`. So `2e308` overflows to `Inf`.

```
x = optimvar('x');
y = 1e308;
expr = 2*x*y

expr =

  OptimizationExpression

    Expression contains Inf or NaN.
```

Similarly, because `Inf - Inf = NaN`, the following expression cannot be displayed.

```
expr = 2*x*y - 3*x*y

expr =

  OptimizationExpression

    Expression contains Inf or NaN.
```

If any of your optimization expressions contain `Inf` or `NaN`, try to eliminate these values before calling `solve`. To do so:

- Search for these expressions by using the `show` or `write` functions.
- Check whether the expressions came from a division by zero or from the addition or multiplication of large quantities. If so, eliminate or correct the expressions.
- Usually, these expressions appear as the result of errors. However, sometimes they arise from poor scaling. If necessary, divide each relevant expression by a large enough scalar so that the expression no longer overflows, or use another scaling operation.

See Also
show | write

More About
- "Problem-Based Optimization Workflow" on page 10-2
- "Review or Modify Optimization Problems" on page 10-14

Automatic Differentiation Background

What Is Automatic Differentiation?

Automatic differentiation (also known as autodiff, AD, or algorithmic differentiation) is a widely used tool in optimization. The `solve` function uses automatic differentiation by default in problem-based optimization for general nonlinear objective functions and constraints; see "Automatic Differentiation in Optimization Toolbox" on page 10-41.

Automatic differentiation is a set of techniques for evaluating derivatives (gradients) numerically. The method uses symbolic rules for differentiation, which are more accurate than finite difference approximations. Unlike a purely symbolic approach, automatic differentiation evaluates expressions numerically early in the computations, rather than carrying out large symbolic computations. In other words, automatic differentiation evaluates derivatives at particular numeric values; it does not construct symbolic expressions for derivatives.

- Forward mode automatic differentiation evaluates a numerical derivative by performing elementary derivative operations concurrently with the operations of evaluating the function itself. As detailed in the next section, the software performs these computations on a computational graph.
- Reverse mode automatic differentiation uses an extension of the forward mode computational graph to enable the computation of a gradient by a reverse traversal of the graph. As the software runs the code to compute the function and its derivative, it records operations in a data structure called a trace.

As many researchers have noted (for example, Baydin, Pearlmutter, Radul, and Siskind [1]), for a scalar function of many variables, reverse mode calculates the gradient more efficiently than forward mode. Because an objective function is scalar, `solve` automatic differentiation uses reverse mode.

Forward Mode

Consider the problem of evaluating this function and its gradient:

$$f(x) = x_1 \exp\left(-\frac{1}{2}\left(x_1^2 + x_2^2\right)\right).$$

Automatic differentiation works at particular points. In this case, take $x_1 = 2$, $x_2 = 1/2$.

The following computational graph encodes the calculation of the function $f(x)$.

Computational Graph
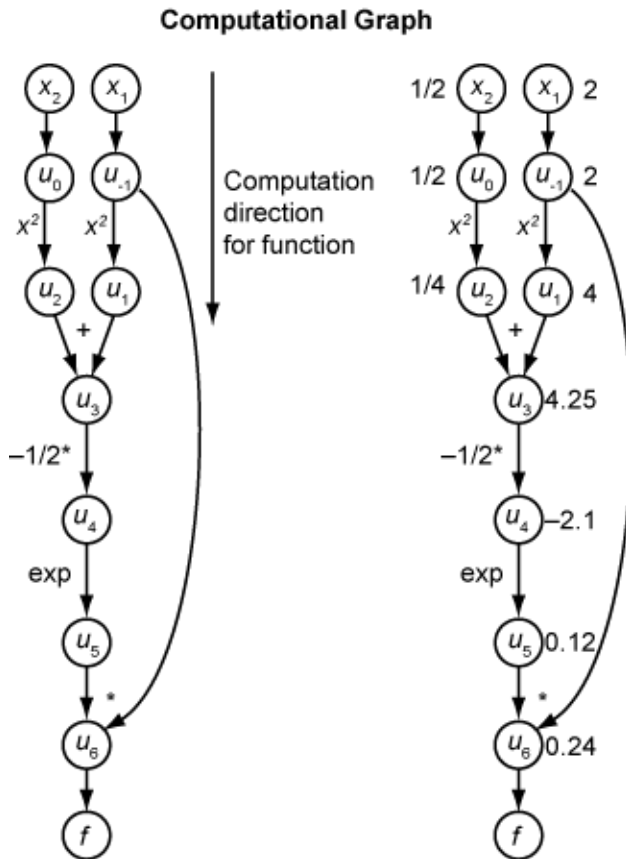
To compute the gradient of $f(x)$ using forward mode, you compute the same graph in the same direction, but modify the computation based on the elementary rules of differentiation. To further simplify the calculation, you fill in the value of the derivative of each subexpression u_i as you go. To compute the entire gradient, you must traverse the graph twice, once for the partial derivative with respect to each independent variable. Each subexpression in the chain rule has a numeric value, so the entire expression has the same sort of evaluation graph as the function itself.

The computation is a repeated application of the chain rule. In this example, the derivative of f with respect to x_1 expands to this expression:

$$\frac{df}{dx_1} = \frac{du_6}{dx_1}$$

$$= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial x_1}$$

$$= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial x_1}$$

$$= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial u_3}\frac{\partial u_3}{\partial x_1}$$

$$= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial u_3}\frac{\partial u_3}{\partial u_1}\frac{\partial u_1}{\partial x_1}.$$

Let $\dot{u}_i$ represent the derivative of the expression u_i with respect to x_1. Using the evaluated values of the u_i from the function evaluation, you compute the partial derivative of f with respect to x_1 as shown

in the following figure. Notice that all the values of the $\dot{u}_i$ become available as you traverse the graph from top to bottom.

Partial Derivative with Respect to x_1
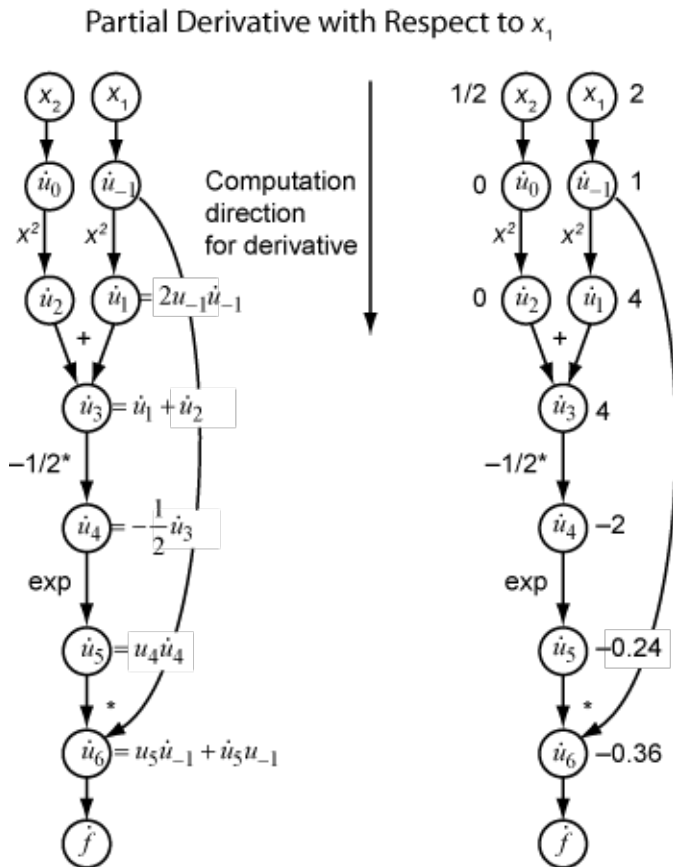
To compute the partial derivative with respect to x_2, you traverse a similar computational graph. Therefore, when you compute the gradient of the function, the number of graph traversals is the same as the number of variables. This process can be slow for many applications, when the objective function or nonlinear constraints depend on many variables.

Reverse Mode

Reverse mode uses one forward traversal of a computational graph to set up the trace. Then it computes the entire gradient of the function in one traversal of the graph in the opposite direction. For problems with many variables, this mode is far more efficient.

The theory behind reverse mode is also based on the chain rule, along with associated adjoint variables denoted with an overbar. The adjoint variable for u_i is

$$\bar{u}_i = \frac{\partial f}{\partial u_i}.$$

In terms of the computational graph, each outgoing arrow from a variable contributes to the corresponding adjoint variable by its term in the chain rule. For example, the variable u_{-1} has outgoing arrows to two variables, u_1 and u_6. The graph has the associated equation

$$\frac{\partial f}{\partial u_{-1}} = \frac{\partial f}{\partial u_1}\frac{\partial u_1}{\partial u_{-1}} + \frac{\partial f}{\partial u_6}\frac{\partial u_6}{\partial u_{-1}}$$

$$= \bar{u}_1\frac{\partial u_1}{\partial u_{-1}} + \bar{u}_6\frac{\partial u_6}{\partial u_{-1}}\ .$$

In this calculation, recalling that $u_1 = u_{-1}^2$ and $u_6 = u_5 u_{-1}$, you obtain

$$\bar{u}_{-1} = \bar{u}_1 2u_{-1} + \bar{u}_6 u_5\ .$$

During the forward traversal of the graph, the software calculates the intermediate variables u_i. During the reverse traversal, starting from the seed value $\bar{u}_6 = \dfrac{\partial f}{\partial f} = 1$, the reverse mode computation obtains the adjoint values for all variables. Therefore, reverse mode computes the gradient in just one computation, saving a great deal of time compared to forward mode.

The following figure shows the computation of the gradient in reverse mode for the function

$$f(x) = x_1\exp\left(-\frac{1}{2}\left(x_1^2 + x_2^2\right)\right)\ .$$

Again, the computation takes $x_1 = 2$, $x_2 = 1/2$. The reverse mode computation relies on the u_i values that are obtained during the computation of the function in the original computational graph. In the right portion of the figure, the computed values of the adjoint variables appear next to the adjoint variable names, using the formulas from the left portion of the figure.
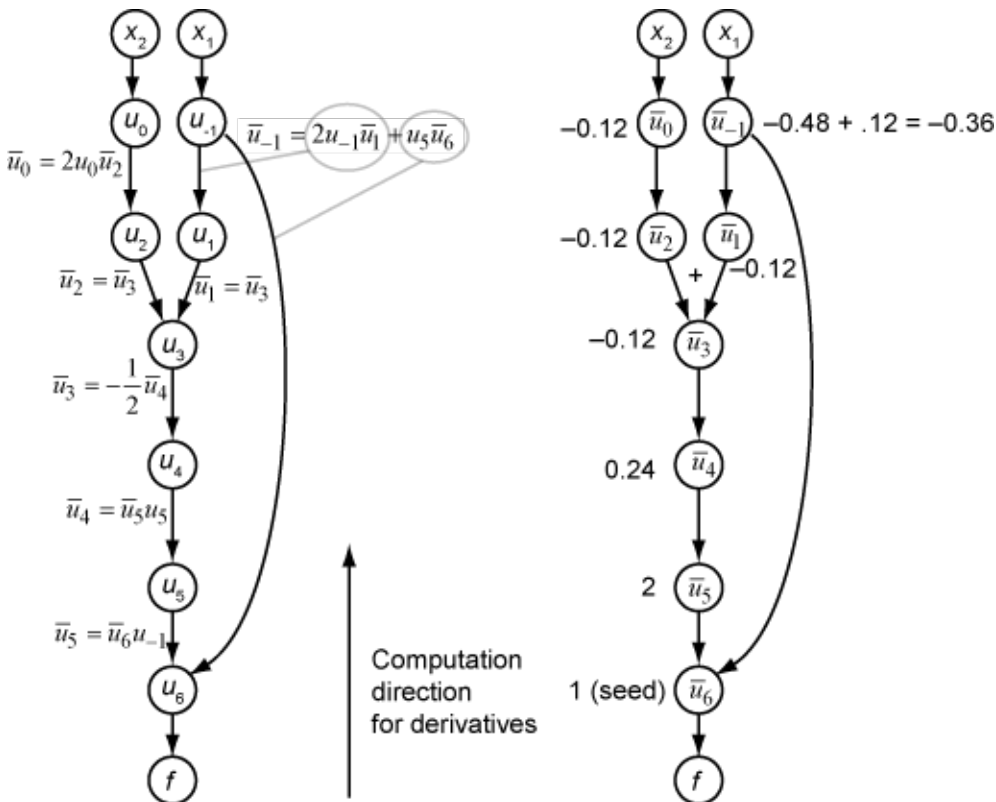
The final gradient values appear as $\bar{u}_0 = \dfrac{\partial f}{\partial u_0} = \dfrac{\partial f}{\partial x_2}$ and $\bar{u}_{-1} = \dfrac{\partial f}{\partial u_{-1}} = \dfrac{\partial f}{\partial x_1}$.

For more details, see Baydin, Pearlmutter, Radul, and Siskind [1] or the Wikipedia article on automatic differentiation [2].

Automatic Differentiation in Optimization Toolbox

Automatic differentiation (AD) applies to the `solve` and `prob2struct` functions under the following conditions:

- The objective and constraint functions are supported, as described in "Supported Operations on Optimization Variables and Expressions" on page 10-43. They do not require use of the `fcn2optimexpr` function.
- The solver called by `solve` is either `fmincon` or `fminunc`.
- The `'ObjectiveDerivative'` and `'ConstraintDerivative'` name-value pair arguments for `solve` or `prob2struct` are set to their default value, `'auto'`.

When AD Applies	All Constraint Functions Supported	One or More Constraints Not Supported
Objective Function Supported	AD used for objective and constraints	AD used for objective only
Objective Function Not Supported	AD used for constraints only	AD not used

When these conditions are not satisfied, `solve` estimates gradients by finite differences, and `prob2struct` does not create gradients in its generated function files.

Note To use automatic derivatives in a problem converted by `prob2struct`, pass options specifying these derivatives.

```
options = optimoptions('fmincon','SpecifyObjectiveGradient',true,...
    'SpecifyConstraintGradient',true);
problem.options = options;
```

Currently, AD works only for first derivatives; it does not apply to second or higher derivatives. So, for example, if you want to use an analytic Hessian to speed your optimization, you cannot use `solve` directly, and must instead use the approach described in "Supply Derivatives in Problem-Based Workflow" on page 7-26.

References

[1] Baydin, Atilim Gunes, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. "Automatic Differentiation in Machine Learning: A Survey." *The Journal of Machine Learning Research,* 18(153), 2018, pp. 1–43. Available at https://arxiv.org/abs/1502.05767.

[2] *Automatic differentiation.* Wikipedia. Available at https://en.wikipedia.org/wiki/Automatic_differentiation.

See Also
`prob2struct` | `solve`

More About

- "Problem-Based Optimization Setup"
- "Effect of Automatic Differentiation in Problem-Based Optimization" on page 7-23

External Websites

- Books on Automatic Differentiation

Supported Operations on Optimization Variables and Expressions

In this section...
"Notation for Supported Operations" on page 10-43
"Operations Returning Optimization Expressions" on page 10-43
"Operations Returning Optimization Variables" on page 10-45
"Operations on Optimization Expressions" on page 10-45
"Operations Returning Constraint Expressions" on page 10-45
"Some Undocumented Operations Work on Optimization Variables and Expressions" on page 10-45
"Unsupported Functions and Operations Require fcn2optimexpr" on page 10-45

Notation for Supported Operations

Optimization variables and expressions are the basic elements of the "Problem-Based Optimization Workflow" on page 10-2. For the legal operations on optimization variables and expressions:

- x and y represent optimization arrays of arbitrary size (usually the same size).
- x2D represents a 2-D optimization array.
- a is a scalar numeric constant.
- M is a constant numeric matrix.
- c is a numeric array of the same size as x.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

Operations Returning Optimization Expressions

These operations on optimization variables or expressions return an optimization expression.

Category	Operation	Example
Arithmetic	Add constant	x+c or c+x
	Add variable	x+y
	Unary plus	+x
	Subtract a constant	x-c
	Subtract variables	x−y
	Unary minus	-x
	Multiply by a constant scalar	a*x or a.*x or x*a or x.*a
	Divide by a constant scalar	x/a or x./a or a\x or a.\x
	Pointwise multiply by an array	c.*x or x.*c

Category	Operation	Example
	Pointwise divide by an array	`x./c` or `c.\x`
	Pointwise multiply variables	`x.*y`
	Matrix multiply variables	`x*y`
	Matrix multiply variable and matrix	`M*x2D` or `x2D*M`
	Dot product of variable and array	`dot(x,c)` or `dot(c,x)`
	Linear combination of variables	`sum(x)`, `sum(x,dim)`, `sum(x,'all')`, `mean(x)`, and `mean(x,dim)`
	Product of array elements	`prod(x)`, `prod(x,dim)`, and `prod(x,'all')`
	Trace of matrix	`trace(x2D)`
	Cumulative sum or product	`cumsum(x)` or `cumprod(x)`, including the syntaxes `cumsum(x,dim)`, `cumsum(_,direction)`, `cumprod(x,dim)`, and `cumprod(_,direction)`
	Differences	`diff(x)`, including the syntaxes `diff(x,n)` and `diff(x,n,dim)`
Concatenate and Reshape	Transpose	`x'` or `x.'`
	Concatenate	`cat`, `vertcat`, and `horzcat`
	Reshape	`reshape(x,[10 1])`
	Create diagonal matrix or get diagonal elements of matrix	`diag(x2D)`, where x2D is a matrix or vector, including the syntax `diag(x2D,k)`
Elementary Functions	Power of square matrix	`x2D^a`
	Pointwise power	`x.^a`
	Square root	`sqrt(x)`
	Sine	`sin(x)`
	Cosine	`cos(x)`
	Tangent	`tan(x)`
	Arcsine	`asin(x)`
	Arccosine	`acos(x)`
	Arctangent	`atan(x)`
	Exponential	`exp(x)`
	Logarithm	`log(x)`
	Hyperbolic sine	`sinh(x)`
	Hyperbolic cosine	`cosh(x)`
	Hyperbolic tangent	`tanh(x)`

Category	Operation	Example
	Inverse hyperbolic sine	`asinh(x)`
	Inverse hyperbolic cosine	`acosh(x)`
	Inverse hyperbolic tangent	`atanh(x)`

Operations Returning Optimization Variables

These operations on optimization variables return an optimization variable.

Operation	Example
N-D numeric indexing (includes colon and `end`)	`x(3,5:end)`
N-D logical indexing	`x(ind)`, where `ind` is a logical array
N-D string indexing	`x(str1,str2)`, where `str1` and `str2` are strings
N-D mixed indexing (combination of numeric, logical, colon, end, and string)	`x(ind,str1,:)`
Linear numeric indexing (includes colon and `end`)	`x(17:end)`
Linear logical indexing	`x(ind)`
Linear string indexing	`x(str1)`

Operations on Optimization Expressions

Optimization expressions support all the operations that optimization variables support, and return optimization expressions. Also, you can index into or assign into an optimization expression using numeric, logical, string, or linear indexing, including the colon and `end` operators for numeric or linear indexing.

Operations Returning Constraint Expressions

Constraints are any two comparable expressions that include one of these comparison operators: ==, <=, or >=. Comparable expressions have the same size, or one of the expressions must be scalar, meaning of size 1-by-1. For examples, see "Expressions for Constraints and Equations" on page 10-7.

Some Undocumented Operations Work on Optimization Variables and Expressions

Internally, some functions and operations call only the documented supported operations. In these cases you can obtain sensible results from the functions or operations. For example, currently `squeeze` internally calls `reshape`, which is a documented supported operation. So if you `squeeze` an optimization variable then you can obtain a sensible expression.

Unsupported Functions and Operations Require fcn2optimexpr

If your objective function or nonlinear constraint functions are not supported, convert a MATLAB function to an optimization expression by using `fcn2optimexpr`. For examples, see "Convert

Nonlinear Function to Optimization Expression" on page 7-8 or the `fcn2optimexpr` function reference page.

See Also

`OptimizationExpression` | `OptimizationVariable` | `fcn2optimexpr`

More About

- "Problem-Based Optimization Setup"
- "Problem-Based Optimization Workflow" on page 10-2
- "Optimization Expressions" on page 10-6
- "Convert Nonlinear Function to Optimization Expression" on page 7-8

Mixed-Integer Linear Programming Basics: Problem-Based

This example shows how to solve a mixed-integer linear problem. Although not complex, the example shows the typical steps in formulating a problem using the problem-based approach. For a video showing this example, see Solve a Mixed-Integer Linear Programming Problem using Optimization Modeling.

For the solver-based approach to this problem, see "Mixed-Integer Linear Programming Basics: Solver-Based" on page 9-54.

Problem Description

You want to blend 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 https://doi.org/10.1287/inte.7.2.39.

Four ingots of steel are available for purchase. Only one of each ingot is available.

Ingot	Weight in Tons	% Carbon	% Molybdenum	Cost $\frac{\text{Cost}}{\text{Ton}}$
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 and one grade of scrap steel are available for purchase. Alloy and scrap steels can be purchased in fractional amounts.

Alloy	% Carbon	% Molybdenum	$\frac{\text{Cost}}{\text{Ton}}$
1	8	6	$ 500
2	7	7	$ 450
3	6	8	$ 400
Scrap	3	9	$ 100

Formulate Problem

To formulate the problem, first decide on the control variables. Take variable `ingots(1) = 1` to mean that you purchase ingot **1**, and `ingots(1) = 0` to mean that you do not purchase the ingot. Similarly, variables `ingots(2)` through `ingots(4)` are binary variables indicating whether you purchase ingots **2** through **4**.

Variables `alloys(1)` through `alloys(3)` are the quantities in tons of alloys **1**, **2**, and **3** that you purchase. `scrap` is the quantity of scrap steel that you purchase.

```
steelprob = optimproblem;
ingots = optimvar('ingots',4,'Type','integer','LowerBound',0,'UpperBound',1);
```

```
alloys = optimvar('alloys',3,'LowerBound',0);
scrap = optimvar('scrap','LowerBound',0);
```

Create expressions for the costs associated with the variables.

```
weightIngots = [5,3,4,6];
costIngots = weightIngots.*[350,330,310,280];
costAlloys = [500,450,400];
costScrap = 100;
cost = costIngots*ingots + costAlloys*alloys + costScrap*scrap;
```

Include the cost as the objective function in the problem.

```
steelprob.Objective = cost;
```

The problem has three equality constraints. The first constraint is that the total weight is 25 tons. Calculate the weight of the steel.

```
totalWeight = weightIngots*ingots + sum(alloys) + scrap;
```

The second constraint is that the weight of carbon is 5% of 25 tons, or 1.25 tons. Calculate the weight of the carbon in the steel.

```
carbonIngots = [5,4,5,3]/100;
carbonAlloys = [8,7,6]/100;
carbonScrap = 3/100;
totalCarbon = (weightIngots.*carbonIngots)*ingots + carbonAlloys*alloys + carbonScrap*scrap;
```

The third constraint is that the weight of molybdenum is 1.25 tons. Calculate the weight of the molybdenum in the steel.

```
molybIngots = [3,3,4,4]/100;
molybAlloys = [6,7,8]/100;
molybScrap = 9/100;
totalMolyb = (weightIngots.*molybIngots)*ingots + molybAlloys*alloys + molybScrap*scrap;
```

Include the constraints in the problem.

```
steelprob.Constraints.conswt = totalWeight == 25;
steelprob.Constraints.conscarb = totalCarbon == 1.25;
steelprob.Constraints.consmolyb = totalMolyb == 1.25;
```

Solve Problem

Now that you have all the inputs, call the solver.

```
[sol,fval] = solve(steelprob);

Solving problem using intlinprog.
LP:              Optimal objective value is 8125.600000.

Cut Generation:    Applied 3 mir cuts.
                   Lower bound is 8495.000000.
                   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.AbsoluteGapTolerance = 0 (the default value). The intcon variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the default value).

View the solution.

```
sol.ingots
```

ans = *4×1*

```
    1.0000
    1.0000
         0
    1.0000
```

```
sol.alloys
```

ans = *3×1*

```
    7.2500
         0
    0.2500
```

```
sol.scrap
```

ans = 3.5000

```
fval
```

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.

See Also

More About

- "Mixed-Integer Linear Programming Basics: Solver-Based" on page 9-54
- "Problem-Based Optimization Workflow" on page 10-2
- Solve a Mixed-Integer Linear Programming Problem using Optimization Modeling

Create Initial Point for Optimization with Named Index Variables

This example shows how to create an initial point for an optimization problem that has named index variables. For named index variables, often the easiest way to specify an initial point is to use the `findindex` function.

The problem is a multiperiod inventory problem that involves blending raw and refined oils. The objective is to maximize profit subject to various constraints on production and inventory capacities and on the "hardness" of oil blends. This problem is taken from Williams [1].

Problem Description

The problem involves two types of raw vegetable oil and three types of raw nonvegetable oil that a manufacturer can refine into edible oil. The manufacturer can refine up to 200 tons of vegetable oils, and up to 250 tons of nonvegetable oils per month. The manufacturer can store 1000 tons of each raw oil, which is beneficial because the cost of purchasing raw oils depends on the month as well as the type of oil. A quality called "hardness" is associated with each oil. The hardness of blended oil is the linearly weighted hardness of the constituent oils.

Because of processing limitations, the manufacturer restricts the number of oils refined in any one month to no more than three. Also, if an oil is refined in a month, at least 20 tons of that oil must be refined. Finally, if a vegetable oil is refined in a month, then nonvegetable oil 3 must also be refined.

The revenue is a constant for each ton of oil sold. The costs are the cost of purchasing the oils, which varies by oil and month, and there is a fixed cost per ton of storing each oil for each month. There is no cost for refining an oil, but the manufacturer cannot store refined oil (it must be sold).

Enter Problem Data

Create named index variables for the planning periods and oils.

```
months = {'January','February','March','April','May','June'};
oils = {'veg1','veg2','non1','non2','non3'};
vegoils = {'veg1','veg2'};
nonveg = {'non1','non2','non3'};
```

Create variables with storage and usage parameters.

```
maxstore = 1000; % Maximum storage of each type of oil
maxuseveg = 200; % Maximum vegetable oil use
maxusenon = 250; % Maximum nonvegetable oil use
minuseraw = 20; % Minimum raw oil use
maxnraw = 3; % Maximum number of raw oils in a blend
saleprice = 150; % Sale price of refined and blended oil
storecost = 5; % Storage cost per period per oil quantity
stockend = 500; % Stock at beginning and end of problem
hmin = 3; % Minimum hardness of refined oil
hmax = 6; % Maximum hardness of refined oil
```

Specify the hardness of the raw oils as this vector.

```
h = [8.8,6.1,2,4.2,5.0];
```

Specify the costs of the raw oils as this array. Each row of the array represents the cost of the raw oils in a month. The first row represents the costs in January, and the last row represents the costs in June.

```
costdata = [...
110 120 130 110 115
130 130 110  90 115
110 140 130 100  95
120 110 120 120 125
100 120 150 110 105
 90 100 140  80 135];
```

Create Variables

Create these problem variables:

- sell, the quantity of each oil sold each month
- store, the quantity of each oil stored at the end of each month
- buy, the quantity of each oil purchased each month

Additionally, to account for constraints on the number of oils refined and sold each month and the minimum quantity produced, create an auxiliary binary variable induse that is 1 exactly when an oil is sold in a month.

```
sell = optimvar('sell', months, oils, 'LowerBound', 0);
buy = optimvar('buy', months, oils, 'LowerBound', 0);
store = optimvar('store', months, oils, 'LowerBound', 0, 'UpperBound', maxstore);

induse = optimvar('induse', months, oils, 'Type', 'integer', ...
    'LowerBound', 0, 'UpperBound', 1);
```

Name the total quantity of oil sold each month produce.

```
produce = sum(sell,2);
```

Create Objective

To create the objective function for the problem, calculate the revenue, and subtract the costs of purchasing and storing oils.

Create an optimization problem for maximization, and include the objective function as the Objective property.

```
prob = optimproblem('ObjectiveSense', 'maximize');
prob.Objective = sum(saleprice*produce) - sum(sum(costdata.*buy)) - sum(sum(storecost*store));
```

The objective expression is quite long. If you like, you can see it using the showexpr(prob.Objective) command.

Create Constraints

The problem has several constraints that you need to set.

The quantity of each oil stored in June is 500. Set this constraint by using lower and upper bounds.

```
store('June', :).LowerBound = 500;
store('June', :).UpperBound = 500;
```

The manufacturer cannot refine more than `maxuseveg` vegetable oil in any month. Set this and all subsequent constraints by using "Expressions for Constraints and Equations" on page 10-7.

```
vegoiluse = sell(:, vegoils);
vegused = sum(vegoiluse, 2) <= maxuseveg;
```

The manufacturer cannot refine more than `maxusenon` nonvegetable oil any month.

```
nonvegoiluse = sell(:,nonveg);
nonvegused = sum(nonvegoiluse,2) <= maxusenon;
```

The hardness of the blended oil must be from `hmin through hmax`.

```
hardmin = sum(repmat(h, 6, 1).*sell, 2) >= hmin*produce;
hardmax = sum(repmat(h, 6, 1).*sell, 2) <= hmax*produce;
```

The amount of each oil stored at the end of the month is equal to the amount at the beginning of the month, plus the amount bought, minus the amount sold.

```
initstockbal = 500 + buy(1, :) == sell(1, :) + store(1, :);
stockbal = store(1:5, :) + buy(2:6, :) == sell(2:6, :) + store(2:6, :);
```

If an oil is refined at all in a month, at least `minuseraw` of the oil must be refined and sold.

```
minuse = sell >= minuseraw*induse;
```

Ensure that the `induse` variable is 1 exactly when the corresponding oil is refined.

```
maxusev = sell(:, vegoils) <= maxuseveg*induse(:, vegoils);
maxusenv = sell(:, nonveg) <= maxusenon*induse(:, nonveg);
```

The manufacturer can sell no more than `maxnraw` oils each month.

```
maxnuse = sum(induse, 2) <= maxnraw;
```

If a vegetable oil is refined, oil `non3` must also be refined and sold.

```
deflogic1 = sum(induse(:,vegoils), 2) <= induse(:,'non3')*numel(vegoils);
```

Include the constraint expressions in the problem.

```
prob.Constraints.vegused = vegused;
prob.Constraints.nonvegused = nonvegused;
prob.Constraints.hardmin = hardmin;
prob.Constraints.hardmax = hardmax;
prob.Constraints.initstockbal = initstockbal;
prob.Constraints.stockbal = stockbal;
prob.Constraints.minuse = minuse;
prob.Constraints.maxusev = maxusev;
prob.Constraints.maxusenv = maxusenv;
prob.Constraints.maxnuse = maxnuse;
prob.Constraints.deflogic1 = deflogic1;
```

Solve Problem

To show the eventual difference between using an initial point and not using one, set options to use no heuristics. Then solve the problem.

```
opts = optimoptions('intlinprog','Heuristics','none');
[sol1,fval1,exitstatus1,output1] = solve(prob,'options',opts)
```

```
Solving problem using intlinprog.
LP:                 Optimal objective value is -1.075130e+05.

Cut Generation:     Applied 41 Gomory cuts, 2 cover cuts,
                    1 mir cut, and 1 clique cut.
                    Lower bound is -1.047522e+05.

Branch and Bound:
```

nodes explored	total time (s)	num int solution	integer fval	relative gap (%)
19	0.21	1	-7.190185e+04	4.535003e+01
25	0.22	2	-7.205000e+04	4.505117e+01
25	0.23	3	-8.290000e+04	2.606702e+01
29	0.23	4	-8.775370e+04	1.909426e+01
40	0.24	5	-9.937870e+04	5.163142e+00
91	0.30	6	-9.987222e+04	4.643483e+00
105	0.30	7	-1.002139e+05	4.286718e+00
630	0.57	8	-1.002787e+05	2.283810e+00
1112	0.76	8	-1.002787e+05	0.000000e+00

```
Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the
optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon
variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the
default value).

sol1 = struct with fields:
       buy: [6x5 double]
    induse: [6x5 double]
      sell: [6x5 double]
     store: [6x5 double]


fval1 = 1.0028e+05

exitstatus1 =
    OptimalSolution


output1 = struct with fields:
        relativegap: 0
        absolutegap: 0
      numfeaspoints: 8
           numnodes: 1112
      constrviolation: 1.1867e-11
            message: 'Optimal solution found....'
             solver: 'intlinprog'
```

Use Initial Point

For this problem, using an initial point can save branch-and-bound iterations. Create an initial point
of the correct dimensions.

```
x0.buy = zeros(size(buy));
x0.induse = zeros(size(induse));
x0.store = zeros(size(store));
x0.sell = zeros(size(sell));
```

Set the initial point to sell only vegetable oil `veg2` and nonvegetable oil `non3`. To set this initial point appropriately, use the `findindex` function.

```
numMonths = size(induse,1);
[idxMonths,idxOils] = findindex(induse,1:numMonths,{'veg2','non3'});
x0.induse(idxMonths,idxOils) = 1;
```

Satisfy the maximum vegetable and nonvegetable oil constraints.

```
x0.sell(:,idxOils) = repmat([200,250],numMonths,1)
```

```
x0 = struct with fields:
       buy: [6x5 double]
    induse: [6x5 double]
     store: [6x5 double]
      sell: [6x5 double]
```

Set the initial point to buy no oil the first month.

```
x0.buy(1,:) = 0;
```

Satisfy the `initstockbal` constraint for the first month based on the initial store of 500 for each oil type, and no purchase the first month, and constant usage of `veg2` and `non3`.

```
x0.store(1,:) = [500 300 500 500 250];
```

Satisfy the remaining stock balance constraints stockbal by using the `findindex` function.

```
[idxMonths,idxOils] = findindex(store,2:6,{'veg2'});
x0.store(idxMonths,idxOils) = [100;0;0;0;500];

[idxMonths,idxOils] = findindex(store,2:6,{'veg1','non1','non2'});
x0.store(idxMonths,idxOils) =   500;

[idxMonths,idxOils] = findindex(store,2:6,{'non3'});
x0.store(idxMonths,idxOils) = [0;0;0;0;500];

[idxMonths,idxOils] = findindex(buy,2:6,{'veg2'});
x0.buy(idxMonths,idxOils) = [0;100;200;200;700];

[idxMonths,idxOils] = findindex(buy,2:6,{'non3'});
x0.buy(idxMonths,idxOils) = [0;250;250;250;750];
```

Check that the initial point is feasible. Because the constraints have different dimensions, set the `cellfun UniformOutput` name-value pair to `false` when checking the infeasibilities.

```
inf{1} = infeasibility(vegused,x0);
inf{2} = infeasibility(nonvegused,x0);
inf{3} = infeasibility(hardmin,x0);
inf{4} = infeasibility(hardmax,x0);
inf{5} = infeasibility(initstockbal,x0);
inf{6} = infeasibility(stockbal,x0);
inf{7} = infeasibility(minuse,x0);
```

```
inf{8} = infeasibility(maxusev,x0);
inf{9} = infeasibility(maxusenv,x0);
inf{10} = infeasibility(maxnuse,x0);
inf{11} = infeasibility(deflogic1,x0);
allinfeas = cellfun(@max,inf,'UniformOutput',false);
anyinfeas = cellfun(@max,allinfeas);
disp(anyinfeas)

     0     0     0     0     0     0     0     0     0     0     0
```

All of the infeasibilities are zero, which shows that the initial point is feasible.

Rerun the problem using the initial point.

```
[sol2,fval2,exitstatus2,output2] = solve(prob,x0,'options',opts)
```

```
Solving problem using intlinprog.
LP:                Optimal objective value is -1.075130e+05.

Cut Generation:    Applied 41 Gomory cuts, 2 cover cuts,
                   1 mir cut, and 1 clique cut.
                   Lower bound is -1.047522e+05.
                   Relative gap is 166.88%.

Branch and Bound:

   nodes      total   num int        integer       relative
 explored   time (s)  solution          fval        gap (%)
       15       0.18         2  -7.190185e+04   4.535003e+01
       21       0.21         3  -8.290000e+04   2.606702e+01
       25       0.21         4  -8.775370e+04   1.909426e+01
       30       0.22         5  -9.953889e+04   4.993907e+00
       56       0.25         6  -9.982870e+04   4.689100e+00
      138       0.31         7  -1.002787e+05   3.851839e+00
     1114       0.73         7  -1.002787e+05   0.000000e+00

Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the
optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon
variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the
default value).

sol2 = struct with fields:
      buy: [6x5 double]
   induse: [6x5 double]
     sell: [6x5 double]
    store: [6x5 double]


fval2 = 1.0028e+05

exitstatus2 =
    OptimalSolution


output2 = struct with fields:
      relativegap: 0
      absolutegap: 0
```

```
   numfeaspoints: 7
        numnodes: 1114
   constrviolation: 1.7580e-12
         message: 'Optimal solution found....'
          solver: 'intlinprog'
```

This time, `solve` took fewer branch-and-bound steps to find the solution.

```
fprintf(['Using the initial point took %d branch-and-bound steps,\nbut ',...
    'using no initial point took %d steps.'],output2.numnodes,output1.numnodes)
```

```
Using the initial point took 1114 branch-and-bound steps,
but using no initial point took 1112 steps.
```

Reference

[1] Williams, H. Paul. *Model Building in Mathematical Programming.* Fourth edition. J. Wiley, Chichester, England. Problem 12.1, "Food Manufacture1." 1999.

See Also
`findindex` | `solve`

More About
- "Named Index for Optimization Variables" on page 10-20
- "Problem-Based Optimization Workflow" on page 10-2

Quadratic Programming

Quadratic Programming Algorithms

In this section...
"Quadratic Programming Definition" on page 11-2
"interior-point-convex quadprog Algorithm" on page 11-2
"trust-region-reflective quadprog Algorithm" on page 11-7
"active-set quadprog Algorithm" on page 11-11

Quadratic Programming Definition

Quadratic programming is the problem of finding a vector x that minimizes a quadratic function, possibly subject to linear constraints:

$$\min_{x}\frac{1}{2}x^{T}Hx + c^{T}x \tag{11-1}$$

such that $A{\cdot}x \le b$, $Aeq{\cdot}x = beq$, $l \le x \le u$.

interior-point-convex quadprog Algorithm

The `interior-point-convex` algorithm performs the following steps:

1. "Presolve/Postsolve" on page 11-2
2. "Generate Initial Point" on page 11-3
3. "Predictor-Corrector" on page 11-3
4. "Stopping Conditions" on page 11-6
5. "Infeasibility Detection" on page 11-7

Note The algorithm has two code paths. It takes one when the Hessian matrix H is an ordinary (full) matrix of doubles, and it takes the other when H is a sparse matrix. For details of the sparse data type, see "Sparse Matrices". Generally, the algorithm is faster for large problems that have relatively few nonzero terms when you specify H as `sparse`. Similarly, the algorithm is faster for small or relatively dense problems when you specify H as `full`.

Presolve/Postsolve

The algorithm first tries to simplify the problem by removing redundancies and simplifying constraints. The tasks performed during the presolve step can include the following:

- 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 only one variable. If so, check for feasibility, and then change the linear constraint to a bound.
- Check if any linear equality constraint involves only 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 then delete the rows.

- Determine 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 then fix the variables at their appropriate bounds.
- Change any linear inequality constraints to linear equality constraints by adding slack variables.

If the 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, and if the presolve has not produced the solution, the algorithm continues to its next steps. After reaching a stopping criterion, the algorithm reconstructs the original problem KKT, 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.
4   Take a predictor step (see "Predictor-Corrector" on page 11-3), with minor corrections for feasibility, not a full predictor-corrector step. This places the initial point closer to the central path without entailing the overhead of a full predictor-corrector step. For details of the central path, see Nocedal and Wright [7], page 397.

Predictor-Corrector

The sparse and full interior-point-convex algorithms differ mainly in the predictor-corrector phase. The algorithms are similar, but differ in some details. For the basic algorithm description, see Mehrotra [47].

The algorithms begin by turning the linear inequalities Ax <= b into inequalities of the form Ax >= b by multiplying A and b by -1. This has no bearing on the solution, but makes the problem of the same form found in some literature.

- "Sparse Predictor-Corrector" on page 11-3
- "Full Predictor-Corrector" on page 11-5

Sparse Predictor-Corrector

Similar to the `fmincon` interior-point algorithm on page 6-30, the sparse `interior-point-convex` algorithm tries to find a point where the Karush-Kuhn-Tucker (KKT) on page 3-12 conditions hold. For the quadratic programming problem described in "Quadratic Programming Definition" on page 11-2, 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:

$$
\begin{pmatrix} H & 0 & -A_{eq}^T & -\bar{A}^T \\ A_{eq} & 0 & 0 & 0 \\ \bar{A} & -I & 0 & 0 \\ 0 & Z & 0 & S \end{pmatrix}
\begin{pmatrix} \Delta x \\ \Delta s \\ \Delta y \\ \Delta z \end{pmatrix}
= -
\begin{pmatrix} r_d \\ r_{eq} \\ r_{ineq} \\ r_{sz} \end{pmatrix}.
\tag{11-2}
$$

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.

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 [4].

Full Predictor-Corrector

The full predictor-corrector algorithm does not combine bounds into linear constraints, so it has another set of slack variables corresponding to the bounds. The algorithm shifts lower bounds to zero. And, if there is only one bound on a variable, the algorithm turns it into a lower bound of zero, by negating the inequality of an upper bound.

$\bar{A}$ 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.

The KKT conditions are

$$
\begin{aligned}
Hx + c - \bar{A}^T y - v + w &= 0 \\
\bar{A}x &= \bar{b} \\
x + t &= u \\
v_i x_i &= 0, \ i = 1, 2, \ldots, m \\
w_i t_i &= 0, \ i = 1, 2, \ldots, n \\
x, v, w, t &\geq 0.
\end{aligned}
$$

(11-3)

To find the solution x, slack variables and dual variables to "Equation 11-3", the algorithm basically considers a Newton-Raphson step:

$$
\begin{pmatrix}
H & -\bar{A}^T & 0 & -I & I \\
\bar{A} & 0 & 0 & 0 & 0 \\
-I & 0 & -I & 0 & 0 \\
V & 0 & 0 & X & 0 \\
0 & 0 & W & 0 & T
\end{pmatrix}
\begin{pmatrix}
\Delta x \\ \Delta y \\ \Delta t \\ \Delta v \\ \Delta w
\end{pmatrix}
= -
\begin{pmatrix}
Hx + c - \bar{A}^T y - v + w \\
\bar{A}x - \bar{b} \\
u - x - t \\
VX \\
WT
\end{pmatrix}
= -
\begin{pmatrix}
r_d \\ r_p \\ r_{ub} \\ r_{vx} \\ r_{wt}
\end{pmatrix},
$$

(11-4)

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

The algorithm solves "Equation 11-4" by first converting it to the symmetric matrix form

$$\begin{pmatrix} -D & \bar{A}^T \\ \bar{A} & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = -\begin{pmatrix} R \\ r_p \end{pmatrix}, \tag{11-5}$$

where

$$D = H + X^{-1}V + T^{-1}W$$
$$R = -r_d - X^{-1}r_{vx} + T^{-1}r_{wt} + T^{-1}Wr_{ub}.$$

All the matrix inverses in the definitions of D and R are simple to compute because the matrices are diagonal.

To derive "Equation 11-5" from "Equation 11-4", notice that the second row of "Equation 11-5" is the same as the second matrix row of "Equation 11-4". The first row of "Equation 11-5" comes from solving the last two rows of "Equation 11-4" for Δv and Δw, and then solving for Δt.

To solve "Equation 11-5", the algorithm follows the essential elements of Altman and Gondzio [1]. The algorithm solves the symmetric system by an LDL decomposition. As pointed out by authors such as Vanderbei and Carpenter [2], this decomposition is numerically stable without any pivoting, so can be fast.

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 [4].

The full `quadprog` predictor-corrector algorithm is largely the same as that in the `linprog` `'interior-point'` algorithm, but includes quadratic terms as well. See "Predictor-Corrector" on page 9-3.

References

[1] Altman, Anna and J. Gondzio. *Regularized symmetric indefinite systems in interior point methods for linear and quadratic optimization*. Optimization Methods and Software, 1999. Available for download here.

[2] Vanderbei, R. J. and T. J. Carpenter. *Symmetric indefinite systems for interior point methods*. Mathematical Programming 58, 1993. pp. 1–32. Available for download here.

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

$$\rho = \max\left(1, \|H\|, \|\bar{A}\|, \|A_{eq}\|, \|c\|, \|\bar{b}\|, \|b_{eq}\|\right)$$

The algorithm stops when all of these conditions are satisfied:

$$\|r_p\|_1 + \|r_{ub}\|_1 \leq \rho\text{TolCon}$$
$$\|r_d\|_\infty \leq \rho\text{TolFun}$$
$$r_c \leq \text{TolFun},$$

where

$$r_c = \max_i(\min(|x_i v_i|, |x_i|, |v_i|), \min(|t_i w_i|, |t_i|, |w_i|)).$$

r_c essentially measures the size of the complementarity residuals xv and tw, which are each vectors of zeros at a solution.

Infeasibility Detection

quadprog calculates a *merit function* φ at every iteration. The merit function is a measure of feasibility. 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 11-3. Use the following definitions:

$$\rho = \max\left(1, \|H\|, \|\bar{A}\|, \|A_{eq}\|, \|c\|, \|\bar{b}\|, \|b_{eq}\|\right)$$
$$r_{eq} = A_{eq}x - b_{eq}$$
$$r_{ineq} = \bar{A}x - \bar{b} - s$$
$$r_d = Hx + c + A_{eq}^T\lambda_{eq} + \bar{A}^T\bar{\lambda}_{ineq}$$
$$g = \frac{1}{2}x^T Hx + c^T x - \bar{b}^T\bar{\lambda}_{ineq} - b_{eq}^T\lambda_{eq}.$$

The notation $\bar{A}$ and $\bar{b}$ means the linear inequality coefficients, augmented with terms to represent bounds for the sparse algorithm. The notation $\bar{\lambda}_{ineq}$ similarly represents Lagrange multipliers for the linear inequality constraints, including bound constraints. This was called z in "Predictor-Corrector" on page 11-3, and λ_{eq} was called y.

The merit function φ is

$$\frac{1}{\rho}\left(\max\left(\|r_{eq}\|_\infty, \|r_{ineq}\|_\infty, \|r_d\|_\infty\right) + g\right).$$

If this merit function becomes too large, quadprog declares the problem to be infeasible and halts with exit flag -2.

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,

$$\min_{s}\{q(s),\ s \in N\}.\tag{11-6}$$

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

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

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

$$\min\left\{\frac{1}{2}s^{T}Hs + s^{T}g \ \text{ such that } \ \|Ds\| \leq \Delta\right\},\tag{11-7}$$

where g is the gradient of f at the current point x, H is the Hessian matrix (the symmetric matrix of second derivatives), D is a diagonal scaling matrix, Δ is a positive scalar, and $\| \cdot \|$ is the 2-norm. Good algorithms exist for solving "Equation 11-7" (see [48]); such algorithms typically involve the computation of all eigenvalues of H and a Newton process applied to the secular equation

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

Such algorithms provide an accurate solution to "Equation 11-7". 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 11-7", 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 11-7" is trivial even if full eigenvalue/eigenvector information is needed (since in the subspace, the problem is only two-dimensional). The dominant work has now shifted to the determination of the subspace.

The two-dimensional subspace S is determined with the aid of a preconditioned conjugate gradient process described below. The solver defines S as the linear space spanned by s_1 and s_2, where s_1 is in the direction of the gradient g, and s_2 is either an approximate Newton direction, i.e., a solution to

$$H \cdot s_2 = -g,\tag{11-8}$$

or a direction of negative curvature,

$$s_2^{T} \cdot H \cdot s_2 < 0.\tag{11-9}$$

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 11-7" 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 it encounters a direction of negative (or zero) curvature, that is, $d^THd \leq 0$. The PCG output direction *p* is either a direction of negative curvature or an approximate solution to the Newton system $Hp = -g$. In either case, *p* helps to 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

$$\min\{f(x) \ \text{ such that } \ Ax = b\}, \tag{11-10}$$

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 11-10" differs from the unconstrained approach in two significant ways. First, an initial feasible point x_0 is computed, using a sparse least-squares step, so that $Ax_0 = b$. Second, Algorithm PCG is replaced with Reduced Preconditioned Conjugate Gradients (RPCG), see [46], in order to compute an approximate reduced Newton step (or a direction of negative curvature in the null space of A). The key linear algebra step involves solving systems of the form

$$\begin{bmatrix} C & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \tag{11-11}$$

where $\tilde{A}$ approximates A (small nonzeros of A are set to zero provided rank is not lost) and C is a sparse symmetric positive-definite approximation to H, i.e., $C = H$. See [46] for more details.

Box Constraints

The box constrained problem is of the form

$$\min\{f(x) \ \text{such that} \ l \le x \le u\}, \tag{11-12}$$

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 11-12",

$$(D(x))^{-2}g = 0, \tag{11-13}$$

where

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

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 11-13" 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 11-13" is defined as the solution to the linear system

$$\widehat{M}Ds^N = -\widehat{g} \tag{11-14}$$

at the kth iteration, where

$$\widehat{g} = D^{-1}g = \mathrm{diag}\left(|v|^{1/2}\right)g, \tag{11-15}$$

and

$$\widehat{M} = D^{-1}HD^{-1} + \mathrm{diag}(g)J^v. \tag{11-16}$$

Here J^v plays the role of the Jacobian of $|v|$. Each diagonal component of the diagonal matrix J^v equals 0, –1, or 1. If all the components of l and u are finite, $J^v = \mathrm{diag}(\mathrm{sign}(g))$. At a point where $g_i = 0$, v_i might not be differentiable. $J^v_{ii} = 0$ is defined at such a point. Nondifferentiability of this type is not a cause for concern because, for such a component, it is not significant which value v_i takes. Further, $|v_i|$ will still be discontinuous at this point, but the function $|v_i| \cdot g_i$ is continuous.

Second, reflections are used to increase the 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

After completing a presolve step, the `active-set` algorithm proceeds in two phases.

- Phase 1 — Obtain a feasible point with respect to all constraints.
- Phase 2 — Iteratively lower the objective function while maintaining a list of the active constraints and maintaining feasibility in each iteration.

The `active-set` strategy (also known as a projection method) is similar to the strategy of Gill et al., described in [18] and [17].

Presolve Step

The algorithm first tries to simplify the problem by removing redundancies and simplifying constraints. The tasks performed during the presolve step can include the following:

- 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 only one variable. If so, check for feasibility, and then change the linear constraint to a bound.
- Check if any linear equality constraint involves only 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 then delete the rows.
- Determine 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 then fix the variables at their appropriate bounds.
- Change any linear inequality constraints to linear equality constraints by adding slack variables.

If the 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, and if the presolve has not produced the solution, the algorithm continues to its next steps. After reaching a stopping criterion, the algorithm reconstructs the original problem, undoing any presolve transformations. This final step is the postsolve step.

For details, see Gould and Toint [63].

Phase 1 Algorithm

In Phase 1, the algorithm attempts to find a point x that satisfies all constraints, with no consideration of the objective function. `quadprog` runs the Phase 1 algorithm only if the supplied initial point `x0` is infeasible.

To begin, the algorithm tries to find a point that is feasible with respect to all equality constraints, such as `X = Aeq\beq`. If there is no solution x to the equations `Aeq*x = beq`, then the algorithm halts. If there is a solution X, the next step is to satisfy the bounds and linear inequalities. In the case of no equality constraints set `X = x0`, the initial point.

Starting from X, the algorithm calculates `M = max(A*X − b, X - ub, lb − X)`. If `M <= options.ConstraintTolerance`, then the point X is feasible and the Phase 1 algorithm halts.

If `M > options.ConstraintTolerance`, the algorithm introduces a nonnegative slack variable γ for the auxiliary linear programming problem

$$\min_{x,\, \gamma} \gamma$$

such that

$$
\begin{aligned}
Ax - \gamma &\le b \\
\text{Aeq}\, x &= \text{beq} \\
\text{lb} - \gamma \le x &\le \text{ub} + \gamma \\
\gamma &\ge -\rho.
\end{aligned}
$$

Here, ρ is the `ConstraintTolerance` option multiplied by the absolute value of the largest element in A and Aeq. If the algorithm finds $\gamma = 0$ and a point x that satisfies the equations and inequalities, then x is a feasible Phase 1 point. If there is no solution to the auxiliary linear programming problem x with $\gamma = 0$, then the Phase 1 problem is infeasible.

To solve the auxiliary linear programming problem, the algorithm sets $\gamma_0 = M + 1$, sets $x_0 = X$, and initializes the active set as the fixed variables (if any) and all the equality constraints. The algorithm reformulates the linear programming variables p to be the offset of x from the current point x_0, namely $x = x_0 + p$. The algorithm solves the linear programming problem by the same iterations as it takes in Phase 2 to solve the quadratic programming problem, with an appropriately modified Hessian.

Phase 2 Algorithm

In terms of a variable d, the problem is

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

(11-17)

Here, A_i refers to the `ith` row of the m-by-n matrix A.

During Phase 2, an active set $\bar{A}_k$, which is an estimate of the active constraints (those on the constraint boundaries) at the solution point.

The algorithm updates $\bar{A}_k$ at each iteration k, forming the basis for a search direction d_k. Equality constraints always remain in the active set $\bar{A}_k$. The search direction d_k is calculated and minimizes the objective function while remaining on any active constraint boundaries. The algorithm forms the feasible subspace for d_k from a basis Z_k whose columns are orthogonal to the estimate of the active set $\bar{A}_k$ (that is, $\bar{A}_k Z_k = 0$). Therefore, 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 algorithm forms the matrix Z_k from the last $m - l$ columns of the QR decomposition of the matrix $\bar{A}_k^T$, where l is the number of active constraints and $l < m$. That is, Z_k is given by

$$Z_k = Q[:, l+1:m],\tag{11-18}$$

where

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

After finding Z_k, the algorithm looks for a new search direction d_k that minimizes $q(d)$, 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: $\widehat{d}_k = Z_k p$ for some vector p.

Viewing the quadratic as a function of p by substituting for d_k, gives

$$q(p) = \frac{1}{2} p^T Z_k^T H Z_k p + c^T Z_k p \,.\tag{11-19}$$

Differentiating this equation with respect to p yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c \,.\tag{11-20}$$

$\nabla q(p)$ 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 projected Hessian matrix $Z_k^T H Z_k$ is positive semidefinite, the minimum of the function $q(p)$ in the subspace defined by Z_k occurs when $\nabla q(p) = 0$, which is the solution of the system of linear equations

$$Z_k^T H Z_k p = -Z_k^T c \,.\tag{11-21}$$

The algorithm then takes a step of the form

$$x_{k+1} = x_k + \alpha d_k,$$

where

$$d_k = Z_k p \,.$$

Due to the quadratic nature of the objective function, only two choices of step length α exist at each iteration. A step of unity along d_k is the exact step to the minimum of the function restricted to the null space of $\bar{A}_k$. If the algorithm can take such a step without violating the constraints, then this step is the solution to the quadratic program ("Equation 6-32"). Otherwise, the step along d_k to the nearest constraint is less than unity, and the algorithm includes a new constraint in the active set at the next iteration. The distance to the constraint boundaries in any direction d_k is given by

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

which is defined for constraints not in the active set, and where the direction d_k is towards the constraint boundary, that is, $A_i d_k > 0, \ i = 1, \dots, m$.

When the active set includes n independent constraints, without location of the minimum, the algorithm calculates the Lagrange multipliers λ_k, which satisfy the nonsingular set of linear equations

$$\bar{A}_k^T \lambda_k = c + H x_k. \tag{11-22}$$

If all elements of λ_k are positive, x_k is the optimal solution of the quadratic programming problem "Equation 11-1". However, if any component of λ_k is negative, and the component does not correspond to an equality constraint, then the minimization is not complete. The algorithm deletes the element corresponding to the most negative multiplier and starts a new iteration.

Sometimes, when the solver finishes with all nonnegative Lagrange multipliers, the first-order optimality measure is above the tolerance, or the constraint tolerance is not met. In these cases, the solver attempts to reach a better solution by following the restart procedure described in [1]. In this procedure, the solver discards the current set of active constraints, relaxes the constraints a bit, and constructs a new set of active constraints while attempting to solve the problem in a manner that avoids cycling (repeatedly returning to the same state). If necessary, the solver can perform the restart procedure several times.

Note Do not try to stop the algorithm early by setting large values of the `ConstraintTolerance` and `OptimalityTolerance` options. Generally, the solver iterates without checking these values until it reaches a potential stopping point, and only then checks to see whether the tolerances are satisfied.

Occasionally, the `active-set` algorithm can have difficulty detecting when a problem is unbounded. This issue can occur if the direction of unboundedness v is a direction where the quadratic term $v'Hv$ = 0. For numerical stability and to enable a Cholesky factorization, the `active-set` algorithm adds a small, strictly convex term to the quadratic objective. This small term causes the objective function to be bounded away from –Inf. In this case, the `active-set` algorithm reaches an iteration limit instead of reporting that the solution is unbounded. In other words, the algorithm halts with exit flag `0` instead of –3.

References

[1] Gill, P. E., W. Murray, M. A. Saunders, and M. H. Wright. *A practical anti-cycling procedure for linearly constrained optimization.* Math. Programming 45 (1), August 1989, pp. 437–474.

Second-Order Cone Programming Algorithm

Definition of Second-Order Cone Programming

A second-order cone programming problem has the form

$$\min_x f^T x$$

subject to the constraints

$$\|A_{\mathrm{sc}}(i) \cdot x - b_{\mathrm{sc}}(i)\| \le d_{\mathrm{sc}}^T(i) \cdot x - \gamma(i)$$
$$A \cdot x \le b$$
$$\mathrm{Aeq} \cdot x = \mathrm{beq}$$
$$\mathrm{lb} \le x \le \mathrm{ub}.$$

f, x, b, beq, lb, and ub are vectors, and A and Aeq are matrices. For each i, the matrix $A_{\mathrm{sc}}(i)$, the vectors $b_{\mathrm{sc}}(i)$ and $d_{\mathrm{sc}}(i)$, and the scalar $\gamma(i)$ are in a second-order cone constraint that you create using `secondordercone`.

In other words, the problem has a linear objective function and linear constraints, as well as a set of second-order cone constraints of the form $\|A_{\mathrm{sc}}(i) \cdot x - b_{\mathrm{sc}}(i)\| \le d_{\mathrm{sc}}^T(i) \cdot x - \gamma(i)$.

coneprog Algorithm

The `coneprog` solver uses the algorithm described in Andersen, Roos, and Terlaky [1]. This method is an interior-point algorithm similar to the "Interior-Point linprog Algorithm" on page 9-2.

Standard Form

The algorithm starts by placing the problem in standard form. The algorithm adds nonnegative slack variables so that the problem has the form

$$\min_x f^T x$$

subject to the constraints

$$A \cdot x = b$$
$$x \in K.$$

The solver expands the sizes of the linear coefficient vector f and linear constraint matrix A to account for the slack variables.

The region K is the cross product of Lorentz cones "Equation 11-23" and the nonnegative orthant. To convert each convex cone

$$\|A_{\mathrm{sc}}(i) \cdot x - b_{\mathrm{sc}}(i)\| \le d_{\mathrm{sc}}^T(i) \cdot x - \gamma(i)$$

to a Lorentz cone "Equation 11-23", create a column vector of variables t_1, t_2, ..., t_{n+1}:

$$t_1 = d^T x - \gamma$$

$$t_{2:(n+1)} = A_{sc} x - b_{sc}.$$

Here, the number of variables n for each cone i is the number of rows in $A_{sc}(i)$. By its definition, the variable vector t satisfies the inequality

$$\|t_{2:(n+1)}\| \le t_1. \tag{11-23}$$

"Equation 11-23" is the definition of a Lorentz cone in $(n+1)$ variables. The variables t appear in the problem in place of the variables x in the convex region K.

Internally, the algorithm also uses a rotated Lorentz cone in the reformulation of cone constraints, but this topic does not address that case. For details, see Andersen, Roos, and Terlaky [1].

When adding slack variables, the algorithm negates variables, as needed, and adds appropriate constants so that:

- Variables with only one bound have a lower bound of zero.
- Variables with two bounds have a lower bound of zero and, using a slack variable, have no upper bound.
- Variables without bounds are placed in a Lorentz cone with a slack variable as the constrained variable. This slack variable is not part of any other expression, objective or constraint.

Dual Problem

The dual cone is

$$K_* = \left\{ s : s^T x \ge 0 \ \forall x \in K \right\}.$$

The dual problem is

$$\max_{y} b^T y$$

such that

$$A^T y + s = f$$

for some

$$s \in K_*.$$

A dual optimal solution is a point (y,s) that satisfies the dual constraints and maximizes the dual objective.

Homogeneous Self-Dual Formulation

To handle potentially infeasible or unbounded problems, the algorithm adds two more variables τ and κ and formulates the problem as homogeneous (equal to zero) and self-dual.

$$Ax - b\tau = 0$$

$$A^T y + s - f\tau = 0 \tag{11-24}$$

$$-f^T x + b^T y - \kappa = 0$$

along with the constraints

$$(x; \tau) \in \bar{K}, \quad (s; \kappa) \in \bar{K}_* \, . \tag{11-25}$$

Here, $\bar{K}$ is the cone K adjoined with the nonnegative real line, which is the space for $(x;\tau)$. Similarly $\bar{K}_*$ is the cone K_* adjoined with the nonnegative real line, which is the space for $(s;\kappa)$. In this formulation, the following lemma shows that τ is the scaling for feasible solutions, and κ is the indicator of an infeasible problem.

Lemma ([1] Lemma 2.1)

Let (x, τ, y, s, κ) be a feasible solution of "Equation 11-24" along with the constraints in "Equation 11-25".

- $x^T s + \tau \kappa = 0$.
- If $\tau > 0$, then $(x, y, s)/\tau$ is a primal-dual optimal solution of the standard form second-order cone problem.
- If $\kappa > 0$, then at least one of these strict inequalities holds:
  $b^T y > 0$
  $f^T x < 0$.

  If the first inequality holds, then the standard form, primal second-order cone problem is infeasible. If the second inequality holds, then the standard form, dual second-order cone problem is infeasible.

In summary, for feasible problems, the variable τ scales the solution between the original standard form problem and the homogeneous self-dual problem. For infeasible problems, the final iterate (x, y, s, τ, κ) provides a certificate of infeasibility for the original standard form problem.

Start Point

The start point for the iterations is the feasible point:

- $x = 1$ for each nonnegative variable, 1 for the first variable in each Lorentz cone, and 0 otherwise.
- $y = 0$.
- $s = (1,0,...,0)$ for each cone, 1 for each nonnegative variable.
- $\tau = 1$.
- $\kappa = 1$.

Central Path

The algorithm attempts to follow the central path, which is the parameterized solution to the following equations for γ decreasing from 1 toward 0.

$$Ax - b\tau = \gamma(Ax_0 - b\tau_0)$$
$$A^T y + s - c\tau = \gamma\left(A^T y_0 + s_0 - f\tau_0\right)$$
$$-f^T x + b^T y - \kappa = \gamma\left(-f^T x_0 + b^T y_0 - \kappa_0\right) \tag{11-26}$$
$$XSe = \gamma\mu_0 e$$
$$\tau\kappa = \gamma\mu_0 \, .$$

- Each variable with a 0 subscript indicates the start point of the variable.
- The variables X and S are arrow head matrices formed from the x and s vectors, respectively. For a vector $x = [x_1, x_2, \ldots, x_n]$, the arrow head matrix X has the definition

$$X = \text{mat}(x) = \begin{bmatrix} x_1 & x_{2:n}^T \\ x_{2:n} & x_1 I \end{bmatrix}.$$

By its definition, X is symmetric.

- The variable e is the vector with a 1 in each cone coordinate corresponding to the x_1 Lorentz cone coordinate.
- The variable μ_0 has the definition

$$\mu_0 = \frac{x_0^T s_0 + \tau_0 \kappa_0}{k + 1},$$

where k is the number of nonzero elements in x_0.

The central path begins at the start point and ends at an optimal solution to the homogeneous self-dual problem.

Andersen, Roos, and Terlaky [1] show in Lemma 3.1 that the complementarity condition $x^T s = 0$, where x and s are in a product of Lorentz cones L, is equivalent to the condition

$$X_i S_i e_i = S_i X_i e_i = 0$$

for every cone i. Here $X_i = \text{mat}(x_i)$, x_i is the variable associated with the Lorentz cone i, $S_i = \text{mat}(s_i)$, and e_i is the unit vector $[1,0,0,\ldots,0]$ of the appropriate dimension. This discussion shows that the central path satisfies the complementarity condition at its end point.

Search Direction

To obtain points near the central path as the parameter γ decreases from 1 toward 0, the algorithm uses Newton's method. The variables to find are labeled (x, τ, y, s, κ). Let d_x represent the search direction for the x variables, and so on. Then the Newton step solves the following linear system, derived from "Equation 11-26".

$$A d_x - b d_\tau = (\gamma - 1)(A x_0 - b \tau_0)$$

$$A^T d_y + d_s - f d_\tau = (\gamma - 1)\left(A^T y_0 + s_0 - f \tau_0\right)$$

$$-f^T d_x + b^T d_y - d_\kappa = (\gamma - 1)\left(-f^T x_0 + b^T y_0 - \kappa\right)$$

$$X_0 d_s + S_0 d_x = -X_0 S_0 e + \gamma \mu_0 e$$

$$\tau_0 d_\kappa + \kappa_0 d_t au = -\tau_0 \kappa_0 + \gamma \mu_0.$$

The algorithm obtains its next point by taking a step in the d direction.

$$\begin{bmatrix} x_1 \\ \tau_1 \\ y_1 \\ s_1 \\ \kappa_1 \end{bmatrix} = \begin{bmatrix} x_0 \\ \tau_0 \\ y_0 \\ s_0 \\ \kappa_0 \end{bmatrix} + \alpha \begin{bmatrix} d_x \\ d_\tau \\ d_y \\ d_s \\ d_\kappa \end{bmatrix}$$

for some step $\alpha \in [0, 1]$.

For both numerical stability and accelerated convergence, the algorithm scales the step according to a suggestion in Nesterov and Todd [5]. Also, the algorithm corrects the step according to a variant of Mehrotra's predictor-corrector [4]. (For further details, see Andersen, Roos, and Terlaky [1].)

Iterative Display and Stopping Conditions

At each iteration k, the algorithm computes three relative convergence measures:

- Primal infeasibility

$$\text{Infeas}^k_{\text{Primal}} = \frac{\|Ax_k - b\tau_k\|}{\max(1, \|Ax_0 - b\tau_0\|)} .$$

- Dual infeasibility

$$\text{Infeas}^k_{\text{Dual}} = \frac{\|A^T y_k + s_k - f\tau_k\|}{\max\left(1, \|A^T y_0 + s_0 - f\tau_0\|\right)} .$$

- Gap infeasibility

$$\text{Infeas}^k_{\text{Gap}} = \frac{\left|-f^T x_k + b^T y_k - \kappa_k\right|}{\max\left(1, \left|-f^T x_0 + b^T y_0 - \kappa_0\right|\right)} .$$

You can view these three statistics at the command line by specifying iterative display.

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

All three should approach zero when the problem is feasible and the solver converges. For a feasible problem, the variable κ_k approaches zero, and the variable τ_k approaches a positive constant.

One stopping condition is somewhat related to the gap infeasibility. The stopping condition is when the following optimality measure decreases below the optimality tolerance.

$$\text{Optimality}^k = \frac{\left|f^T x_k - b^T y_k\right|}{\tau_k + \left|b^T y_k\right|} = \frac{\left|f^T x_k/\tau_k - b^T y_k/\tau_k\right|}{1 + \left|b^T y_k/\tau_k\right|} .$$

This statistic measures the precision of the objective value.

The solver also stops and declares the problem to be infeasible under the following conditions. The three relative infeasibility measures are less than $c = \texttt{ConstraintTolerance}$, and

$$\tau_k \le c \max(1, \kappa_k) .$$

If $b^T y_k > 0$, then the solver declares that the primal problem is infeasible. If $f^T x_k < 0$, then the solver declares that the dual problem is infeasible.

The algorithm also stops when

$$\mu_k \le c\mu_0$$

and

$$\tau_k \leq c \max(1, \kappa_k).$$

In this case, `coneprog` reports that the problem is numerically unstable (exit flag -10).

The remaining stopping condition occurs when at least one infeasibility measure is greater than `ConstraintTolerance` and the computed step size is too small. In this case, `coneprog` reports that the search direction became too small and no further progress could be made (exit flag -7).

References

[1] Andersen, E. D., C. Roos, and T. Terlaky. *On implementing a primal-dual interior-point method for conic quadratic optimization.* Math. Program., Ser. B **95**, pp. 249–277 (2003). https://doi.org/10.1007/s10107-002-0349-3

[2] Ben-Tal, Aharon, and Arkadi Nemirovski. *Convex Optimization in Engineering: Modeling, Analysis, Algorithms.* (1998). Available at http://wwwtmp.st.ewi.tudelft.nl/roos/courses/wi485/tud00r.pdf.

[3] Luo, Zhi-Quan, Jos F. Sturm, and Shuzhong Zhang. *Duality and Self-Duality for Conic Convex Programming.* (1996). Available at https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.6432

[4] Mehrotra, Sanjay. "On the Implementation of a Primal-Dual Interior Point Method." *SIAM Journal on Optimization 2*, no. 4 (November 1992): 575–601. https://doi.org/10.1137/0802028.

[5] Nesterov, Yu. E., and M. J. Todd. "Self-Scaled Barriers and Interior-Point Methods for Convex Programming." *Mathematics of Operations Research 22*, no. 1 (February 1997): 1–42. https://doi.org/10.1287/moor.22.1.1.

See Also
coneprog | secondordercone

More About
- "Quadratic Programming and Cone Programming"

Quadratic Minimization with Bound Constraints

This example shows the effects of some option settings on a sparse, bound-constrained, positive definite quadratic problem.

Create the quadratic matrix H as a tridiagonal symmetric matrix of size 400-by-400 with entries +4 on the main diagonal and –2 on the off-diagonals.

```
Bin = -2*ones(399,1);
H = spdiags(Bin,-1,400,400);
H = H + H';
H = H + 4*speye(400);
```

Set bounds of [0,0.9] in each component except the 400th. Allow the 400th component to be unbounded.

```
lb = zeros(400,1);
lb(400) = -inf;
ub = 0.9*ones(400,1);
ub(400) = inf;
```

Set the linear vector f to zeros, except set f(400) = –2.

```
f = zeros(400,1);
f(400) = -2;
```

Trust-Region-Reflective Solution

Solve the quadratic program using the 'trust-region-reflective' algorithm.

```
options = optimoptions('quadprog','Algorithm',"trust-region-reflective");
tic
[x1,fval1,exitflag1,output1] = ...
        quadprog(H,f,[],[],[],[],lb,ub,[],options);
```

Local minimum possible.

quadprog stopped because the relative change in function value is less than the function toleran

```
time1 = toc
```

time1 = 0.0836

Examine the solution.

```
fval1,exitflag1,output1.iterations,output1.cgiterations
```

fval1 = -0.9930

exitflag1 = 3

ans = 18

ans = 1682

The algorithm converges in relatively few iterations, but takes over a thousand CG iterations. To avoid the CG iterations, set options to use a direct solver instead.

```
options = optimoptions(options,'SubproblemAlgorithm','factorization');
tic
```

```
[x2,fval2,exitflag2,output2] = ...
        quadprog(H,f,[],[],[],[],lb,ub,[],options);
```

```
Local minimum possible.

quadprog stopped because the relative change in function value is less than the function tolerance

time2 = toc

time2 = 0.0180

fval2,exitflag2,output2.iterations,output2.cgiterations

fval2 = -0.9930

exitflag2 = 3

ans = 10

ans = 0
```

This time, the algorithm takes fewer iterations, and no CG iterations. The solution time decreases substantially, despite the relatively time-consuming direct factorization steps, because the solver avoids taking many CG steps.

Interior-Point Solution

The default `'interior-point-convex'` algorithm can solve this problem.

```
tic
[x3,fval3,exitflag3,output3] = ...
    quadprog(H,f,[],[],[],[],lb,ub); % No options means use the default algorithm
```

```
Minimum found that satisfies the constraints.

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

<stopping criteria details>

time3 = toc

time3 = 0.0435

fval3,exitflag3,output3.iterations

fval3 = -0.9930

exitflag3 = 1

ans = 8
```

Compare Results

All algorithms give the same objective function value to display precision, –0.9930.

The `'interior-point-convex'` algorithm takes the fewest iterations. However, the `'trust-region-reflective'` algorithm with direct subproblem solver reaches the solution fastest.

```matlab
tt = table([time1;time2;time3],[output1.iterations;output2.iterations;output3.iterations],...
    'VariableNames',["Time" "Iterations"],'RowNames',["TRR" "TRR Direct" "IP"])
```

tt=*3×2 table*

	Time	Iterations
TRR	0.083564	18
TRR Direct	0.018008	10
IP	0.04352	8

Quadratic Minimization with Dense, Structured Hessian

In this section...
"Take advantage of a structured Hessian" on page 11-24
"Step 1: Decide what part of H to pass to quadprog as the first argument." on page 11-24
"Step 2: Write a function to compute Hessian-matrix products for H." on page 11-24
"Step 3: Call a quadratic minimization routine with a starting point." on page 11-25
"Preconditioning" on page 11-26

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 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 11-26).

```
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 `HessianMultiplyFcn` 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 'HessianMultiplyFcn' 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 HessianMultiplyFcn option
options = optimoptions(@quadprog,'Algorithm','trust-region-reflective','HessianMultiplyFcn',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`, `output.iterations`, and `output.cgiterations`.

```
fval,exitflag,output.iterations, output.cgiterations

fval =

  -1.0538e+03


exitflag =

     3


ans =

    18


ans =

    30
```

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 'HessianMultiplyFcn' 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 HessianMultiplyFcn option, and override the TolPCG option
```

```
options = optimoptions(@quadprog,'Algorithm','trust-region-reflective',...
    'HessianMultiplyFcn',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
```

and the first-order optimality is essentially the same.

```
output.firstorderopt
ans =
    0.0043
```

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

See Also

More About

- "Jacobian Multiply Function with Linear Least Squares" on page 12-30

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, and a few nonzero diagonal bands. 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]. H is a sparse symmetric banded matrix.
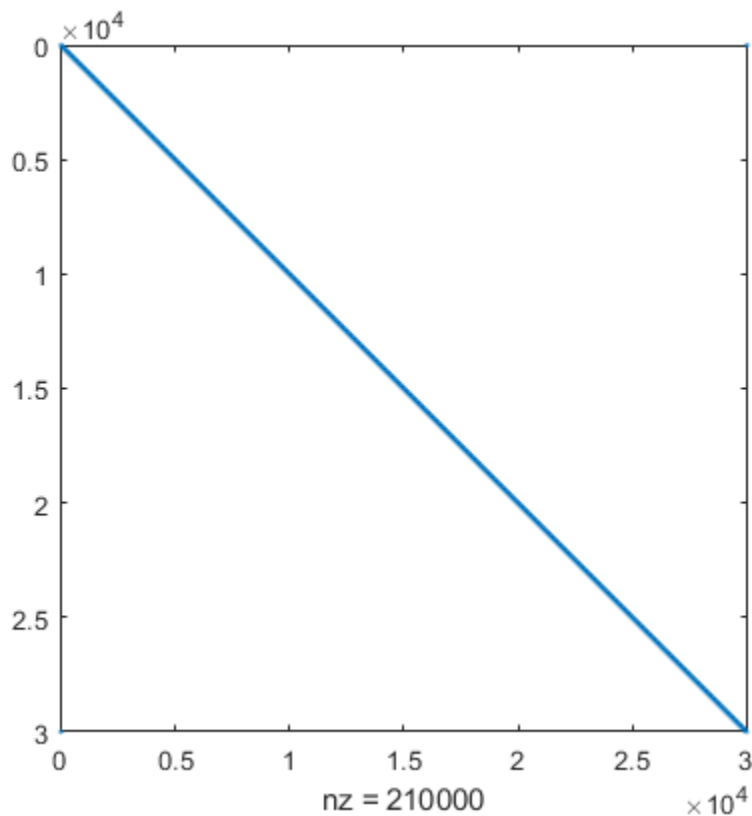
Create Sparse Quadratic Matrix

Create a symmetric circulant matrix based on shifts of the vector [3,6,2,14,2,6,3], with 14 being on the main diagonal. Have the matrix be n-by-n, where n = 30,000.

```
n = 3e4;
H2 = speye(n);
H = 3*circshift(H2,-3,2) + 6*circshift(H2,-2,2) + 2*circshift(H2,-1,2)...
    + 14*H2 + 2*circshift(H2,1,2) + 6*circshift(H2,2,2) + 3*circshift(H2,3,2);
```

View the matrix structure.

```
spy(H)
```

Create Linear Constraint and Objective

The linear constraint is that the sum of the solution elements is nonpositive. The objective function contains a linear term expressed in the vector f.

```
A = ones(1,n);
b = 0;
f = 1:n;
f = -f;
```

Solve Problem

Solve the quadratic programming problem using the `'interior-point-convex'` algorithm. To keep the solver from stopping prematurely, set the `StepTolerance` option to 0.

```
options = optimoptions(@quadprog,'Algorithm','interior-point-convex','StepTolerance',0);
[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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

<stopping criteria details>
```

On many computers you cannot create a full n-by-n matrix when n = 30,000. So you can run this problem only by using sparse matrices.

Examine Solution

View the objective function value, number of iterations, and Lagrange multiplier associated with linear inequality.

```
fprintf('The objective function value is %d.\nThe number of iterations is %d.\nThe Lagrange mult:
    fval,output.iterations,lambda.ineqlin)
```

```
The objective function value is -3.133073e+10.
The number of iterations is 7.
The Lagrange multiplier is 1.500050e+04.
```

Because there are no lower bounds, upper bounds, or linear equality constraints, the only meaningful Lagrange multiplier is `lambda.ineqlin`. Because `lambda.ineqlin` is nonzero, you can tell that the inequality constraint is active. Evaluate the constraint to see that the solution is on the boundary.

```
fprintf('The linear inequality constraint A*x has value %d.\n',A*x)
```

```
The linear inequality constraint A*x has value 9.150244e-08.
```

The sum of the solution components is zero to within tolerances.

The solution x has three regions: an initial portion, a final portion, and an approximately linear portion over most of the solution. Plot the three regions.

```
subplot(3,1,1)
plot(x(1:60))
title('x(1) Through x(60)')
```

```
subplot(3,1,2)
plot(x(61:n-60))
title('x(61) Through x(n-60)')
subplot(3,1,3)
plot(x(n-59:n))
title('x(n-59) Through x(n)')
```
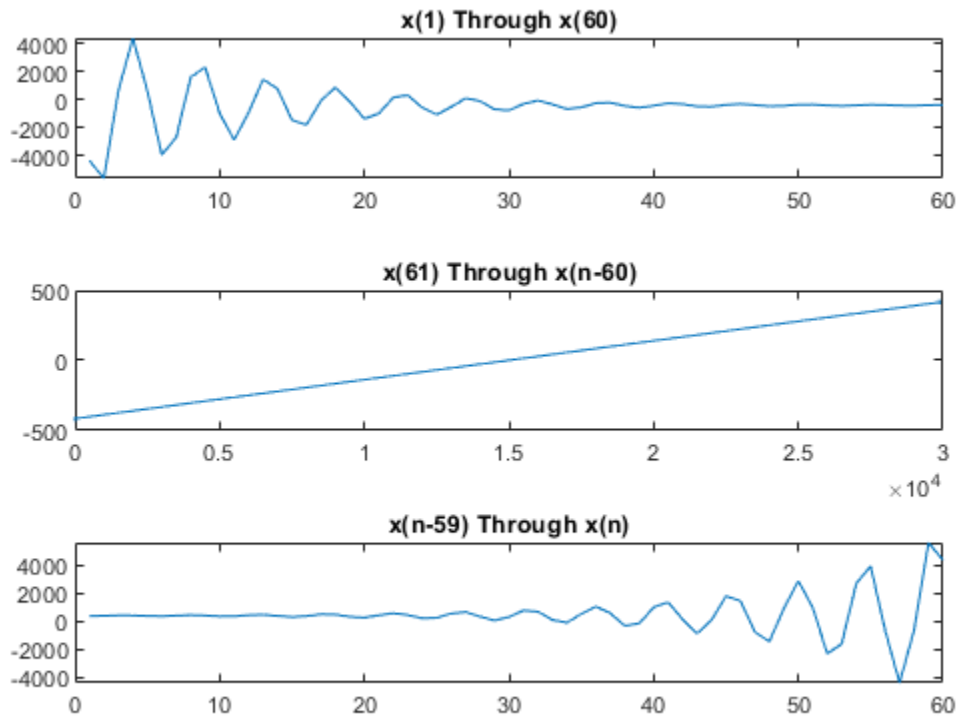

See Also
circshift | quadprog

More About

- "Sparse Matrices"

Bound-Constrained Quadratic Programming, Solver-Based

This example shows how to determine the shape of a circus tent by solving a quadratic optimization problem. The tent is formed from heavy, elastic material, and settles into a shape that has minimum potential energy subject to constraints. A discretization of the problem leads to a bound-constrained quadratic programming problem.

For a problem-based version of this example, see "Bound-Constrained Quadratic Programming, Problem-Based" on page 11-47.

Problem Definition

Consider building a circus tent to cover a square lot. The tent has five poles covered with a heavy, elastic material. The problem is to find the natural shape of the tent. Model the shape as the height $x(p)$ of the tent at position p.

The potential energy of heavy material lifted to height x is cx, for a constant c that is proportional to the weight of the material. For this problem, choose $c = 1/3000$.

The elastic potential energy of a piece of the material E_{stretch} is approximately proportional to the second derivative of the material height, times the height. You can approximate the second derivative by the 5-point finite difference approximation (assume that the finite difference steps are of size 1). Let Δx represent a shift of 1 in the first coordinate direction, and Δy represent a shift by 1 in the second coordinate direction.

$$E_{\text{stretch}}(p) = \left(-1\left(x(p + \Delta_x) + x(p - \Delta_x) + x(p + \Delta_y) + x(p - \Delta_y)\right) + 4x(p)\right)x(p).$$

The natural shape of the tent minimizes the total potential energy. By discretizing the problem, you find that the total potential energy to minimize is the sum over all positions p of $E_{\text{stretch}}(p) + cx(p)$.

This potential energy is a quadratic expression in the variable x.

Specify the boundary condition that the height of the tent at the edges is zero. The tent poles have a cross section of 1-by-1 unit, and the tent has a total size of 33-by-33 units. Specify the height and location of each pole. Plot the square lot region and tent poles.

```
height = zeros(33);
height(6:7,6:7) = 0.3;
height(26:27,26:27) = 0.3;
height(6:7,26:27) = 0.3;
height(26:27,6:7) = 0.3;
height(16:17,16:17) = 0.5;
colormap(gray);
surfl(height)
axis tight
view([-20,30]);
title('Tent Poles and Region to Cover')
```

Tent Poles and Region to Cover

Create Boundary Conditions

The `height` matrix defines the lower bounds on the solution x. To restrict the solution to be zero at the boundary, set the upper bound `ub` to be zero on the boundary.

```
boundary = false(size(height));
boundary([1,33],:) = true;
boundary(:,[1,33]) = true;
ub = inf(size(boundary)); % No upper bound on most of the region
ub(boundary) = 0;
```

Create Objective Function Matrices

The `quadprog` problem formulation is to minimize

$$\frac{1}{2}x^{T}Hx + f^{T}x.$$

In this case, the linear term $f^{T}x$ corresponds to the potential energy of the material height. Therefore, specify $f = 1/3000$ for each component of x.

```
f = ones(size(height))/3000;
```

Create the finite difference matrix representing E_{stretch} by using the `delsq` function. The `delsq` function returns a sparse matrix with entries of 4 and -1 corresponding to the entries of 4 and -1 in the formula for $E_{\text{stretch}}(p)$. Multiply the returned matrix by 2 to have `quadprog` solve the quadratic program with the energy function as given by E_{stretch}.

```
H = delsq(numgrid('S',33+2))*2;
```

View the structure of the matrix H. The matrix operates on x(:), which means the matrix x is converted to a vector by linear indexing.

```
spy(H);
title('Sparsity Structure of Hessian Matrix');
```
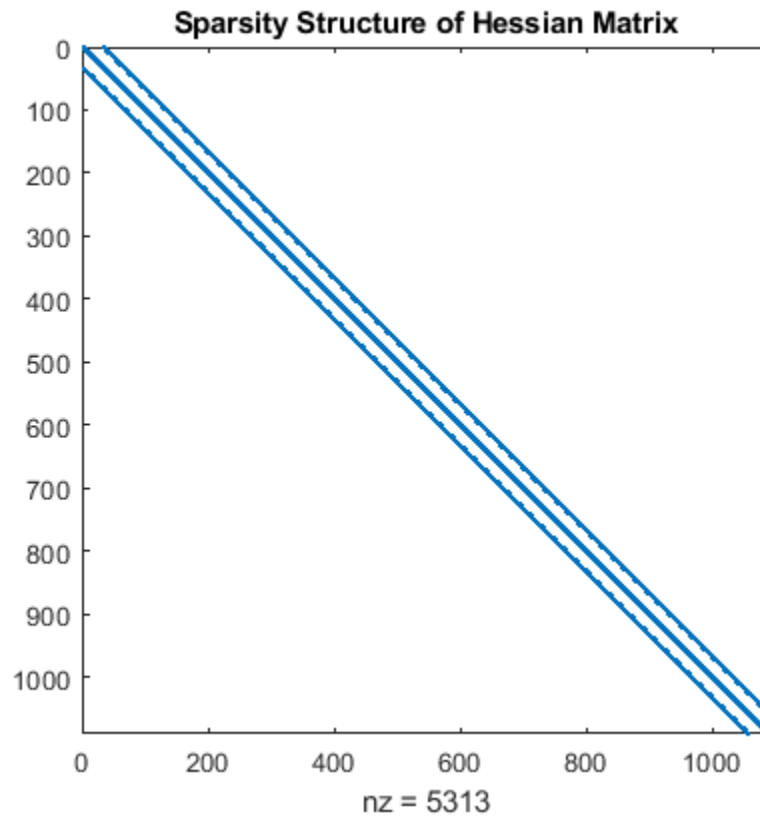

Run Optimization Solver

Solve the problem by calling quadprog.

```
x = quadprog(H,f,[],[],[],[],height,ub);
```

Minimum found that satisfies the constraints.

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

Plot Solution

Reshape the solution x to a matrix S. Then plot the solution.

```
S = reshape(x,size(height));
surfl(S);
axis tight;
view([-20,30]);
```

See Also

More About

- "Bound-Constrained Quadratic Programming, Problem-Based" on page 11-47

Quadratic Programming for Portfolio Optimization Problems, Solver-Based

This example shows how to solve portfolio optimization problems using the interior-point quadratic programming algorithm in `quadprog`. The function `quadprog` belongs to Optimization Toolbox™.

The matrices that define the problems in this example are dense; however, the interior-point algorithm in `quadprog` can also exploit sparsity in the problem matrices for increased speed. For a sparse example, see "Large Sparse Quadratic Program with Interior Point Algorithm" on page 11-28.

The Quadratic Model

Suppose that there are n different assets. The rate of return of asset i is a random variable with expected value m_i. The problem is to find what fraction x_i to invest in each asset i in order to minimize risk, subject to a specified minimum expected rate of return.

Let C denote the covariance matrix of rates of asset returns.

The classical mean-variance model consists of minimizing portfolio risk, as measured by

$$\frac{1}{2}x^T C x$$

subject to a set of constraints.

The expected return should be no less than a minimal rate of portfolio return r that the investor desires,

$$\sum_{i=1}^{n} m_i \, x_i \geq r,$$

the sum of the investment fractions x_i's should add up to a total of one,

$$\sum_{i=1}^{n} x_i = 1,$$

and, being fractions (or percentages), they should be numbers between zero and one,

$$0 \leq x_i \leq 1, \quad i = 1 \ldots n.$$

Since the objective to minimize portfolio risk is quadratic, and the constraints are linear, the resulting optimization problem is a quadratic program, or QP.

225-Asset Problem

Let us now solve the QP with 225 assets. The dataset is from the OR-Library [Chang, T.-J., Meade, N., Beasley, J.E. and Sharaiha, Y.M., "Heuristics for cardinality constrained portfolio optimisation" Computers & Operations Research 27 (2000) 1271-1302].

We load the dataset and then set up the constraints in a format expected by `quadprog`. In this dataset the rates of return m_i range between -0.008489 and 0.003971; we pick a desired return r in between, e.g., 0.002 (0.2 percent).

Load dataset stored in a MAT-file.

```
load('port5.mat','Correlation','stdDev_return','mean_return')
```

Calculate covariance matrix from correlation matrix.

```
Covariance = Correlation .* (stdDev_return * stdDev_return');
nAssets = numel(mean_return); r = 0.002;      % number of assets and desired return
Aeq = ones(1,nAssets); beq = 1;               % equality Aeq*x = beq
Aineq = -mean_return'; bineq = -r;            % inequality Aineq*x <= bineq
lb = zeros(nAssets,1); ub = ones(nAssets,1); % bounds lb <= x <= ub
c = zeros(nAssets,1);                         % objective has no linear term; set it to zero
```

Select the Interior Point Algorithm in Quadprog

In order solve the QP using the interior-point algorithm, we set the option Algorithm to 'interior-point-convex'.

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

Solve 225-Asset Problem

We now set some additional options, and call the solver quadprog.

Set additional options: turn on iterative display, and set a tighter optimality termination tolerance.

```
options = optimoptions(options,'Display','iter','TolFun',1e-10);
```

Call solver and measure wall-clock time.

```
tic
[x1,fval1] = quadprog(Covariance,c,Aineq,bineq,Aeq,beq,lb,ub,[],options);
toc
```

```
 Iter            Fval  Primal Infeas    Dual Infeas  Complementarity
    0    2.384401e+01   2.253410e+02   1.337381e+00     1.000000e+00
    1    1.338822e-03   7.394864e-01   4.388791e-03     1.038098e-02
    2    1.186079e-03   6.443975e-01   3.824446e-03     8.727381e-03
    3    5.923977e-04   2.730703e-01   1.620650e-03     1.174211e-02
    4    5.354880e-04   5.303581e-02   3.147632e-04     1.549549e-02
    5    5.181994e-04   2.651791e-05   1.573816e-07     2.848171e-04
    6    5.066191e-04   9.285375e-06   5.510794e-08     1.041224e-04
    7    3.923090e-04   7.619855e-06   4.522322e-08     5.536006e-04
    8    3.791545e-04   1.770065e-06   1.050519e-08     1.382075e-04
    9    2.923749e-04   8.850312e-10   5.252597e-12     3.858983e-05
   10    2.277722e-04   4.431646e-13   2.626591e-15     6.204101e-06
   11    1.992243e-04   1.002236e-15   6.938894e-18     4.391483e-07
   12    1.950468e-04   6.661338e-16   6.938894e-18     1.429441e-08
   13    1.949141e-04   6.670012e-16   6.938894e-18     9.731942e-10
   14    1.949121e-04   4.458239e-16   6.938894e-18     2.209702e-12

Minimum found that satisfies the constraints.

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

Elapsed time is 0.605831 seconds.
```

Plot results.
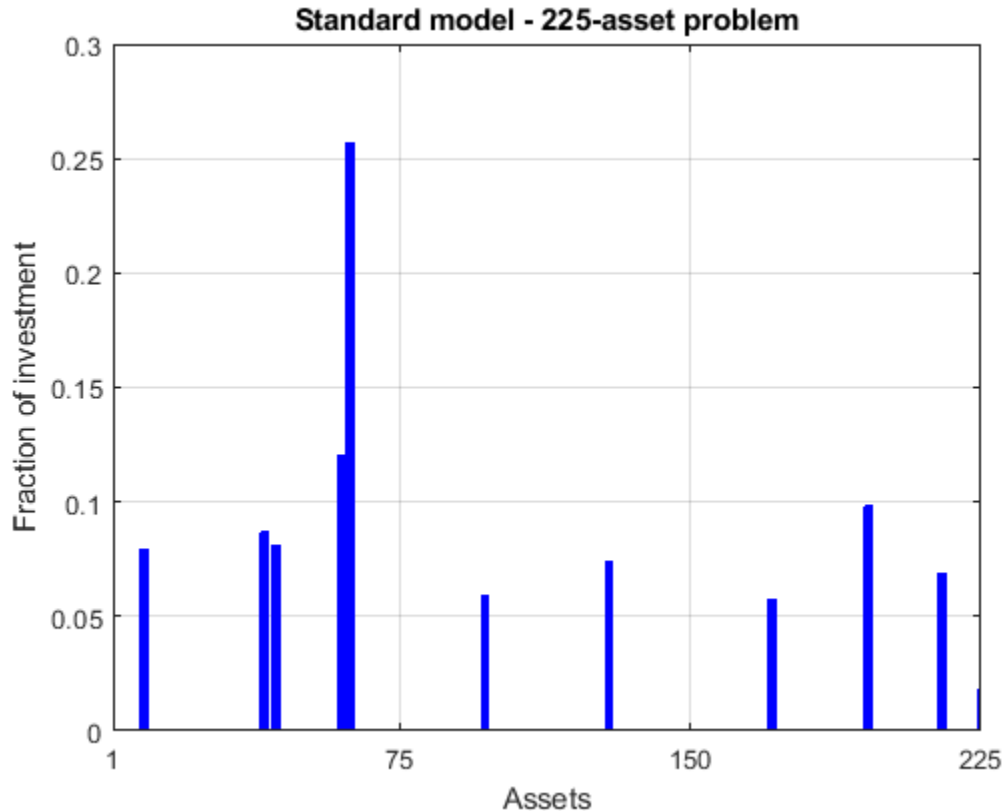
```
plotPortfDemoStandardModel(x1)
```


225-Asset Problem with Group Constraints

We now add to the model group constraints that require that 30% of the investor's money has to be invested in assets 1 to 75, 30% in assets 76 to 150, and 30% in assets 151 to 225. Each of these groups of assets could be, for instance, different industries such as technology, automotive, and pharmaceutical. The constraints that capture this new requirement are

$$\sum_{i=1}^{75} x_i \geq 0.3, \qquad \sum_{i=76}^{150} x_i \geq 0.3, \qquad \sum_{i=151}^{225} x_i \geq 0.3.$$

Add group constraints to existing equalities.

```
Groups = blkdiag(ones(1,nAssets/3),ones(1,nAssets/3),ones(1,nAssets/3));
Aineq = [Aineq; -Groups];         % convert to <= constraint
bineq = [bineq; -0.3*ones(3,1)];  % by changing signs
```

Call solver and measure wall-clock time.

```
tic
[x2,fval2] = quadprog(Covariance,c,Aineq,bineq,Aeq,beq,lb,ub,[],options);
toc
```

```
Iter          Fval    Primal Infeas     Dual Infeas   Complementarity
   0   2.384401e+01    4.464410e+02    1.337324e+00      1.000000e+00
   1   1.346872e-03    1.474737e+00    4.417606e-03      3.414918e-02
   2   1.190113e-03    1.280566e+00    3.835962e-03      2.934585e-02
   3   5.990845e-04    5.560762e-01    1.665738e-03      1.320038e-02
   4   3.890097e-04    2.780381e-04    8.328691e-07      7.287370e-03
   5   3.887354e-04    1.480950e-06    4.436214e-09      4.641988e-05
   6   3.387787e-04    8.425389e-07    2.523842e-09      2.578178e-05
   7   3.089240e-04    2.707587e-07    8.110631e-10      9.217509e-06
   8   2.639458e-04    6.586817e-08    1.973094e-10      6.509001e-06
   9   2.252657e-04    2.225507e-08    6.666550e-11      6.783212e-06
  10   2.105838e-04    5.811528e-09    1.740855e-11      1.967570e-06
  11   2.024362e-04    4.129112e-12    1.237009e-14      5.924109e-08
  12   2.009703e-04    4.675080e-15    1.404388e-17      6.353270e-10
  13   2.009650e-04    2.220446e-16    6.938894e-18      1.596041e-13

Minimum found that satisfies the constraints.

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

Elapsed time is 0.351748 seconds.
```

Plot results, superimposed on results from previous problem.

```
plotPortfDemoGroupModel(x1,x2);
```
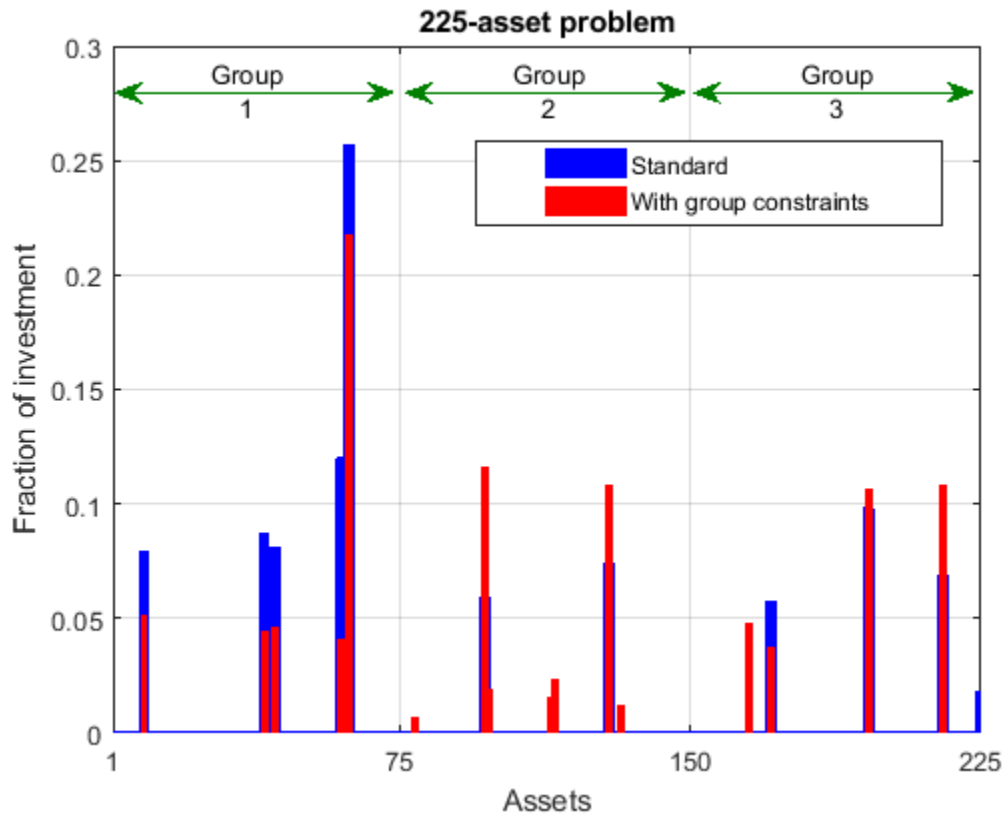
Summary of Results So Far

We see from the second bar plot that, as a result of the additional group constraints, the portfolio is now more evenly distributed across the three asset groups than the first portfolio. This imposed diversification also resulted in a slight increase in the risk, as measured by the objective function (see column labeled "f(x)" for the last iteration in the iterative display for both runs).

1000-Asset Problem Using Random Data

In order to show how `quadprog`'s interior-point algorithm behaves on a larger problem, we'll use a 1000-asset randomly generated dataset. We generate a random correlation matrix (symmetric, positive-semidefinite, with ones on the diagonal) using the `gallery` function in MATLAB®.

Reset random stream for reproducibility.

```
rng(0,'twister');
nAssets = 1000; % desired number of assets
```

Generate means of returns between -0.1 and 0.4.

```
a = -0.1; b = 0.4;
mean_return = a + (b-a).*rand(nAssets,1);
```

Generate standard deviations of returns between 0.08 and 0.6.

```
a = 0.08; b = 0.6;
stdDev_return = a + (b-a).*rand(nAssets,1);
% Correlation matrix, generated using Correlation = gallery('randcorr',nAssets).
% (Generating a correlation matrix of this size takes a while, so we load
% a pre-generated one instead.)
load('correlationMatrixDemo.mat','Correlation');
% Calculate covariance matrix from correlation matrix.
Covariance = Correlation .* (stdDev_return * stdDev_return');
```

Define and Solve Randomly Generated 1000-Asset Problem

We now define the standard QP problem (no group constraints here) and solve.

```
r = 0.15;                                    % desired return
Aeq = ones(1,nAssets); beq = 1;              % equality Aeq*x = beq
Aineq = -mean_return'; bineq = -r;           % inequality Aineq*x <= bineq
lb = zeros(nAssets,1); ub = ones(nAssets,1); % bounds lb <= x <= ub
c = zeros(nAssets,1);                        % objective has no linear term; set it to zero
```

Call solver and measure wall-clock time.

```
tic
x3 = quadprog(Covariance,c,Aineq,bineq,Aeq,beq,lb,ub,[],options);
toc
```

```
 Iter          Fval  Primal Infeas    Dual Infeas  Complementarity
    0   7.083800e+01   1.142266e+03   1.610094e+00     1.000000e+00
    1   5.603619e-03   7.133717e+00   1.005541e-02     9.857295e-02
    2   1.076070e-04   3.566858e-03   5.027704e-06     9.761758e-03
    3   1.068230e-04   2.513041e-06   3.542286e-09     8.148386e-06
    4   7.257177e-05   1.230928e-06   1.735068e-09     3.979480e-06
    5   3.610589e-05   2.634706e-07   3.713780e-10     1.175001e-06
```

11-39

```
 6    2.077811e-05   2.562892e-08   3.612554e-11    5.617206e-07
 7    1.611590e-05   4.711783e-10   6.641533e-13    5.652911e-08
 8    1.491953e-05   4.924200e-12   6.940795e-15    2.427880e-09
 9    1.477930e-05   1.316725e-13   1.850666e-16    2.454705e-10
10    1.476910e-05   2.498002e-16   8.316111e-19    2.786060e-11
```

Minimum found that satisfies the constraints.

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

Elapsed time is 4.014608 seconds.

Summary

This example illustrates how to use the interior-point algorithm in `quadprog` on a portfolio optimization problem, and shows the algorithm running times on quadratic problems of different sizes.

More elaborate analyses are possible by using features specifically designed for portfolio optimization in Financial Toolbox™.

See Also

"Mixed-Integer Quadratic Programming Portfolio Optimization: Solver-Based" on page 9-82

Quadratic Programming with Bound Constraints: Problem-Based

This example shows how to formulate and solve a scalable bound-constrained problem with a quadratic objective function. The example shows the solution behavior using several algorithms. The problem can have any number of variables; the number of variables is the scale. For the solver-based version of this example, see "Quadratic Minimization with Bound Constraints" on page 11-21.

The objective function, as a function of the number of problem variables n, is

$$2 \sum_{i=1}^{n} x_i^2 - 2 \sum_{i=1}^{n-1} x_i x_{i+1} - 2x_1 - 2x_n \, .$$

Create Problem

Create a problem variable named `x` that has 400 components. Also, create an expression named `objec` for the objective function. Bound each variable below by 0 and above by 0.9, except allow x_n to be unbounded.

```
n = 400;
x = optimvar('x',n,'LowerBound',0,'UpperBound',0.9);
x(n).LowerBound = -Inf;
x(n).UpperBound = Inf;
prevtime = 1:n-1;
nexttime = 2:n;
objec = 2*sum(x.^2) - 2*sum(x(nexttime).*x(prevtime)) - 2*x(1) - 2*x(end);
```

Create an optimization problem named `qprob`. Include the objective function in the problem.

```
qprob = optimproblem('Objective',objec);
```

Create options that specify the `quadprog` `'trust-region-reflective'` algorithm and no display. Create an initial point approximately centered between the bounds.

```
opts = optimoptions('quadprog','Algorithm','trust-region-reflective','Display','off');
x0 = 0.5*ones(n,1);
x00 = struct('x',x0);
```

Solve Problem and Examine Solution

Solve the problem.

```
[sol,qfval,qexitflag,qoutput] = solve(qprob,x00,'options',opts);
```

Plot the solution.

```
plot(sol.x,'b-')
xlabel('Index')
ylabel('x(index)')
```
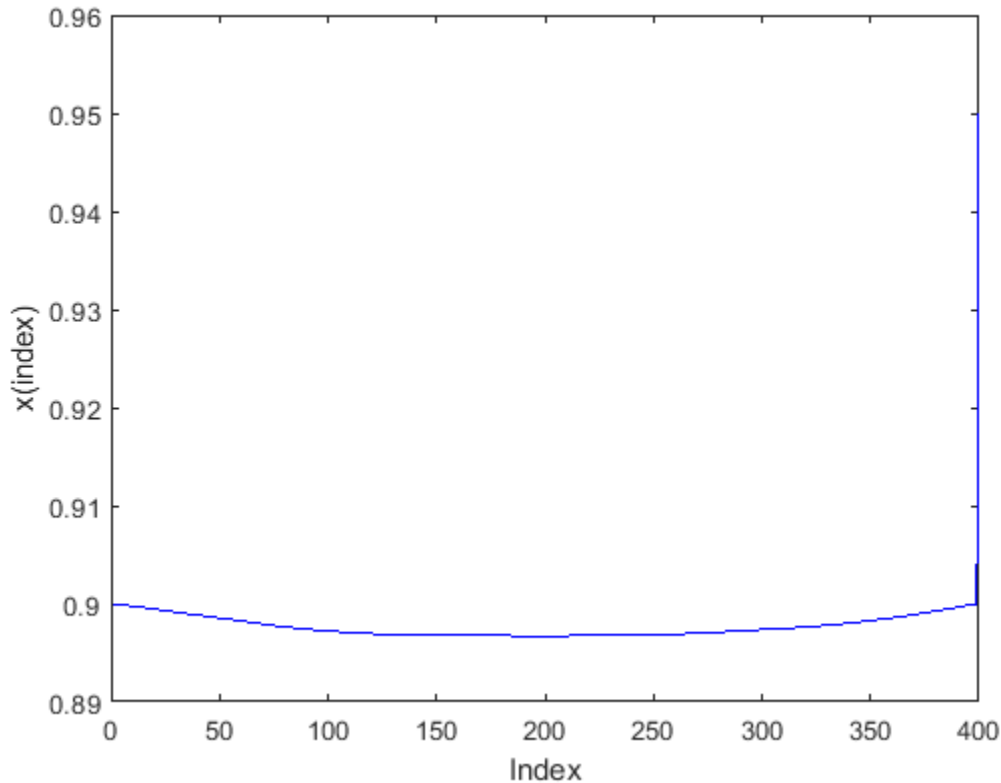
Report the exit flag, the number of iterations, and the number of conjugate gradient iterations.

```
fprintf('Exit flag = %d, iterations = %d, cg iterations = %d\n',...
    double(qexitflag),qoutput.iterations,qoutput.cgiterations)
```

```
Exit flag = 3, iterations = 19, cg iterations = 1637
```

There were a lot of conjugate gradient iterations.

Adjust Options for Increased Efficiency

Reduce the number of conjugate gradient iterations by setting the `SubproblemAlgorithm` option to `'factorization'`. This option causes the solver to use a more expensive internal solution technique that eliminates conjugate gradient steps, for a net overall savings of time in this case.

```
opts.SubproblemAlgorithm = 'factorization';
[sol2,qfval2,qexitflag2,qoutput2] = solve(qprob,x00,'options',opts);
fprintf('Exit flag = %d, iterations = %d, cg iterations = %d\n',...
    double(qexitflag2),qoutput2.iterations,qoutput2.cgiterations)
```

```
Exit flag = 3, iterations = 10, cg iterations = 0
```

The number of iterations and of conjugate gradient iterations decreased.

Compare Solutions With 'interior-point' Solution

Compare these solutions with that obtained using the default `'interior-point'` algorithm. The `'interior-point'` algorithm does not use an initial point, so do not pass `x00` to `solve`.

```
opts = optimoptions('quadprog','Algorithm','interior-point-convex','Display','off');
[sol3,qfval3,qexitflag3,qoutput3] = solve(qprob,'options',opts);
fprintf('Exit flag = %d, iterations = %d, cg iterations = %d\n',...
    double(qexitflag3),qoutput3.iterations,0)
```

```
Exit flag = 1, iterations = 8, cg iterations = 0
```

```
middle = floor(n/2);
fprintf('The three solutions are slightly different.\nThe middle component is %f, %f, or %f.\n',
    sol.x(middle),sol2.x(middle),sol3.x(middle))
```

```
The three solutions are slightly different.
The middle component is 0.896688, 0.898676, or 0.857389.
```

```
fprintf('The relative norm of sol - sol2 is %f.\n',norm(sol.x-sol2.x)/norm(sol.x))
```

```
The relative norm of sol - sol2 is 0.001629.
```

```
fprintf('The relative norm of sol2 - sol3 is %f.\n',norm(sol2.x-sol3.x)/norm(sol2.x))
```

```
The relative norm of sol2 - sol3 is 0.035894.
```

```
fprintf(['The three objective function values are %f, %f, and %f.\n' ...
    'The ''interior-point'' algorithm is slightly less accurate.'],qfval,qfval2,qfval3)
```

```
The three objective function values are -1.985000, -1.985000, and -1.984963.
The 'interior-point' algorithm is slightly less accurate.
```

See Also

More About

- "Quadratic Minimization with Bound Constraints" on page 11-21
- "Problem-Based Optimization Workflow" on page 10-2

Large Sparse Quadratic Program, Problem-Based

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, and a few nonzero diagonal bands. 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]. H is a sparse symmetric banded matrix.

Create Sparse Quadratic Matrix

Create a symmetric circulant matrix H based on shifts of the vector [3,6,2,14,2,6,3], with 14 being on the main diagonal. Have the matrix be n-by-n, where n = 30,000.

```
n = 3e4;
H2 = speye(n);
H = 3*circshift(H2,-3,2) + 6*circshift(H2,-2,2) + 2*circshift(H2,-1,2)...
    + 14*H2 + 2*circshift(H2,1,2) + 6*circshift(H2,2,2) + 3*circshift(H2,3,2);
```

View the sparse matrix structure.

```
spy(H)
```

Create Optimization Variables and Problem

Create an optimization variable x and problem qprob.

```
x = optimvar('x',n);
qprob = optimproblem;
```

Create the objective function and constraints. Place the objective and constraints into qprob.

```
f = 1:n;
obj = 1/2*x'*H*x - f*x;
qprob.Objective = obj;
cons = sum(x) <= 0;
qprob.Constraints = cons;
```

Solve Problem

Solve the quadratic programming problem using the default `'interior-point-convex'` algorithm and sparse linear algebra. To keep the solver from stopping prematurely, set the `StepTolerance` option to `0`.

```
options = optimoptions('quadprog','Algorithm','interior-point-convex',...
    'LinearSolver','sparse','StepTolerance',0);
[sol,fval,exitflag,output,lambda] = solve(qprob,'Options',options);
```
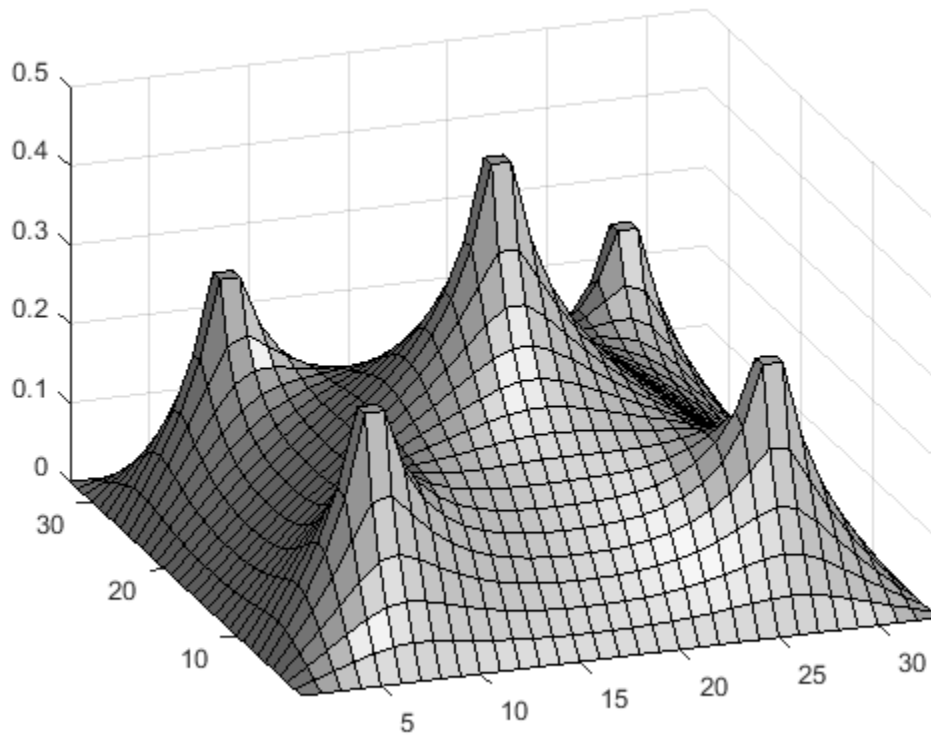
```
Minimum found that satisfies the constraints.

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

<stopping criteria details>
```

Examine Solution

View the objective function value, number of iterations, and Lagrange multiplier associated with the linear inequality constraint.

```
fprintf('The objective function value is %d.\nThe number of iterations is %d.\nThe Lagrange mult:
    fval,output.iterations,lambda.Constraints)
```

```
The objective function value is -3.133073e+10.
The number of iterations is 7.
The Lagrange multiplier is 1.500050e+04.
```

Evaluate the constraint to see that the solution is on the boundary.

```
fprintf('The linear inequality constraint sum(x) has value %d.\n',sum(sol.x))
```

```
The linear inequality constraint sum(x) has value 7.599738e-09.
```

The sum of the solution components is zero to within tolerances.

The solution x has three regions: an initial portion, a final portion, and an approximately linear portion over most of the solution. Plot the three regions.

```
subplot(3,1,1)
plot(sol.x(1:60))
title('x(1) Through x(60)')
```

```
subplot(3,1,2)
plot(sol.x(61:n-60))
title('x(61) Through x(n-60)')
subplot(3,1,3)
plot(sol.x(n-59:n))
title('x(n-59) Through x(n)')
```
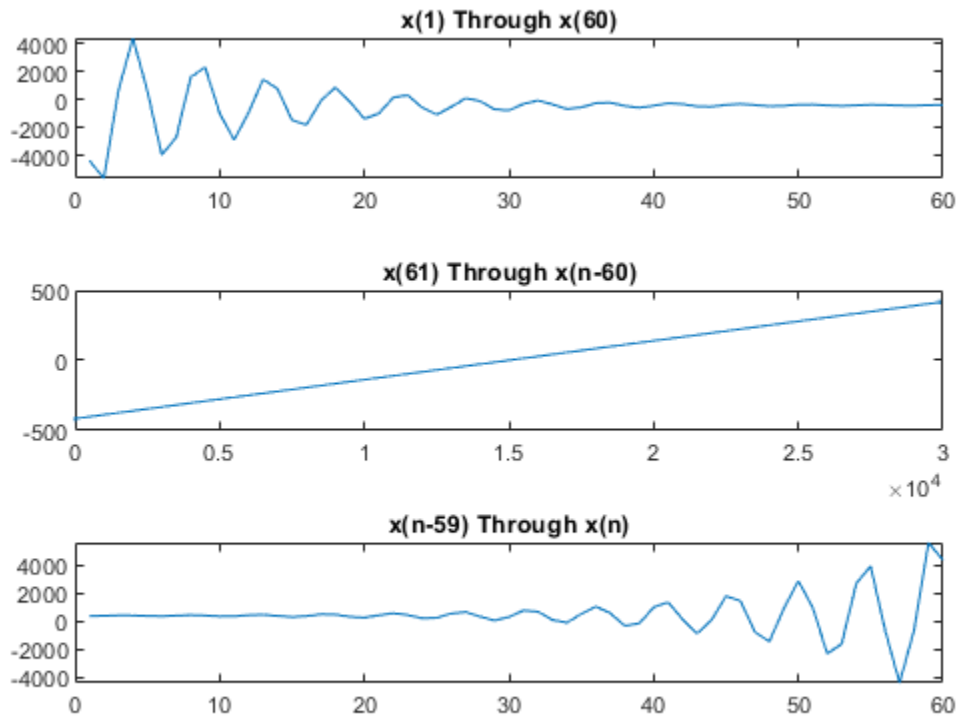

See Also

More About

- "Large Sparse Quadratic Program with Interior Point Algorithm" on page 11-28
- "Problem-Based Optimization Workflow" on page 10-2

Bound-Constrained Quadratic Programming, Problem-Based

This example shows how to determine the shape of a circus tent by solving a quadratic optimization problem. The tent is formed from heavy, elastic material, and settles into a shape that has minimum potential energy subject to constraints. A discretization of the problem leads to a bound-constrained quadratic programming problem.

For a solver-based version of this example, see "Bound-Constrained Quadratic Programming, Solver-Based" on page 11-31.

Problem Definition

Consider building a circus tent to cover a square lot. The tent has five poles covered with a heavy, elastic material. The problem is to find the natural shape of the tent. Model the shape as the height $x(p)$ of the tent at position p.

The potential energy of heavy material lifted to height x is cx, for a constant c that is proportional to the weight of the material. For this problem, choose $c = 1/3000$.

The elastic potential energy of a piece of the material E_{stretch} is approximately proportional to the second derivative of the material height, times the height. You can approximate the second derivative by the five-point finite difference approximation (assume that the finite difference steps are of size 1). Let Δx represent a shift of 1 in the first coordinate direction, and Δy represent a shift of 1 in the second coordinate direction.

$$E_{\text{stretch}}(p) = \left(-1 \Big(x(p + \Delta_x) + x(p - \Delta_x) + x(p + \Delta_y) + x(p - \Delta_y) \Big) + 4x(p) \right) x(p) .$$

The natural shape of the tent minimizes the total potential energy. By discretizing the problem, you find that the total potential energy to minimize is the sum over all positions p of $E_{\text{stretch}}(p) + cx(p)$.

This potential energy is a quadratic expression in the variable x.

Specify the boundary condition that the height of the tent at the edges is zero. The tent poles have a cross section of 1-by-1 unit, and the tent has a total size of 33-by-33 units. Specify the height and location of each pole. Plot the square lot region and tent poles.

```
height = zeros(33);
height(6:7,6:7) = 0.3;
height(26:27,26:27) = 0.3;
height(6:7,26:27) = 0.3;
height(26:27,6:7) = 0.3;
height(16:17,16:17) = 0.5;
colormap(gray);
surfl(height)
axis tight
view([-20,30]);
title('Tent Poles and Region to Cover')
```

Tent Poles and Region to Cover

Formulate Optimization Problem

Create an optimization variable x representing the height of the material.

```
x = optimvar('x',size(height));
```

Set x to zero on the boundaries of the square domain.

```
boundary = false(size(height));
boundary([1,33],:) = true;
boundary(:,[1,33]) = true;
x.LowerBound(boundary) = 0;
x.UpperBound(boundary) = 0;
```

Calculate the elastic potential energy at each point. First, calculate the potential energy in the interior of the region, where the finite differences do not overstep the region containing the solution.

```
L = size(height,1);
peStretch = optimexpr(L,L); % This initializes peStretch to zeros(L,L)
interior = 2:(L-1);
peStretch(interior,interior) = (-1*(x(interior - 1,interior) + x(interior + 1,interior) ...
    + x(interior,interior - 1) + x(interior,interior + 1)) + 4*x(interior,interior))...
    .*x(interior, interior);
```

Because the solution is constrained to be 0 at the edges of the region, you do not need to include the remainder of the terms. All terms have a multiple of x, and x at the edge is zero. For reference in case you want to use a different boundary condition, the following is a commented-out version of the potential energy .

```
% peStretch(1,interior) = (-1*(x(1,interior - 1) + x(1,interior + 1) + x(2,interior))...
%      + 4*x(1,interior)).*x(1,interior);
% peStretch(L,interior) = (-1*(x(L,interior - 1) + x(L,interior + 1) + x(L-1,interior))...
%      + 4*x(L,interior)).*x(L,interior);
% peStretch(interior,1) = (-1*(x(interior - 1,1) + x(interior + 1,1) + x(interior,2))...
%      + 4*x(interior,1)).*x(interior,1);
% peStretch(interior,L) = (-1*(x(interior - 1,L) + x(interior + 1,L) + x(interior,L-1))...
%      + 4*x(interior,L)).*x(interior,L);
% peStretch(1,1) = (-1*(x(2,1) + x(1,2)) + 4*x(1,1)).*x(1,1);
% peStretch(1,L) = (-1*(x(2,L) + x(1,L-1)) + 4*x(1,L)).*x(1,L);
% peStretch(L,1) = (-1*(x(L,2) + x(L-1,1)) + 4*x(L,1)).*x(L,1);
% peStretch(L,L) = (-1*(x(L-1,L) + x(L,L-1)) + 4*x(L,L)).*x(L,L);
```

Define the potential energy due to material height, which is `x/3000`.

```
peHeight = x/3000;
```

Create an optimization problem named `tentproblem`. Include the expression for the objective function, which is the sum of the two potential energies over all locations.

```
tentproblem = optimproblem('Objective',sum(sum(peStretch + peHeight)));
```

Set Constraint

Set the constraint that the solution must lie above the values of the `height` matrix. This matrix is zero at most locations, representing the ground, and includes the height of each tent pole at its location.

```
htcons = x >= height;
tentproblem.Constraints.htcons = htcons;
```

Run Optimization Solver

Solve the problem. Ignore the resulting statement "Your Hessian is not symmetric." `solve` issues this statement because the internal conversion from problem form to a quadratic matrix does not ensure that the matrix is symmetric.

```
sol = solve(tentproblem);
```

```
Solving problem using quadprog.
Your Hessian is not symmetric. Resetting H=(H+H')/2.

Minimum found that satisfies the constraints.

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

Plot Solution

Plot the solution found by the optimization solver.

```
surfl(sol.x);
axis tight;
view([-20,30]);
```

11-49

See Also

More About

- "Bound-Constrained Quadratic Programming, Solver-Based" on page 11-31
- "Problem-Based Optimization Workflow" on page 10-2

Quadratic Programming for Portfolio Optimization, Problem-Based

This example shows how to solve portfolio optimization problems using the problem-based approach. For the solver-based approach, see "Quadratic Programming for Portfolio Optimization Problems, Solver-Based" on page 11-35.

The Quadratic Model

Suppose that a portfolio contains n different assets. The rate of return of asset i is a random variable with expected value m_i. The problem is to find what fraction x_i to invest in each asset i in order to minimize risk, subject to a specified minimum expected rate of return.

Let C denote the covariance matrix of rates of asset returns.

The classical mean-variance model consists of minimizing portfolio risk, as measured by

$$\frac{1}{2}x^T C x$$

subject to a set of constraints.

The expected return should be no less than a minimal rate of portfolio return r that the investor desires,

$$\sum_{i=1}^{n} m_i \, x_i \geq r,$$

the sum of the investment fractions x_i's should add up to a total of one,

$$\sum_{i=1}^{n} x_i = 1,$$

and, being fractions (or percentages), they should be numbers between zero and one,

$$0 \leq x_i \leq 1, \quad i = 1 \ldots n.$$

Since the objective to minimize portfolio risk is quadratic, and the constraints are linear, the resulting optimization problem is a quadratic program, or QP.

225-Asset Problem

Let us now solve the QP with 225 assets. The dataset is from the OR-Library [Chang, T.-J., Meade, N., Beasley, J.E. and Sharaiha, Y.M., "Heuristics for cardinality constrained portfolio optimisation" Computers & Operations Research 27 (2000) 1271-1302].

We load the dataset and then set up the constraints for the problem-based approach. In this dataset the rates of return m_i range between -0.008489 and 0.003971; we pick a desired return r in between, e.g., 0.002 (0.2 percent).

Load dataset stored in a MAT-file.

```
load('port5.mat','Correlation','stdDev_return','mean_return')
```

Calculate the covariance matrix from correlation matrix.

```
Covariance = Correlation .* (stdDev_return * stdDev_return');
nAssets = numel(mean_return); r = 0.002;      % number of assets and desired return
```

Create Optimization Problem, Objective, and Constraints

Create an optimization problem for minimization.

```
portprob = optimproblem;
```

Create an optimization vector variable `'x'` with `nAssets` elements. This variable represents the fraction of wealth invested in each asset, so should lie between 0 and 1.

```
x = optimvar('x',nAssets,'LowerBound',0,'UpperBound',1);
```

The objective function is `1/2*x'*Covariance*x`. Include this objective into the problem.

```
objective = 1/2*x'*Covariance*x;
portprob.Objective = objective;
```

The sum of the variables is 1, meaning the entire portfolio is invested. Express this as a constraint and place it in the problem.

```
sumcons = sum(x) == 1;
portprob.Constraints.sumcons = sumcons;
```

The average return must be greater than `r`. Express this as a constraint and place it in the problem.

```
averagereturn = dot(mean_return,x) >= r;
portprob.Constraints.averagereturn = averagereturn;
```

Solve 225-Asset Problem

Set some options, and call the solver.

Set options to turn on iterative display, and set a tighter optimality termination tolerance.

```
options = optimoptions('quadprog','Display','iter','TolFun',1e-10);
```

Call solver and measure wall-clock time.

```
tic
[x1,fval1] = solve(portprob,'Options',options);
toc
```

```
 Iter          Fval   Primal Infeas    Dual Infeas  Complementarity
    0   7.212813e+00    1.227500e+02   1.195948e+00     2.217295e-03
    1   8.160874e-04    3.615084e-01   3.522160e-03     2.250524e-05
    2   7.220766e-04    3.592574e-01   3.500229e-03     3.378157e-05
    3   4.309434e-04    9.991108e-02   9.734292e-04     2.790551e-05
    4   4.734300e-04    5.551115e-16   7.771561e-16     4.242216e-06
    5   4.719034e-04    6.661338e-16   3.122502e-16     8.002618e-07
    6   3.587475e-04    4.440892e-16   3.035766e-18     3.677066e-07
    7   3.131814e-04    8.881784e-16   3.686287e-18     9.586695e-08
    8   2.760174e-04    7.771561e-16   1.463673e-18     1.521063e-08
    9   2.345751e-04    1.110223e-15   1.138412e-18     4.109608e-09
   10   2.042487e-04    1.221245e-15   1.084202e-18     6.423267e-09
```

```
11    1.961775e-04    1.110223e-16    9.757820e-19    6.068329e-10
12    1.949281e-04    4.440892e-16    9.215718e-19    4.279951e-12

Minimum found that satisfies the constraints.

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

Elapsed time is 0.292806 seconds.
```

Plot results.

```
plotPortfDemoStandardModel(x1.x)
```
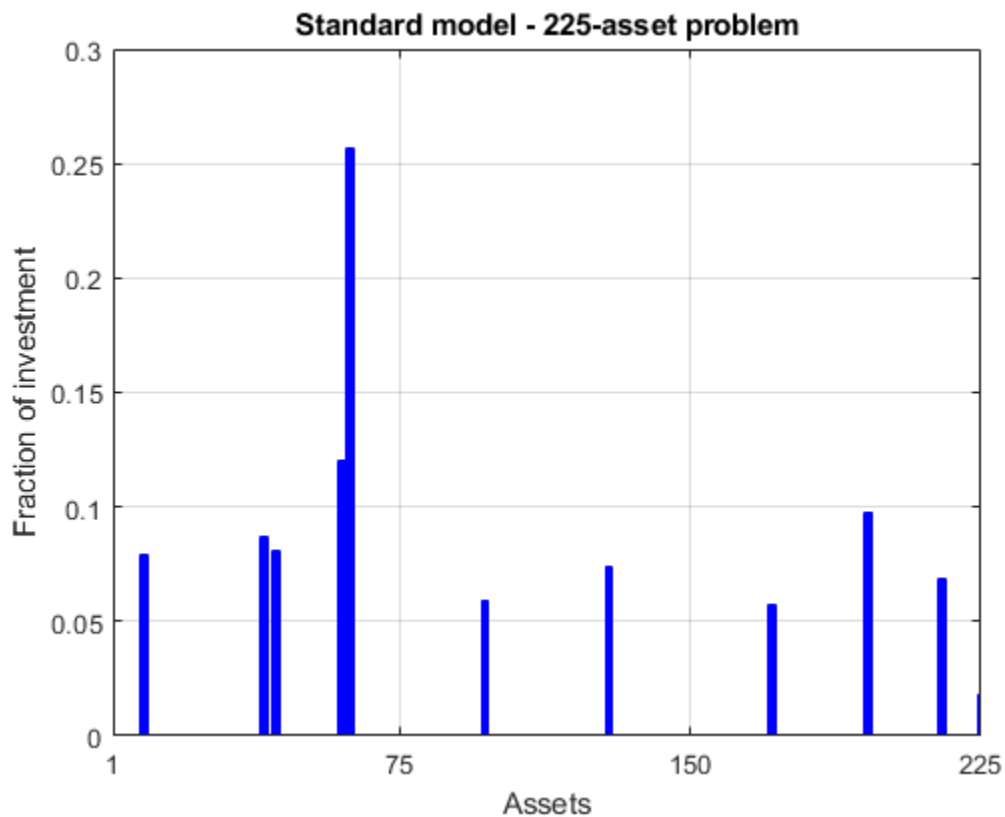

225-Asset Problem with Group Constraints

We now add to the model group constraints that require that 30% of the investor's money has to be invested in assets 1 to 75, 30% in assets 76 to 150, and 30% in assets 151 to 225. Each of these groups of assets could be, for instance, different industries such as technology, automotive, and pharmaceutical. The constraints that capture this new requirement are

$$\sum_{i=1}^{75} x_i \geq 0.3, \qquad \sum_{i=76}^{150} x_i \geq 0.3, \qquad \sum_{i=151}^{225} x_i \geq 0.3.$$

Add group constraints to existing equalities.

```
grp1 = sum(x(1:75)) >= 0.3;
grp2 = sum(x(76:150)) >= 0.3;
grp3 = sum(x(151:225)) >= 0.3;
portprob.Constraints.grp1 = grp1;
portprob.Constraints.grp2 = grp2;
portprob.Constraints.grp3 = grp3;
```

Call solver and measure wall-clock time.

```
tic
[x2,fval2] = solve(portprob,'Options',options);
toc
```

```
 Iter            Fval   Primal Infeas     Dual Infeas   Complementarity
    0    7.212813e+00    1.227500e+02    3.539920e-01      5.253824e-03
    1    7.004556e-03    2.901399e+00    8.367185e-03      2.207460e-03
    2    9.181962e-04    4.095630e-01    1.181116e-03      3.749424e-04
    3    7.515047e-04    3.567918e-01    1.028932e-03      3.486333e-04
    4    4.238346e-04    9.005778e-02    2.597127e-04      1.607718e-04
    5    3.695008e-04    1.909891e-04    5.507829e-07      1.341881e-05
    6    3.691407e-04    6.146337e-07    1.772508e-09      6.817457e-08
    7    3.010636e-04    7.691892e-08    2.218223e-10      1.837302e-08
    8    2.669065e-04    1.088252e-08    3.138350e-11      5.474712e-09
    9    2.195767e-04    8.122574e-10    2.342425e-12      2.814320e-08
   10    2.102910e-04    2.839773e-10    8.189470e-13      1.037476e-08
   11    2.060985e-04    6.713696e-11    1.936133e-13      2.876950e-09
   12    2.015107e-04    0.000000e+00    8.131516e-19      1.522226e-10
   13    2.009670e-04    4.440892e-16    8.673617e-19      5.264375e-13

Minimum found that satisfies the constraints.

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

Elapsed time is 0.162406 seconds.
```

Plot results, superimposed on results from previous problem.

```
plotPortfDemoGroupModel(x1.x,x2.x);
```
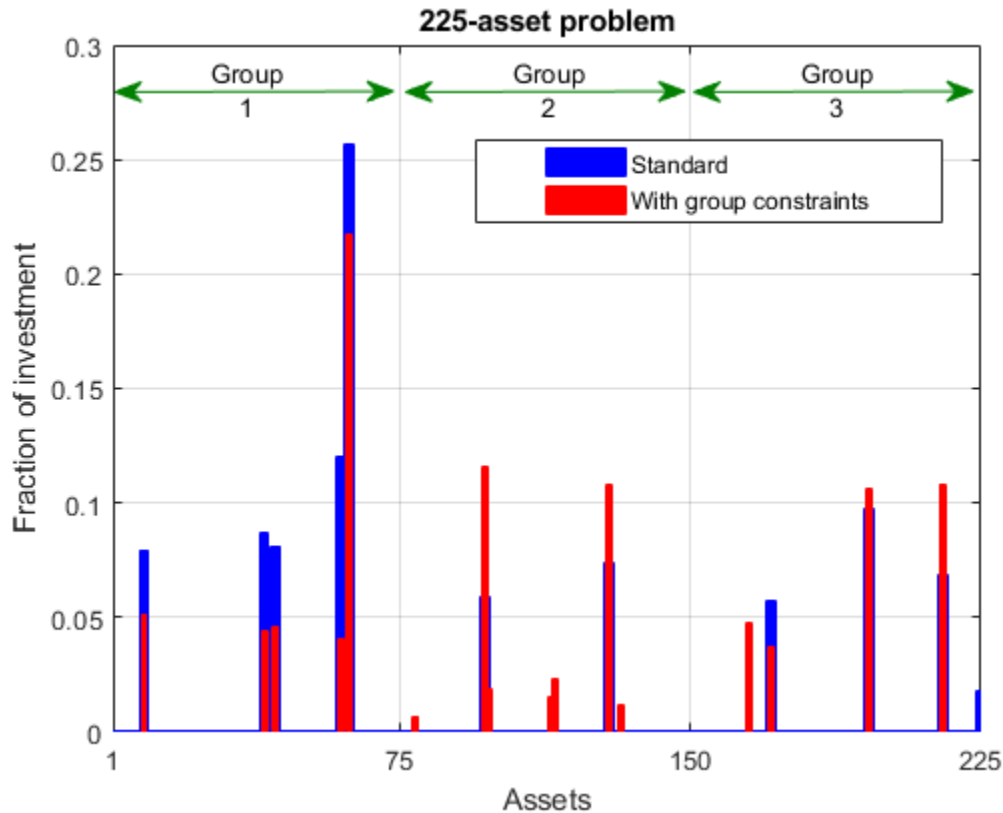
Summary of Results So Far

We see from the second bar plot that, as a result of the additional group constraints, the portfolio is now more evenly distributed across the three asset groups than the first portfolio. This imposed diversification also resulted in a slight increase in the risk, as measured by the objective function (see column labeled "f(x)" for the last iteration in the iterative display for both runs).

1000-Asset Problem Using Random Data

In order to show how the solver behaves on a larger problem, we'll use a 1000-asset randomly generated dataset. We generate a random correlation matrix (symmetric, positive-semidefinite, with ones on the diagonal) using the `gallery` function in MATLAB®.

Reset random stream for reproducibility.

```
rng(0,'twister');
nAssets = 1000; % desired number of assets
```

Create Random Data

Generate means of returns between -0.1 and 0.4.

```
a = -0.1; b = 0.4;
mean_return = a + (b-a).*rand(nAssets,1);
r = 0.15;                                   % desired return
```

Generate standard deviations of returns between 0.08 and 0.6.

```
a = 0.08; b = 0.6;
stdDev_return = a + (b-a).*rand(nAssets,1);
```

Load the correlation matrix, which was generated using `Correlation = gallery('randcorr',nAssets)`. (Generating a correlation matrix of this size takes a while, so load the pre-generated one instead.)

```
load('correlationMatrixDemo.mat','Correlation');
```

Calculate the covariance matrix from correlation matrix.

```
Covariance = Correlation .* (stdDev_return * stdDev_return');
```

Create Optimization Problem, Objective, and Constraints

Create an optimization problem for minimization.

```
portprob2 = optimproblem;
```

Create the optimization vector variable `'x'` with `nAssets` elements.

```
x = optimvar('x',nAssets,'LowerBound',0,'UpperBound',1);
```

Include the objective function into the problem.

```
objective = 1/2*x'*Covariance*x;
portprob2.Objective = objective;
```

Include the constraints that the sum of the variables is 1 and the average return is greater than r.

```
sumcons = sum(x) == 1;
portprob2.Constraints.sumcons = sumcons;
averagereturn = dot(mean_return,x) >= r;
portprob2.Constraints.averagereturn = averagereturn;
```

Solve 1000-Asset Problem

Call solver and measure wall-clock time.

```
tic
x3 = solve(portprob2,'Options',options);
toc
```

```
 Iter          Fval  Primal Infeas   Dual Infeas   Complementarity
    0   2.142849e+01   5.490000e+02   3.031839e+00      5.210929e-03
    1   9.378552e-03   6.439102e+00   3.555978e-02      6.331676e-04
    2   1.128129e-04   3.705915e-03   2.046582e-05      1.802721e-05
    3   1.118804e-04   1.852958e-06   1.023291e-08      1.170562e-07
    4   8.490176e-05   7.650016e-08   4.224702e-10      7.048637e-09
    5   3.364597e-05   4.440892e-16   3.062871e-18      1.037370e-09
    6   1.980189e-05   2.220446e-16   8.876905e-19      8.465558e-11

Minimum found that satisfies the constraints.

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

```
Elapsed time is 0.913357 seconds.
```

Summary

This example illustrates how to use problem-based approach on a portfolio optimization problem, and shows the algorithm running times on quadratic problems of different sizes.

More elaborate analyses are possible by using features specifically designed for portfolio optimization in Financial Toolbox™.

See Also

More About

- "Quadratic Programming for Portfolio Optimization Problems, Solver-Based" on page 11-35
- "Problem-Based Optimization Workflow" on page 10-2

Code Generation for quadprog Background

In this section...
"What Is Code Generation?" on page 11-58
"Code Generation Requirements" on page 11-58
"Generated Code Not Multithreaded" on page 11-59

What Is Code Generation?

Code generation is the conversion of MATLAB code to C/C++ code using MATLAB Coder. Code generation requires a MATLAB Coder license.

Typically, you use code generation to deploy code on hardware that is not running MATLAB.

For an example, see "Generate Code for quadprog" on page 11-60. For code generation in other optimization solvers, see "Generate Code for fmincon" on page 6-127, "Generate Code for fsolve" on page 13-38, or "Generate Code for lsqcurvefit or lsqnonlin" on page 12-89.

Code Generation Requirements

- `quadprog` supports code generation using either the `codegen` function or the MATLAB Coder app. You must have a MATLAB Coder license to generate code.
- The target hardware must support standard double-precision floating-point computations. You cannot generate code for single-precision or fixed-point computations.
- Code generation targets do not use the same math kernel libraries as MATLAB solvers. Therefore, code generation solutions can vary from solver solutions, especially for poorly conditioned problems.
- `quadprog` does not support the `problem` argument for code generation.

  ```
  [x,fval] = quadprog(problem) % Not supported
  ```
- All `quadprog` input matrices such as `A`, `Aeq`, `lb`, and `ub` must be full, not sparse. You can convert sparse matrices to full by using the `full` function.
- The `lb` and `ub` arguments must have the same number of entries as the number of columns in `H` or must be empty `[]`.
- For advanced code optimization involving embedded processors, you also need an Embedded Coder license.
- You must include options for `quadprog` and specify them using `optimoptions`. The options must include the `Algorithm` option, set to `'active-set'`.

  ```
  options = optimoptions('quadprog','Algorithm','active-set');
  [x,fval,exitflag] = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0,options);
  ```
- Code generation supports these options:

  - `Algorithm` — Must be `'active-set'`
  - `ConstraintTolerance`
  - `MaxIterations`
  - `ObjectiveLimit`

- • OptimalityTolerance
- • StepTolerance

- Generated code has limited error checking for options. The recommended way to update an option is to use `optimoptions`, not dot notation.

```
opts = optimoptions('quadprog','Algorithm','active-set');
opts = optimoptions(opts,'MaxIterations',1e4); % Recommended
opts.MaxIterations = 1e4; % Not recommended
```

- Do not load options from a file. Doing so can cause code generation to fail. Instead, create options in your code.

- If you specify an option that is not supported, the option is typically ignored during code generation. For reliable results, specify only supported options.

Generated Code Not Multithreaded

By default, generated code for use outside the MATLAB environment uses linear algebra libraries that are not multithreaded. Therefore, this code can run significantly slower than code in the MATLAB environment.

If your target hardware has multiple cores, you can achieve better performance by using custom multithreaded LAPACK and BLAS libraries. To incorporate these libraries in your generated code, see "Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls" (MATLAB Coder).

See Also
codegen | optimoptions | quadprog

More About
- • "Generate Code for quadprog" on page 11-60
- • "Static Memory Allocation for fmincon Code Generation" on page 6-131
- • "Optimization Code Generation for Real-Time Applications" on page 6-133

Generate Code for quadprog

First Steps in quadprog Code Generation

This example shows how to generate code for the quadprog optimization solver. Code generation requires a MATLAB Coder license. For details about code generation requirements, see "Code Generation for quadprog Background" on page 11-58.

The problem is to minimize the quadratic expression

$$\frac{1}{2}x^T H x + f^T x$$

where

$$H = \begin{bmatrix} 1 & -1 & 1 \\ -1 & 2 & -2 \\ 1 & -2 & 4 \end{bmatrix}$$

and

$$f = \begin{bmatrix} 2 \\ -3 \\ 1 \end{bmatrix}$$

subject to the constraints $0 \le x \le 1$, $\sum x = 1/2$.

Create a file named test_quadp.m containing the following code.

```
function [x,fval] = test_quadp
H = [1,-1,1
     -1,2,-2
     1,-2,4];
f = [2;-3;1];
lb = zeros(3,1);
ub = ones(size(lb));
Aeq = ones(1,3);
beq = 1/2;
x0 = zeros(3,1);
opts = optimoptions('quadprog','Algorithm','active-set');
[x,fval] = quadprog(H,f,[],[],Aeq,beq,lb,ub,x0,opts)
```

Generate code for the test_quadp file.

```
codegen -config:mex test_quadp
```

After some time, codegen creates a MEX file named test_quadp_mex.mexw64 (the file extension varies, depending on your system). Run the resulting C code.

```
[x,fval] = test_quadp_mex

x =

         0
    0.5000
```

```
        0
```

```
fval =

   -1.2500
```

Modify Example for Efficiency

Following some of the suggestions in the topic "Optimization Code Generation for Real-Time Applications" on page 6-133, configure the generated code to have fewer checks and to use static memory allocation.

```
cfg = coder.config('mex');
cfg.IntegrityChecks = false;
cfg.SaturateOnIntegerOverflow = false;
cfg.DynamicMemoryAllocation = 'Off';
```

Create a file named `test_quadp2.m` containing the following code. This code sets a looser optimality tolerance than the default `1e-8`.

```
function [x,fval,eflag,output] = test_quadp2
H = [1,-1,1
    -1,2,-2
    1,-2,4];
f = [2;-3;1];
lb = zeros(3,1);
ub = ones(size(lb));
Aeq = ones(1,3);
beq = 1/2;
x0 = zeros(3,1);
opts = optimoptions('quadprog','Algorithm','active-set',...
    'OptimalityTolerance',1e-5);
[x,fval,eflag,output] = quadprog(H,f,[],[],Aeq,beq,lb,ub,x0,opts)
```

Generate code for the `test_quadp2` file.

```
codegen -config cfg test_quadp2
```

Run the resulting code.

```
[x,fval,eflag,output] = test_quadp2_mex
```

```
x =

        0
   0.5000
        0
```

```
fval =

   -1.2500
```

```
eflag =
```

```
        1


output =

  struct with fields:

          algorithm: 'active-set'
      firstorderopt: 8.8818e-16
     constrviolation: 0
          iterations: 3
```

Changing the optimality tolerance does not affect the optimization process, because the `'active-set'` algorithm does not check this tolerance until it reaches a point where it stops.

Create a third file that limits the number of allowed iterations to 2 to see the effect on the optimization process.

```
function [x,fval,exitflag,output] = test_quadp3
H = [1,-1,1
    -1,2,-2
    1,-2,4];
f = [2;-3;1];
lb = zeros(3,1);
ub = ones(size(lb));
Aeq = ones(1,3);
beq = 1/2;
x0 = zeros(3,1);
opts = optimoptions('quadprog','Algorithm','active-set','MaxIterations',2);
[x,fval,exitflag,output] = quadprog(H,f,[],[],Aeq,beq,lb,ub,x0,opts)
```

To see the effect of these settings on the solver, run `test_quadp3` in MATLAB without generating code.

```
[x,fval,exitflag,output] = test_quadp3
```

```
Solver stopped prematurely.

quadprog stopped because it exceeded the iteration limit,
options.MaxIterations = 2.000000e+00.


x =

   -0.0000
    0.5000
         0


fval =

   -1.2500


exitflag =

     0
```

```
output =

  struct with fields:

        algorithm: 'active-set'
       iterations: 2
    constrviolation: 1.6441e-18
     firstorderopt: 2
          message: '↵Solver stopped prematurely.↵↵quadprog stopped because it exceeded the iter
      linearsolver: []
      cgiterations: []
```

In this case, the solver reached the solution in fewer steps than the default. Usually, though, limiting the number of iterations does not allow the solver to reach a correct solution.

See Also
codegen | optimoptions | quadprog

More About
- "Code Generation for quadprog Background" on page 11-58
- "Static Memory Allocation for fmincon Code Generation" on page 6-131
- "Optimization Code Generation for Real-Time Applications" on page 6-133

Quadratic Programming with Many Linear Constraints

This example shows how well the quadprog `'active-set'` algorithm performs in the presence of many linear constraints, as compared to the default `'interior-point-convex'` algorithm. Furthermore, the Lagrange multipliers from the `'active-set'` algorithm are exactly zero at inactive constraints, which can be helpful when you are looking for active constraints.

Problem Description

Create a pseudorandom quadratic problem with N variables and 10*N linear inequality constraints. Specify N = 150.

```
rng default % For reproducibility
N = 150;
rng default
A = randn([10*N,N]);
b = 10*ones(size(A,1),1);
f = sqrt(N)*rand(N,1);
H = 18*eye(N) + randn(N);
H = H + H';
```

Check that the resulting quadratic matrix is convex.

```
ee = min(eig(H))
```

```
ee = 3.6976
```

All of the eigenvalues are positive, so the quadratic form x'*H*x is convex.

Include no linear equality constraints or bounds.

```
Aeq = [];
beq = [];
lb = [];
ub = [];
```

Solve Problem Using Two Algorithms

Set options to use the quadprog `'active-set'` algorithm. This algorithm requires an initial point. Set the initial point x0 to be a zero vector of length N.

```
opts = optimoptions('quadprog','Algorithm','active-set');
x0 = zeros(N,1);
```

Time the solution.

```
tic
[xa,fvala,eflaga,outputa,lambdaa] = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0,opts);
```

```
Minimum found that satisfies the constraints.

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

<stopping criteria details>
```

```
toc
```

Elapsed time is 0.042058 seconds.

Compare the solution time to the time of the default `'interior-point-convex'` algorithm.

```
tic
[xi,fvali,eflagi,outputi,lambdai] = quadprog(H,f,A,b);
```

Minimum found that satisfies the constraints.

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

<stopping criteria details>

```
toc
```

Elapsed time is 2.305694 seconds.

The `'active-set'` algorithm is much faster on problems with many linear constraints.

Examine Lagrange Multipliers

The `'active-set'` algorithm reports only a few nonzero entries in the Lagrange multiplier structure associated with the linear constraint matrix.

```
nnz(lambdaa.ineqlin)
```

ans = 14

In contrast, the `'interior-point-convex'` algorithm returns a Lagrange multiplier structure with all nonzero elements.

```
nnz(lambdai.ineqlin)
```

ans = 1500

Nearly all of these Lagrange multipliers are smaller than `N*eps` in size.

```
nnz(abs(lambdai.ineqlin) > N*eps)
```

ans = 20

In other words, the `'active-set'` algorithm gives clear indications of active constraints in the Lagrange multiplier structure, whereas the `'interior-point-convex'` algorithm does not.

See Also
`lsqlin` | `quadprog`

More About

- "Potential Inaccuracy with Interior-Point Algorithms" on page 2-10

Convert Quadratic Constraints to Second-Order Cone Constraints

This example shows how to convert a quadratic constraint to the second-order cone constraint form. A quadratic constraint has the form

$$x^T Q x + 2q^T x + c \leq 0.$$

Second-order cone programming has constraints of the form

$$\|A_{sc}(i) \cdot x - b_{sc}(i)\| \leq d_{sc}(i) \cdot x - \gamma(i).$$

The matrix Q must be symmetric and positive semidefinite for you to convert quadratic constraints. Let S be the square root of Q, meaning $Q = S*S = S^T*S$. You can compute S using `sqrtm`. Suppose that there is a solution b to the equation $S^T b = -q$, which is always true when Q is positive definite. Compute b using `b = -S\q`.

$$
\begin{aligned}
x^T Q x + 2q^T x + c &= x^T S^T S x - 2\left(S^T b\right)^T x + c \\
&= (Sx - b)^T (Sx - b) - b^T b + c \\
&= \|Sx - b\|^2 + c - b^T b.
\end{aligned}
$$

Therefore, if $b^T b > c$, then the quadratic constraint is equivalent to the second-order cone constraint with

- $A_{sc} = S$
- $b_{sc} = b$
- $d_{sc} = 0$
- $\gamma = -\sqrt{b^T b - c}$

Numeric Example

Specify a five-element vector `f` representing the objective function $f^T x$.

```
f = [1;-2;3;-4;5];
```

Set the quadratic constraint matrix `Q` as a 5-by-5 random positive definite matrix. Set `q` as a random 5-element vector, and take the additive constant $c = -1$.

```
rng default % For reproducibility
Q = randn(5) + 3*eye(5);
Q = (Q + Q')/2; % Make Q symmetric
q = randn(5,1);
c = -1;
```

To create the inputs for `coneprog`, create the matrix `S` as the square root of `Q`.

```
S = sqrtm(Q);
```

Create the remaining inputs for the second-order cone constraint as specified in the first part of this example.

```
b = -S\q;
d = zeros(size(b));
gamma = -sqrt(b'*b-c);
sc = secondordercone(S,b,d,gamma);
```

Call `coneprog` to solve the problem.

```
[x,fval] = coneprog(f,sc)
```

```
Optimal solution found.
```

```
x = 5×1

   -0.7194
    0.2669
   -0.6309
    0.2543
   -0.0904
```

```
fval = -4.6148
```

Compare this result to the result returned by solving this same problem using `fmincon`. Write the quadratic constraint as described in "Anonymous Nonlinear Constraint Functions" on page 2-38.

```
x0 = randn(5,1); % Initial point for fmincon
nlc = @(x)x'*Q*x + 2*q'*x + c;
nlcon = @(x)deal(nlc(x),[]);
[xfmc,fvalfmc] = fmincon(@(x)f'*x,x0,[],[],[],[],[],[],nlcon)
```

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

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

```
xfmc = 5×1

   -0.7196
    0.2672
   -0.6312
    0.2541
   -0.0902
```

```
fvalfmc = -4.6148
```

The two solutions are nearly identical.

See Also

`coneprog` | `quadprog` | `secondordercone`

More About

- "Convert Quadratic Programming Problem to Second-Order Cone Program" on page 11-68
- "Minimize Energy of Piecewise Linear Mass-Spring System Using Cone Programming" on page 11-72

Convert Quadratic Programming Problem to Second-Order Cone Program

This example show how to convert a positive semidefinite quadratic programming problem to the second-order cone form used by the `coneprog` solver. A quadratic programming problem has the form

$$\min_{x} \frac{1}{2} x^T H x + f^T x,$$

possibly subject to bounds and linear constraints. `coneprog` solves problems in the form

$$\min_{x} f^T x$$

such that

$$\|A_{sc} x - b_{sc}\| \le d_{sc} x - \gamma,$$

possibly subject to bounds and linear constraints.

To convert a quadratic program to `coneprog` form, first calculate the matrix square root of the matrix H. Assuming that H is a symmetric positive semidefinite matrix, the command

`A = sqrtm(H);`

returns a positive semidefinite matrix A such that `A'*A = A*A = H`. Therefore,

$$x^T H x = x^T A^T A x = (Ax)^T A x = \|Ax\|^2.$$

Modify the form of the quadratic program as follows:

$$\min_{x} \frac{1}{2} x^T H x + f^T x = -\frac{1}{2} + \min_{x,t} \left[(t + 1/2) + f^T x \right]$$

where t satisfies the constraint

$$t + 1/2 \ge \frac{1}{2} x^T H x.$$

Extend the control variable x to u, which includes t as its last element:

$$u = \begin{bmatrix} x \\ t \end{bmatrix}.$$

Extend the second-order cone constraint matrices and vectors as follows:

$$A_{sc} = \begin{bmatrix} A & 0 \\ 0 & 1 \end{bmatrix}$$

$$b_{sc} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$d_{\mathrm{sc}} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

$\gamma = -1$.

Extend the coefficient vector f as well:

$$f_{\mathrm{sc}} = \begin{bmatrix} f \\ 1 \end{bmatrix}.$$

In terms of the new variables, the quadratic programming problem becomes

$$\min_{u} \frac{1}{2} u^T A_{\mathrm{sc}} u + f_{\mathrm{sc}}^T u = -1/2 + \min_{u} \left[1/2 + f_{\mathrm{sc}}^T u \right]$$

where

$$u(\text{end}) + 1/2 \geq \frac{1}{2} u^T A_{\mathrm{sc}} u.$$

This quadratic constraint becomes a cone constraint through the following calculation, which uses the earlier definitions of A_{sc}, d_{sc}, and γ:

$$\frac{1}{2} u^T A_{\mathrm{sc}} u = \frac{1}{2} \|Hx\|^2 \leq t + \frac{1}{2}$$

$$\|Hx\|^2 \leq 2t + 1$$

$$\|Hx\|^2 + t^2 \leq t^2 + 2t + 1 = (t+1)^2$$

$$\sqrt{\|Hx\|^2 + t^2} \leq \left| t + 1 \right| = \pm (t+1)$$

$$\|u\| \leq \pm (t - \gamma)$$

$$\|u\| \leq \pm (d_{\mathrm{sc}} - \gamma).$$

The quadratic program has the same solution as the corresponding cone program. The only difference is the added term $-1/2$ in the cone program.

Numerical Example

The quadprog documentation gives this example.

```
H = [1,-1,1
     -1,2,-2
     1,-2,4];
f = [-7;-12;-15];
lb = zeros(3,1);
ub = ones(size(lb));
Aineq = [1,1,1];
bineq = 3;
[xqp fqp] = quadprog(H,f,Aineq,bineq,[],[],lb,ub)
```

```
Minimum found that satisfies the constraints.

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

xqp = 3×1

    1
    1
    1


fqp = -32.5000
```

Referring to the description at the beginning of this example, specify the second-order cone constraint variables, and then call the `coneprog` function.

```
Asc = sqrtm(H);
Asc((end+1),(end+1)) = 1;
d = [zeros(size(f(:)));1];
gamma = -1;
b = zeros(size(d));
qp = secondordercone(Asc,b,d,gamma);
Aq = Aineq;
Aq(:,(end+1)) = 0;
lb(end+1) = -Inf;
ub(end+1) = Inf;
[u,fval,eflag] = coneprog([f(:);1],qp,Aq,bineq,[],[],lb,ub)

Optimal solution found.

u = 4×1

    1.0000
    1.0000
    1.0000
    1.0000


fval = -33.0000

eflag = 1
```

The first three elements of the cone solution `u` are equal to the elements of the quadratic programming solution `xqp`, to display precision:

```
disp([xqp,u(1:(end-1))])

    1.0000    1.0000
    1.0000    1.0000
    1.0000    1.0000
```

The returned quadratic function value `fqp` is the returned cone value minus 1/2 when $2t + 1$ is positive, or plus 1/2 when $2t + 1$ is negative.

```
disp([fqp-sign(2*u(end)+1)*1/2 fval])
```

```
-33.0000   -33.0000
```

See Also
`coneprog` | `quadprog` | `secondordercone`

More About

Minimize Energy of Piecewise Linear Mass-Spring System Using Cone Programming

This example shows how to find the equilibrium position of a mass-spring system hanging from two anchor points. The springs have piecewise linear tensile forces. The system consists of n masses in two dimensions. Mass i is connected to springs i and $i + 1$. Springs 1 and $n + 1$ are also connected to separate anchor points. In this case, the zero-force length of spring i is a positive length $l(i)$, and the spring generates force $k(i)q$ when stretched to length $q + l(i)$. The problem is to find the minimum potential energy configuration of the masses, where potential energy comes from the force of gravity and from the stretching of the nonlinear springs. The equilibrium occurs at the minimum energy configuration.

This illustration shows five springs and four masses suspended from two anchor points.
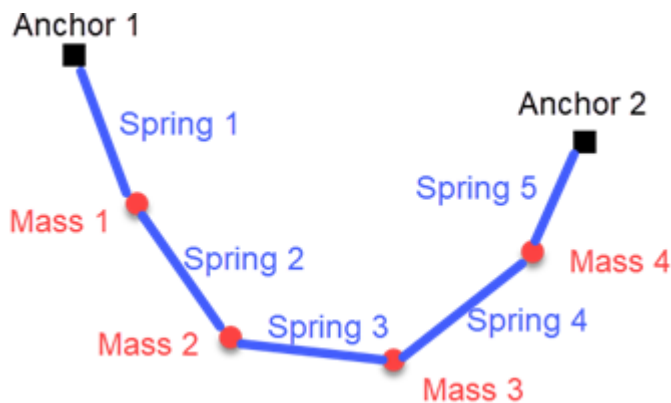
The potential energy of a mass m at height h is mgh, where g is the gravitational constant on Earth. Also, the potential energy of an ideal linear spring with spring constant k stretched to length q is $kq^2/2$. The current model is that the spring is not ideal, but has a nonzero resting length l.

The mathematical basis of this example comes from Lobo, Vandenberg, Boyd, and Lebret [1] on page 11-0 .

Mathematical Formulation

The location of mass i is $x(i)$, with horizontal coordinate $x_1(i)$ and vertical coordinate $x_2(i)$. Mass i has potential energy due to gravity of $gm(i)x_2(i)$. The potential energy in spring i is $k(i)(d(i) - l(i))^2/2$, where $d(i)$ is the length of the spring between mass i and mass $i - 1$. Take anchor point 1 as the position of mass 0, and anchor point 2 as the position of mass $n + 1$. The preceding energy calculation shows that the potential energy of spring i is

$$\text{Energy}(i) = \frac{k(i)\left(\|x(i) - x(i-1)\| - l(i)\right)^2}{2}.$$

Reformulating this potential energy problem as a second-order cone problem requires the introduction of some new variables, as described in Lobo [1] on page 11-0 . Create variables $t(i)$ equal to the square root of the term Energy(i).

$$t(i) = \sqrt{\frac{k(i)\left(\|x(i) - x(i-1)\| - l(i)\right)^2}{2}}.$$

Let e be the unit column vector $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Then $x_2(i) = e^T x(i)$. The problem becomes

$$\min_{x,t}\left(\sum_i gm(i)e^T x(i) + \|t\|^2\right). \qquad (1)$$

Now consider t as a free vector variable, not given by the previous equation for $t(i)$. Incorporate the relationship between $x(i)$ and $t(i)$ in the new set of cone constraints

$$\|x(i) - x(i-1)\| - l(i) \leq \sqrt{\frac{2}{k(i)}}t(i). \quad (2)$$

The objective function is not yet linear in its variables, as required for `coneprog`. Introduce a new scalar variable y. Notice that the inequality $\|t\|^2 \leq y$ is equivalent to the inequality

$$\left\|\begin{bmatrix} 2t \\ 1-y \end{bmatrix}\right\| \leq 1 + y. \qquad (3)$$

Now the problem is to minimize

$$\min_{x,t,y}\left(\sum_i gm(i)e^T x(i) + y\right) \qquad (4)$$

subject to the cone constraints on $x(i)$ and $t(i)$ listed in (2) and the additional cone constraint (3). Cone constraint (3) ensures that $\|t\|^2 \leq y$. Therefore, problem (4) is equivalent to problem (1).

The objective function and cone constraints in problem (4) are suitable for solution with `coneprog`.

MATLAB® Formulation

Define six spring constants k, six length constants l, and five masses m.

```
k = 40*(1:6);
l = [1 1/2 1 2 1 1/2];
m = [2 1 3 2 1];
```

Define the approximate gravitational constant on Earth g.

```
g = 9.807;
```

The variables for optimization are the ten components of the x vectors, the six components of the t vector, and the y variable. Let v be the vector containing all these variables.

- `[v(1),v(2)]` corresponds to the 2-D variable $x(1)$.
- `[v(3),v(4)]` corresponds to the 2-D variable $x(2)$.
- `[v(5),v(6)]` corresponds to the 2-D variable $x(3)$.
- `[v(7),v(8)]` corresponds to the 2-D variable $x(4)$.
- `[v(9),v(10)]` corresponds to the 2-D variable $x(5)$.
- `[v(11):v(16)]` corresponds to the 6-D vector t.
- `v(17)` corresponds to the scalar variable y.

Using these variables, create the corresponding objective function vector f.

```
f = zeros(size(m));
f = [f;g*m];
f = f(:);
f = [f;zeros(length(k)+1,1)];
f(end) = 1;
```

Create the cone constraints corresponding to the springs between the masses (2)

$$\|x(i) - x(i-1)\| - l(i) \le \sqrt{\frac{2}{k(i)}} t(i).$$

The `coneprog` solver uses cone constraints for a variable vector v in the form

$$\|A_{sc} \cdot v - b_{sc}\| \le d_{sc}^T v - \gamma.$$

In the following code, the `Asc` matrix represents the term $\|x(i) - x(i-1)\|$, with `bsc = [0;0]`. The cone variable `dsc` $= \sqrt{2/k(i)}$ and the corresponding `gamma` $= -l(i)$.

```
d = zeros(1,length(f));
Asc = d;
Asc([1 3]) = [1 -1];
A2 = circshift(Asc,1);
Asc = [Asc;A2];
ml = length(m);
dbase = 2*ml;
bsc = [0;0];
for i = 2:ml
    gamma = -l(i);
    dsc = d;
    dsc(dbase + i) = sqrt(2/k(i));
    conecons(i) = secondordercone(Asc,bsc,dsc,gamma);
    Asc = circshift(Asc,2,2);
end
```

Create the cone constraints corresponding to the springs between the end masses and the anchor points by using the anchor points for the locations of the end masses, as in the preceding code.

```
x0 = [0;5];
xn = [5;4];

Asc = zeros(size(Asc));
Asc(1,(dbase-1)) = 1;
Asc(2,dbase) = 1;
bsc = xn;
gamma = -l(ml);
dsc = d;
dsc(dbase + ml) = sqrt(2/k(ml));
conecons(ml + 1) = secondordercone(Asc,bsc,dsc,gamma);

Asc = zeros(size(Asc));
Asc(1,1) = 1;
Asc(2,2) = 1;
bsc = x0;
gamma = -l(1);
dsc = d;
dsc(dbase + 1) = sqrt(2/k(1));
conecons(1) = secondordercone(Asc,bsc,dsc,gamma);
```

Create the cone constraint (3) corresponding to the y variable

$$\left\| \begin{bmatrix} 2t \\ 1 - y \end{bmatrix} \right\| \leq 1 + y$$

by creating the matrix `Asc` which, when multiplied by the `v` vector, gives the vector $\begin{bmatrix} 2t \\ -y \end{bmatrix}$. The `bsc` vector corresponds to the constant 1 in the term $1 - y$. The `dsc` vector, when multiplied by `v`, returns y. And `gamma` $= -1$.

```
Asc = 2*eye(length(f));
Asc(1:dbase,:) = [];
Asc(end,end) = -1;
bsc = zeros(size(Asc,1),1);
bsc(end) = -1;
dsc = d;
dsc(end) = 1;
gamma = -1;
conecons(ml+2) = secondordercone(Asc,bsc,dsc,gamma);
```

Finally, create lower bounds corresponding to the t and y variables.

```
lb = -inf(size(f));
lb(dbase+1:end) = 0;
```

Solve Problem and Plot Solution

The problem formulation is complete. Solve the problem by calling `coneprog`.

```
[v,fval,exitflag,output] = coneprog(f,conecons,[],[],[],[],lb);
```

Optimal solution found.

Plot the solution points and the anchors.

```
pp = v(1:2*ml);
pp = reshape(pp,2,[]);
pp = pp';
plot(pp(:,1),pp(:,2),'ro')
hold on
xx = [x0,xn]';
plot(xx(:,1),xx(:,2),'ks')
xlim([x0(1)-0.5,xn(1)+0.5])
ylim([min(pp(:,2))-0.5,max(x0(2),xn(2))+0.5])
xxx = [x0';pp;xn'];
plot(xxx(:,1),xxx(:,2),'b--')
legend('Calculated points','Anchor points','Springs','Location',"best")
hold off
```
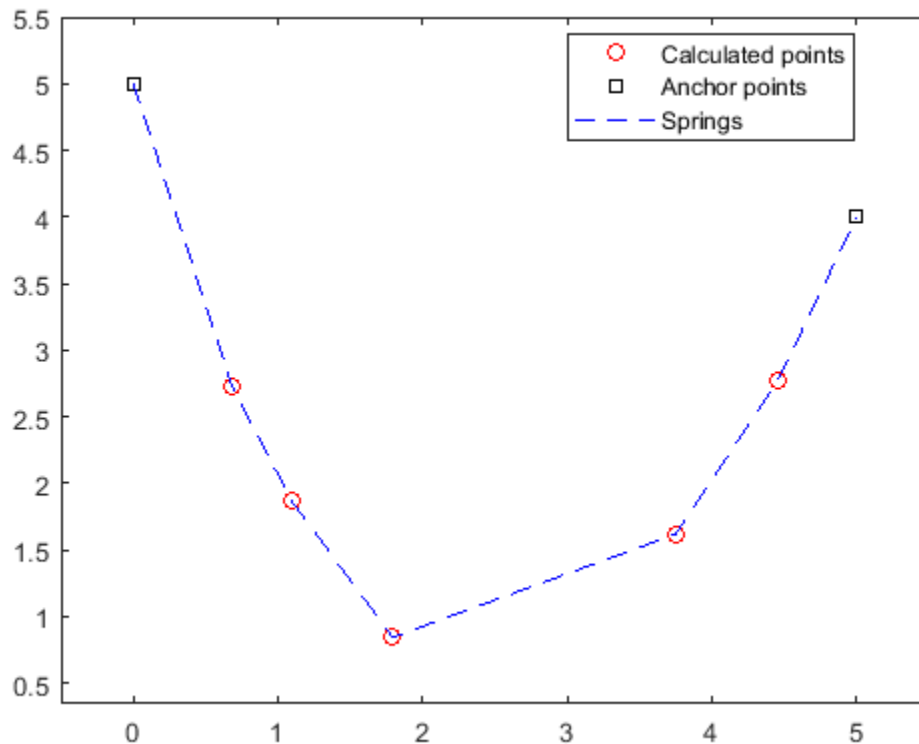
You can change the values of the parameters m, l, and k to see how they affect the solution. You can also change the number of masses; the code takes the number of masses from the data you supply.

References

[1] Lobo, Miguel Sousa, Lieven Vandenberghe, Stephen Boyd, and Hervé Lebret. "Applications of Second-Order Cone Programming." *Linear Algebra and Its Applications* 284, no. 1–3 (November 1998): 193–228. `https://doi.org/10.1016/S0024-3795(98)10032-0`.

See Also

`coneprog` | `secondordercone`

More About

- "Convert Quadratic Constraints to Second-Order Cone Constraints" on page 11-66
- "Convert Quadratic Programming Problem to Second-Order Cone Program" on page 11-68

Least Squares

Least-Squares (Model Fitting) Algorithms

In this section...
"Least Squares Definition" on page 12-2
"Linear Least Squares: Interior-Point or Active-Set" on page 12-2
"Trust-Region-Reflective Least Squares" on page 12-3
"Levenberg-Marquardt Method" on page 12-6

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 $A \cdot x \le b$, $Aeq \cdot x = beq$, $lb \le x \le ub$.

There are several Optimization Toolbox solvers available for various types of $F(x)$ and various types of constraints:

Solver	F(x)	Constraints
mldivide	$C \cdot x - d$	None
lsqnonneg	$C \cdot x - d$	$x \ge 0$
lsqlin	$C \cdot x - d$	Bound, linear
lsqnonlin	General $F(x)$	Bound
lsqcurvefit	$F(x, xdata) - ydata$	Bound

There are five least-squares algorithms in Optimization Toolbox solvers, in addition to the algorithms used in `mldivide`:

- `lsqlin` interior-point
- `lsqlin` active-set
- Trust-region-reflective (nonlinear or linear least-squares)
- Levenberg-Marquardt (nonlinear least-squares)
- The algorithm used by `lsqnonneg`

All the algorithms except `lsqlin` active-set are large-scale; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-10. 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].

Linear Least Squares: Interior-Point or Active-Set

The `lsqlin` `'interior-point'` algorithm uses the "interior-point-convex quadprog Algorithm" on page 11-2, and the `lsqlin` `'active-set'` algorithm uses the active-set `quadprog` algorithm. 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

$$\|Cx - d\|_2^2 = (Cx - d)^T(Cx - d)$$

$$= \left(x^T C^T - d^T\right)(Cx - d)$$

$$= \left(x^T C^T Cx\right) - \left(x^T C^T d - d^T Cx\right) + d^T d$$

$$= \frac{1}{2} x^T \left(2 C^T C\right) x + \left(-2 C^T d\right)^T x + d^T d .$$

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, `quadprog` calculates the solution.

Note The `quadprog` `'interior-point-convex'` algorithm has two code paths. It takes one when the Hessian matrix H is an ordinary (full) matrix of doubles, and it takes the other when H is a sparse matrix. For details of the sparse data type, see "Sparse Matrices". Generally, the algorithm is faster for large problems that have relatively few nonzero terms when you specify H as `sparse`. Similarly, the algorithm is faster for small or relatively dense problems when you specify H as `full`.

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,

$$\min_s \{q(s), \ s \in N\} .$$ (12-1)

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

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

In the standard trust-region method ([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

$$\min\left\{\frac{1}{2}s^THs + s^Tg \text{ such that } \|Ds\| \le \Delta\right\},$$ (12-2)

where g is the gradient of f at the current point x, H is the Hessian matrix (the symmetric matrix of second derivatives), D is a diagonal scaling matrix, Δ is a positive scalar, and $\| . \|$ is the 2-norm. Good algorithms exist for solving "Equation 12-2" (see [48]); such algorithms typically involve the computation of all eigenvalues of H and a Newton process applied to the secular equation

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

Such algorithms provide an accurate solution to "Equation 12-2". 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 12-2", 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 12-2" 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

$$H \cdot s_2 = -g,$$ (12-3)

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0.$$ (12-4)

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

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

1   Formulate the two-dimensional trust-region subproblem.

2   Solve "Equation 12-2" 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., $f(x + s) \ge f(x)$. See [46] and [49] for a discussion of this aspect.

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

Large Scale Nonlinear Least Squares

An important special case for $f(x)$ is the nonlinear least-squares problem

$$\min_x \sum_i f_i^2(x) = \min_x \|F(x)\|_2^2, \tag{12-5}$$

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

$$\min \|Js + F\|_2^2, \tag{12-6}$$

(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.,

$$J^T J s = -J^T F,$$

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.,

$$C^T C x = -C^T d,$$

although the normal equations are not explicitly formed.

The subspace trust-region method is used to determine a search direction. However, instead of restricting the step to (possibly) one reflection step, as in the nonlinear minimization case, a piecewise reflective line search is conducted at each iteration, as in the quadratic case. See [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 12-30.

Levenberg-Marquardt Method

The least-squares problem minimizes a function $f(x)$ that is a sum of squares.

$$\min_{x} f(x) = \|F(x)\|_2^2 = \sum_i F_i^2(x). \tag{12-7}$$

Problems of this type occur in a large number of practical applications, especially those that involve fitting model functions to data, such as nonlinear parameter estimation. This problem type also appears in control systems, where the objective is for the output $y(x,t)$ to follow a continuous model trajectory $\varphi(t)$ for vector x and scalar t. This problem can be expressed as

$$\min_{x \in \mathfrak{R}^n} \int_{t_1}^{t_2} (y(x, t) - \varphi(t))^2 dt, \tag{12-8}$$

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

Discretize the integral to obtain an approximation

$$\min_{x \in \mathfrak{R}^n} f(x) = \sum_{i=1}^{m} (y(x, t_i) - \varphi(t_i))^2, \tag{12-9}$$

where the t_i are evenly spaced. In this problem, the vector $F(x)$ is

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

In this type of problem, the residual $\|F(x)\|$ is likely to be small at the optimum, because the general practice is to set realistically achievable target trajectories. Although you can minimize the function in "Equation 12-7" using a general, unconstrained minimization technique, as described in "Basics of Unconstrained Optimization" on page 6-4, certain characteristics of the problem can often be exploited to improve the iterative efficiency of the solution procedure. The gradient and Hessian matrix of "Equation 12-7" have a special structure.

Denoting the m-by-n Jacobian matrix of $F(x)$ as $J(x)$, gradient vector of $f(x)$ as $G(x)$, Hessian matrix of $f(x)$ as $H(x)$, and Hessian matrix of each $F_i(x)$ as $D_i(x)$,

$$G(x) = 2J(^XF(x)$$

$$H(x) = 2J(^XJ(x) + 2Q(x), \tag{12-10}$$

where

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

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

At each major iteration k, the Gauss-Newton method obtains a search direction d_k that is a solution of the linear least-squares problem

$$\min_{d_k \in \mathfrak{R}^n} \|J(x_k)d_k + F(x_k)\|_2^2. \tag{12-11}$$

The direction derived from this method is equivalent to the Newton direction when the terms of $Q(x)$ = 0. The algorithm can use the search direction d_k as part of a line search strategy to ensure that the function $f(x)$ decreases at each iteration.

The Gauss-Newton method often encounters problems when the second-order term $Q(x)$ is nonnegligible. The Levenberg-Marquardt method overcomes this problem.

The Levenberg-Marquardt method (see [25] and [27]) uses a search direction that is a solution of the linear set of equations

$$\left(J(x_k)^T J(x_k) + \lambda_k I\right)d_k = -J(x_k)^T F(x_k), \tag{12-12}$$

or, optionally, of the equations

$$\left(J(x_k)^T J(x_k) + \lambda_k \text{diag}\left(J(x_k)^T J(x_k)\right)\right)d_k = -J(x_k)^T F(x_k), \tag{12-13}$$

where the scalar λ_k controls both the magnitude and direction of d_k, and `diag(A)` means the matrix of diagonal terms in `A`. Set the option `ScaleProblem` to `'none'` to choose "Equation 12-12", or set `ScaleProblem` to `'Jacobian'` to choose "Equation 12-13".

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 from the default, such as `1e2`.

When λ_k is zero, the direction d_k is identical to that of the Gauss-Newton method. As λ_k tends towards infinity, d_k tends towards the steepest descent direction, with magnitude tending towards zero. Consequently, for some sufficiently large λ_k, the term $F(x_k + d_k) < F(x_k)$ holds true. Therefore, you can control the term λ_k to ensure descent even when the algorithm encounters second-order terms, which restrict the efficiency of the Gauss-Newton method. 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 $\lambda_{k+1} = \lambda_k*10$.

Internally, the Levenberg-Marquardt algorithm uses an optimality tolerance (stopping criterion) of `1e-4` times the function tolerance.

The Levenberg-Marquardt method, therefore, uses a search direction that is a cross between the Gauss-Newton direction and the steepest descent direction.

Another advantage to the Levenberg-Marquardt method is when the Jacobian J is rank-deficient. In this case, the Gauss-Newton method can have numerical issues because the minimization problem in "Equation 12-11" is ill-posed. In contrast, the Levenberg-Marquardt method has full rank at each iteration, and, therefore, avoids these issues.

The following figure shows the iterations of the Levenberg-Marquardt method when minimizing Rosenbrock's function, a notoriously difficult minimization problem that is in least-squares form.
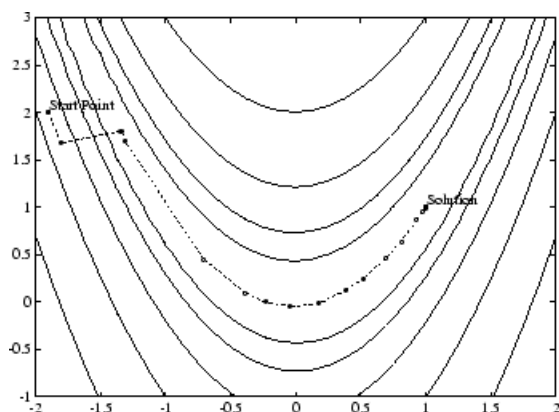
Levenberg-Marquardt Method on Rosenbrock's Function

For a more complete description of this figure, including scripts that generate the iterative points, see "Banana Function Minimization" on page 6-49.

Bound Constraints in Levenberg-Marquardt Method

When the problem contains bound constraints, `lsqcurvefit` and `lsqnonlin` modify the Levenberg-Marquardt iterations. If a proposed iterative point x lies outside of the bounds, the algorithm projects the step onto the nearest feasible point. In other words, with P defined as the projection operator that projects infeasible points onto the feasible region, the algorithm modifies the proposed point x to $P(x)$. By definition, the projection operator P operates on each component x_i independently according to

$$P(x_i) = \begin{cases} \text{lb}_i & \text{if } x_i < \text{lb}_i \\ \text{ub}_i & \text{if } x_i > \text{ub}_i \\ x_i & \text{otherwise} \end{cases}$$

or, equivalently,

$$P(x_i) = \min(\max(x_i, \text{lb}_i), \text{ub}_i).$$

The algorithm modifies the stopping criterion for the first-order optimality measure. Let x be a proposed iterative point. In the unconstrained case, the stopping criterion is

$$\|\nabla f(x)\|_\infty \le \text{tol}, \tag{12-14}$$

where tol is the optimality tolerance value, which is `1e-4*FunctionTolerance`. In the bounded case, the stopping criterion is

$$\|x - P(x - \nabla f(x))\|_\infty^2 \le \text{tol}\|\nabla f(x)\|_\infty. \tag{12-15}$$

To understand this criterion, first note that if x is in the interior of the feasible region, then the operator P has no effect. So, the stopping criterion becomes

$$\|x - P(x - \nabla f(x))\|_\infty^2 = \|\nabla f(x)\|_\infty^2 \leq \text{tol} \|\nabla f(x)\|_\infty,$$

which is the same as the original unconstrained stopping criterion, $\|\nabla f(x)\|_\infty \leq \text{tol}$. If the boundary constraint is active, meaning $x - \nabla f(x)$ is not feasible, then at a point where the algorithm should stop, the gradient at a point on the boundary is perpendicular to the boundary. Therefore, the point x is equal to $P(x - \nabla f(x))$, the projection of the steepest descent step, as shown in the following figure.
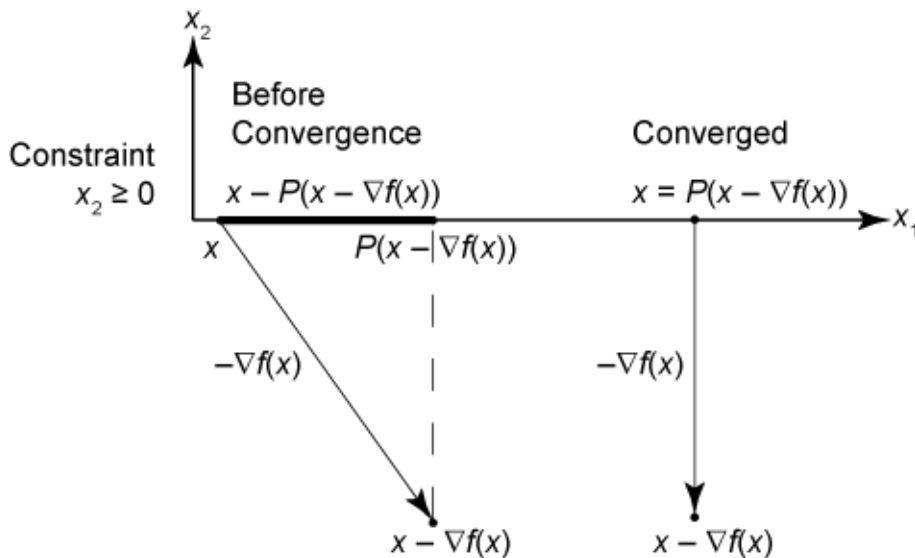
Levenberg-Marquardt Stopping Condition

See Also
lsqcurvefit | lsqlin | lsqnonlin | lsqnonneg | quadprog

More About
*   "Least Squares"

Nonlinear Data-Fitting

This example shows how to fit a nonlinear function to data using several Optimization Toolbox™ algorithms.

Problem Setup

Consider the following data:

```
Data = ...
  [0.0000    5.8955
   0.1000    3.5639
   0.2000    2.5173
   0.3000    1.9790
   0.4000    1.8990
   0.5000    1.3938
   0.6000    1.1359
   0.7000    1.0096
   0.8000    1.0343
   0.9000    0.8435
   1.0000    0.6856
   1.1000    0.6100
   1.2000    0.5392
   1.3000    0.3946
   1.4000    0.3903
   1.5000    0.5474
   1.6000    0.3459
   1.7000    0.1370
   1.8000    0.2211
   1.9000    0.1704
   2.0000    0.2636];
```

Let's plot these data points.

```
t = Data(:,1);
y = Data(:,2);
% axis([0 2 -0.5 6])
% hold on
plot(t,y,'ro')
title('Data points')
```

Data points
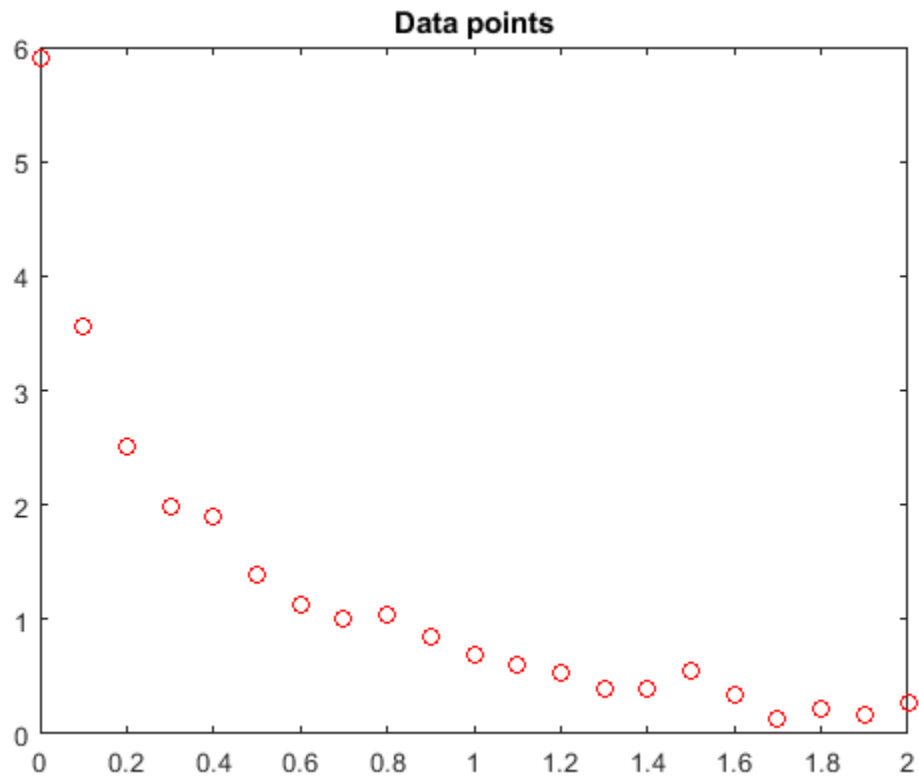
```
% hold off
```

We would like to fit the function

```
y = c(1)*exp(-lam(1)*t) + c(2)*exp(-lam(2)*t)
```

to the data.

Solution Approach Using `lsqcurvefit`

The `lsqcurvefit` function solves this type of problem easily.

To begin, define the parameters in terms of one variable x:

```
x(1) = c(1)
```

```
x(2) = lam(1)
```

```
x(3) = c(2)
```

```
x(4) = lam(2)
```

Then define the curve as a function of the parameters x and the data t:

```
F = @(x,xdata)x(1)*exp(-x(2)*xdata) + x(3)*exp(-x(4)*xdata);
```

We arbitrarily set our initial point x0 as follows: c(1) = 1, lam(1) = 1, c(2) = 1, lam(2) = 0:

```
x0 = [1 1 1 0];
```

We run the solver and plot the resulting fit.

```
[x,resnorm,~,exitflag,output] = lsqcurvefit(F,x0,t,y)
```

Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to its initial value is less than the value of the function tolerance.

x = *1×4*

    3.0068   10.5869   2.8891   1.4003

resnorm = 0.1477

exitflag = 3

output = *struct with fields:*
    firstorderopt: 7.8852e-06
       iterations: 6
        funcCount: 35
      cgiterations: 0
        algorithm: 'trust-region-reflective'
         stepsize: 0.0096
          message: '...'


```
hold on
plot(t,F(x,t))
hold off
```
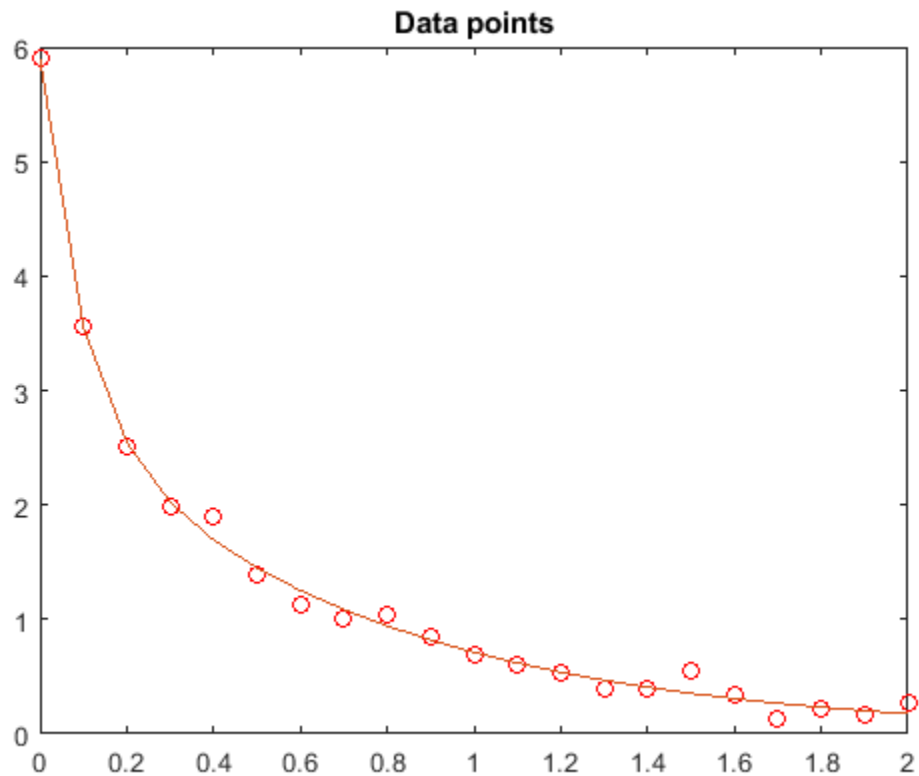
Solution Approach Using `fminunc`

To solve the problem using `fminunc`, we set the objective function as the sum of squares of the residuals.

```
Fsumsquares = @(x)sum((F(x,t) - y).^2);
opts = optimoptions('fminunc','Algorithm','quasi-newton');
[xunc,ressquared,eflag,outputu] = ...
    fminunc(Fsumsquares,x0,opts)

Local minimum found.

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

xunc = 1×4

    2.8890    1.4003    3.0069   10.5862


ressquared = 0.1477

eflag = 1

outputu = struct with fields:
       iterations: 30
        funcCount: 185
         stepsize: 0.0017
```

```
       lssteplength: 1
     firstorderopt: 2.9662e-05
         algorithm: 'quasi-newton'
           message: '...'
```

Notice that `fminunc` found the same solution as `lsqcurvefit`, but took many more function evaluations to do so. The parameters for `fminunc` are in the opposite order as those for `lsqcurvefit`; the larger lam is lam(2), not lam(1). This is not surprising, the order of variables is arbitrary.

```
fprintf(['There were %d iterations using fminunc,' ...
    ' and %d using lsqcurvefit.\n'], ...
    outputu.iterations,output.iterations)
```

```
There were 30 iterations using fminunc, and 6 using lsqcurvefit.
```

```
fprintf(['There were %d function evaluations using fminunc,' ...
    ' and %d using lsqcurvefit.'], ...
    outputu.funcCount,output.funcCount)
```

```
There were 185 function evaluations using fminunc, and 35 using lsqcurvefit.
```

Splitting the Linear and Nonlinear Problems

Notice that the fitting problem is linear in the parameters c(1) and c(2). This means for any values of lam(1) and lam(2), we can use the backslash operator to find the values of c(1) and c(2) that solve the least-squares problem.

We now rework the problem as a two-dimensional problem, searching for the best values of lam(1) and lam(2). The values of c(1) and c(2) are calculated at each step using the backslash operator as described above.

```
type fitvector
```

```
function yEst = fitvector(lam,xdata,ydata)
%FITVECTOR  Used by DATDEMO to return value of fitting function.
%   yEst = FITVECTOR(lam,xdata) returns the value of the fitting function, y
%   (defined below), at the data points xdata with parameters set to lam.
%   yEst is returned as a N-by-1 column vector, where N is the number of
%   data points.
%
%   FITVECTOR assumes the fitting function, y, takes the form
%
%     y =  c(1)*exp(-lam(1)*t) + ... + c(n)*exp(-lam(n)*t)
%
%   with n linear parameters c, and n nonlinear parameters lam.
%
%   To solve for the linear parameters c, we build a matrix A
%   where the j-th column of A is exp(-lam(j)*xdata) (xdata is a vector).
%   Then we solve A*c = ydata for the linear least-squares solution c,
%   where ydata is the observed values of y.

A = zeros(length(xdata),length(lam));  % build A matrix
for j = 1:length(lam)
   A(:,j) = exp(-lam(j)*xdata);
end
c = A\ydata; % solve A*c = y for linear parameters c
yEst = A*c; % return the estimated response based on c
```

Solve the problem using `lsqcurvefit`, starting from a two-dimensional initial point lam(1), lam(2):

```
x02 = [1 0];
F2 = @(x,t) fitvector(x,t,y);

[x2,resnorm2,~,exitflag2,output2] = lsqcurvefit(F2,x02,t,y)
```

Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

x2 = *1×2*

   10.5861   1.4003

resnorm2 = 0.1477

exitflag2 = 3

output2 = *struct with fields:*
    firstorderopt: 4.4018e-06
       iterations: 10
        funcCount: 33
     cgiterations: 0
        algorithm: 'trust-region-reflective'
         stepsize: 0.0080
          message: '...'

The efficiency of the two-dimensional solution is similar to that of the four-dimensional solution:

```
fprintf(['There were %d function evaluations using the 2-d ' ...
    'formulation, and %d using the 4-d formulation.'], ...
    output2.funcCount,output.funcCount)
```

There were 33 function evaluations using the 2-d formulation, and 35 using the 4-d formulation.

Split Problem is More Robust to Initial Guess

Choosing a bad starting point for the original four-parameter problem leads to a local solution that is not global. Choosing a starting point with the same bad lam(1) and lam(2) values for the split two-parameter problem leads to the global solution. To show this we re-run the original problem with a start point that leads to a relatively bad local solution, and compare the resulting fit with the global solution.

```
x0bad = [5 1 1 0];
[xbad,resnormbad,~,exitflagbad,outputbad] = ...
    lsqcurvefit(F,x0bad,t,y)
```

Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

xbad = *1×4*

  -22.9036   2.4792   28.0273   2.4791

12-15

```
resnormbad = 2.2173

exitflagbad = 3

outputbad = struct with fields:
    firstorderopt: 0.0057
       iterations: 32
        funcCount: 165
      cgiterations: 0
        algorithm: 'trust-region-reflective'
         stepsize: 0.0021
          message: '...'
```

```
hold on
plot(t,F(xbad,t),'g')
legend('Data','Global fit','Bad local fit','Location','NE')
hold off
```
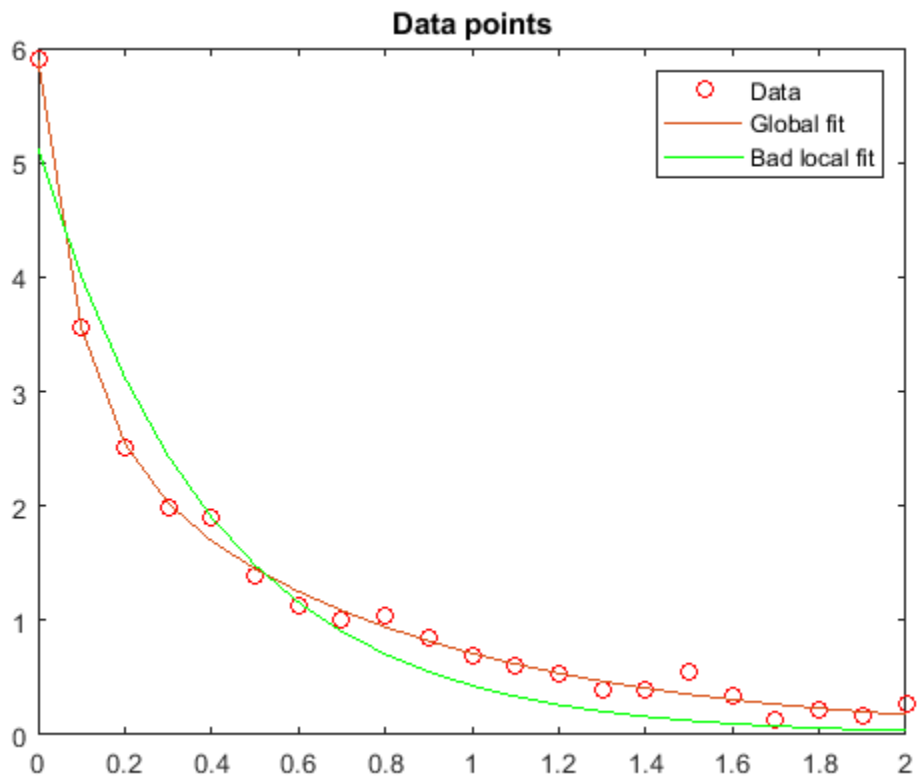


```
fprintf(['The residual norm at the good ending point is %f,' ...
    ' and the residual norm at the bad ending point is %f.'], ...
    resnorm,resnormbad)
```

The residual norm at the good ending point is 0.147723, and the residual norm at the bad ending

See Also

More About

- "Nonlinear Data-Fitting Using Several Problem-Based Approaches" on page 12-77
- "Nonlinear Least Squares Without and Including Jacobian" on page 12-22
- "Nonlinear Curve Fitting with lsqcurvefit" on page 12-48

lsqnonlin with a Simulink® Model

This example shows how to tune the parameters of a Simulink model. The model, `optsim`, ships in the `optim/demos` folder of your MATLAB® installation. The model includes a nonlinear process plant modeled as a Simulink block diagram.

Plant with Actuator Saturation

The plant is an under-damped third-order model with actuator limits. The actuator limits are a saturation limit and a slew rate limit. The actuator saturation limit cuts off input values greater than 2 units or less than –2 units. The slew rate limit of the actuator is 0.8 units/sec. The closed-loop response of the system to a step input is shown in Closed-Loop Response on page 12-0 . 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.
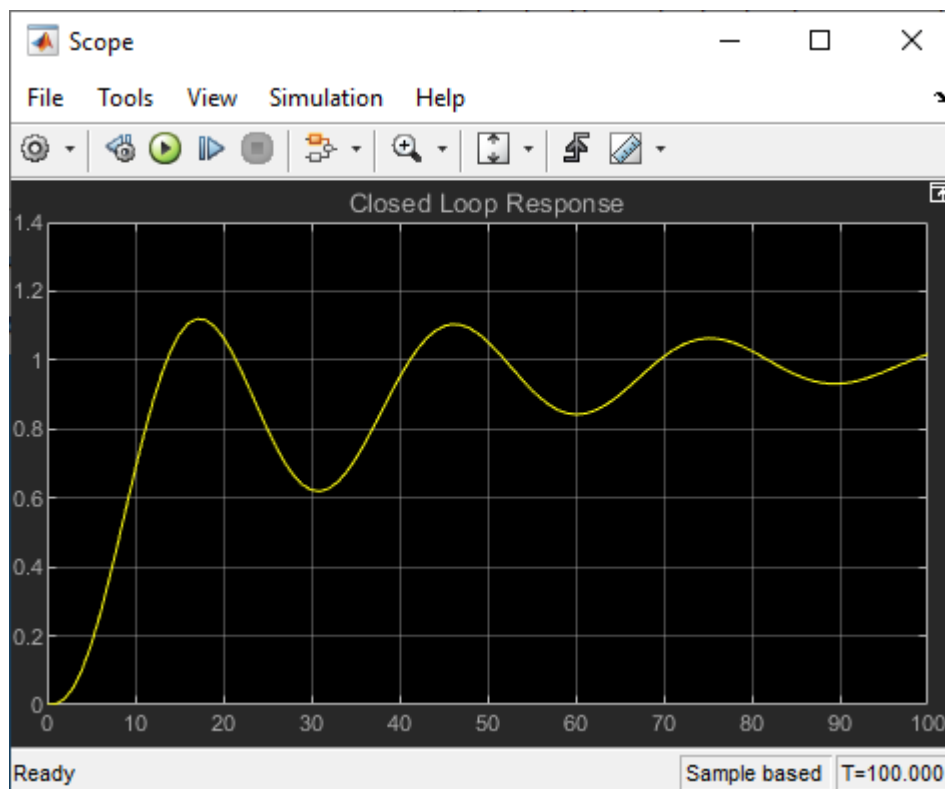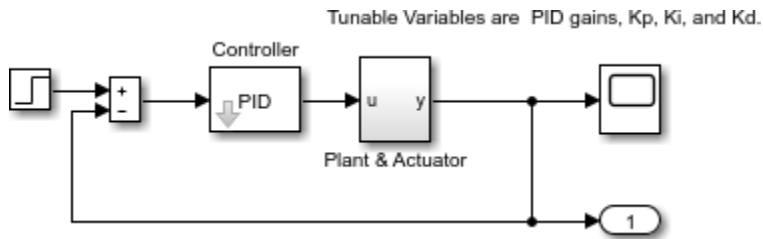
Closed-Loop Model

To solve this problem, minimize the error between the output and the input signal. (In contrast, in "Using fminimax with a Simulink® Model" on page 8-8 the solution involves minimizing maximum value of the output.) The variables are the parameters of the Proportional Integral Derivative (PID) controller. If you only need to minimize the error at one time unit, you would have a scalar 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).

Use `lsqnonlin` to perform a least-squares fit on the tracking of the output. The tracking is performed via the function `tracklsq`, which is nested in `runtracklsq` below on page 12-0   . `tracklsq` returns the error signal `yout`, the output computed by calling `sim`, minus the input signal 1.

The function `runtracklsq` sets up all the needed values and then calls `lsqnonlin` with the objective function `tracklsq`. 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. `runtracklsq` runs the simulation to 100 seconds.

When the simulation is completed, `runtracklsq` creates the `myobj` object 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.

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 = 3.1330

Ki = 0.1465

Kd = 14.3918
```

The scope shows the optimized closed-loop step response.
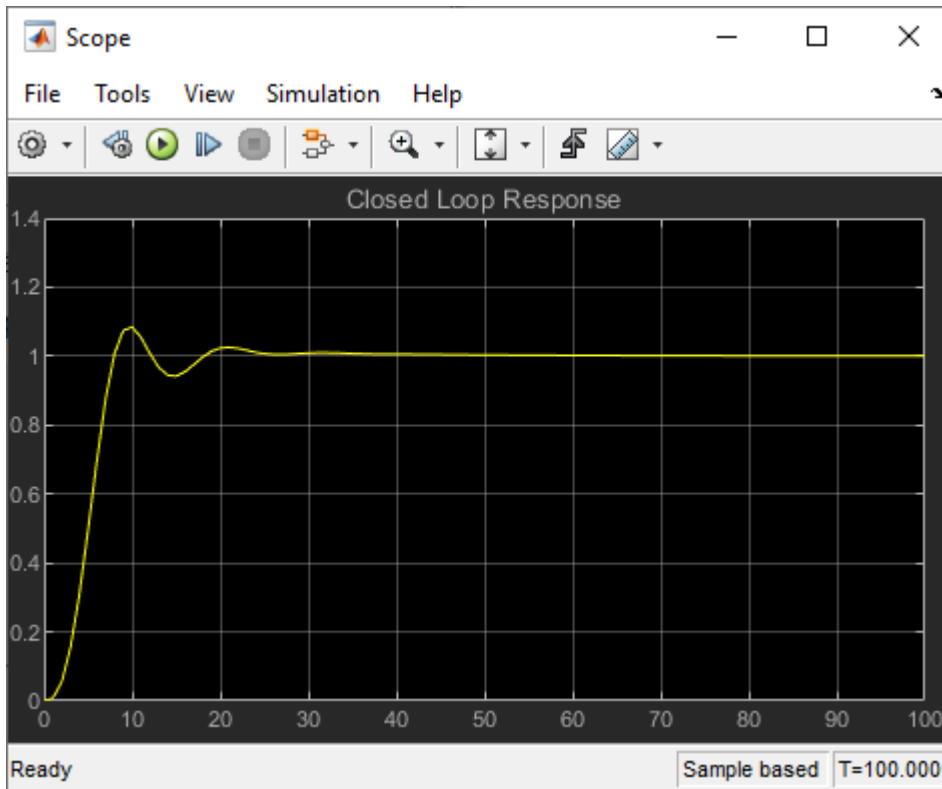
Closed-Loop Response After `lsqnonlin`

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 ODE solver is the best choice if that is sufficient to solve the ODE. However, in the case of a stiff system, a variable-step ODE 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 an 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.

The following code creates the `runtracklsq` function.

```
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','StepTolerance',0.001,'OptimalityTolerance',0.001);
pid = lsqnonlin(@tracklsq, pid0, [], [], options);
Kp = pid(1); Ki = pid(2); Kd = pid(3);
```

```matlab
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
```

Nonlinear Least Squares Without and Including Jacobian

This example shows how to solve a nonlinear least-squares problem in two ways. The example first solves the problem without using a Jacobian function. Then it shows how to include a Jacobian, and illustrates the resulting improved efficiency.

The problem has 10 terms with two 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` must compute the vector-valued function

$$F_k(x) = 2 + 2k - e^{kx_1} - e^{kx_2},$$

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

Solve Problem Without Jacobian

The helper function `myfun` defined at the end of this example on page 12-0   implements the vector-valued objective function with no derivative information. Solve the minimization starting from the point `x0`.

```
x0 = [0.3,0.4]; % Starting guess
[x,resnorm,res,eflag,output] = lsqnonlin(@myfun,x0); % Invoke optimizer
```

```
Local minimum possible.
lsqnonlin stopped because the size of the current step is less than
the value of the step size tolerance.
```

Examine the solution and number of function evaluations.

```
disp(x)
```

```
    0.2578    0.2578
```

```
disp(resnorm)
```

```
  124.3622
```

```
disp(output.funcCount)
```

```
    72
```

Solve Problem Including Jacobian

The objective function is simple enough that you can calculate its Jacobian. Following the definition in "Jacobians of Vector Functions" on page 2-26, 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

$$J_{k1}(x) = -ke^{kx_1}$$
$$J_{k2}(x) = -ke^{kx_2}.$$

The helper function `myfun2` defined at the end of this example on page 12-0   implements the objective function with the Jacobian. Set options so the solver uses the Jacobian.

```
opts = optimoptions(@lsqnonlin,'SpecifyObjectiveGradient',true);
```

Run the solver.

```
lb = []; % No bounds
ub = [];
[x2,resnorm2,res2,eflag2,output2] = lsqnonlin(@myfun2,x0,lb,ub,opts);
```

```
Local minimum possible.
lsqnonlin stopped because the size of the current step is less than
the value of the step size tolerance.
```

The solution is the same as the previous solution.

```
disp(x2)
```

```
    0.2578    0.2578
```

```
disp(resnorm2)
```

```
  124.3622
```

The advantage of using a Jacobian is that the solver takes many fewer function evaluations.

```
disp(output2.funcCount)
```

```
    24
```

Helper Functions

This code creates the `myfun` helper function.

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

This code creates the `myfun2` helper function.

```
function [F,J] = myfun2(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
end
```

See Also

More About

- "Nonlinear Data-Fitting" on page 12-10
- "Nonlinear Curve Fitting with lsqcurvefit" on page 12-48

Nonnegative Linear Least Squares, Solver-Based

This example shows how to use several algorithms to solve a linear least-squares problem with the bound constraint that the solution is nonnegative. A linear least-squares problem has the form

$$\min_{x} \|Cx - d\|^2.$$

In this case, constrain the solution to be nonnegative, $x \geq 0$.

To begin, load the arrays C and d into your workspace.

```
load particle
```

View the size of each array.

```
sizec = size(C)
```

```
sizec = 1×2

    2000         400
```

```
sized = size(d)
```

```
sized = 1×2

    2000           1
```

The `C` matrix has 2000 rows and 400 columns. Therefore, to have the correct size for matrix multiplication, the `x` vector has 400 rows. To represent the nonnegativity constraint, set lower bounds of zero on all variables.

```
lb = zeros(size(C,2),1);
```

Solve the problem using `lsqlin`.

```
[x,resnorm,residual,exitflag,output] = ...
    lsqlin(C,d,[],[],[],[],lb);
```

```
Minimum found that satisfies the constraints.

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

To see details of the optimization process, examine the output structure.

```
disp(output)
```

```
          message: '...'
        algorithm: 'interior-point'
     firstorderopt: 3.6717e-06
    constrviolation: 0
        iterations: 8
      linearsolver: 'sparse'
      cgiterations: []
```

The output structure shows that `lsqlin` uses a sparse internal linear solver for the interior-point algorithm and takes 8 iterations to reach a first-order optimality measure of about 3.7e-6.

Change Algorithm

The trust-region-reflective algorithm handles bound-constrained problems. See how well it performs on this problem.

```
options = optimoptions('lsqlin','Algorithm','trust-region-reflective');
[x2,resnorm2,residual2,exitflag2,output2] = ...
    lsqlin(C,d,[],[],[],[],lb,[],[],options);
```

```
Local minimum possible.

lsqlin stopped because the relative change in function value is less than the square root of the
```

```
disp(output2)

        iterations: 10
         algorithm: 'trust-region-reflective'
      firstorderopt: 2.7870e-05
       cgiterations: 42
     constrviolation: []
        linearsolver: []
             message: 'Local minimum possible....'
```

This time, the solver takes more iterations and reaches a solution with a higher (worse) first-order optimality measure.

To improve the first-order optimality measure, try setting the `SubproblemAlgorithm` option to `'factorization'`.

```
options.SubproblemAlgorithm = 'factorization';
[x3,resnorm3,residual3,exitflag3,output3] = ...
    lsqlin(C,d,[],[],[],[],lb,[],[],options);
```

```
Optimal solution found.
```

```
disp(output3)

        iterations: 12
         algorithm: 'trust-region-reflective'
      firstorderopt: 5.5907e-15
       cgiterations: 0
     constrviolation: []
        linearsolver: []
             message: 'Optimal solution found.'
```

Using this option brings the first-order optimality measure nearly to zero, which is the best possible result.

Change Solver

Try solving t problem using the `lsqnonneg` solver, which is designed to handle nonnegative linear least squares.

```
[x4,resnorm4,residual4,exitflag4,output4] = lsqnonneg(C,d);
disp(output4)
```

```
        iterations: 184
         algorithm: 'active-set'
           message: 'Optimization terminated.'
```

`lsqnonneg` does not report a first-order optimality measure. Instead, investigate the residual norms. To see the lower-significance digits, subtract 22.5794 from each residual norm.

```
t = table(resnorm - 22.5794, resnorm2 - 22.5794, resnorm3 - 22.5794, resnorm4 - 22.5794,...
    'VariableNames',{'default','trust-region-reflective','factorization','lsqnonneg'})
```

t=*1×4 table*

default	trust-region-reflective	factorization	lsqnonneg
7.5411e-05	4.9186e-05	4.9179e-05	4.9179e-05

The default `lsqlin` algorithm has a higher residual norm than the `trust-region-reflective` algorithm. The `factorization` and `lsqnonneg` residual norms are even lower, and are the same at this level of display precision. See which one is lower.

```
disp(resnorm3 - resnorm4)
```

```
   6.7857e-13
```

The `lsqnonneg` residual norm is the lowest by a negligible amount. However, `lsqnonneg` takes the most iterations to converge.

See Also
`lsqlin` | `lsqnonneg`

More About

Optimize Live Editor Task with lsqlin Solver

This example shows how to use the **Optimize** Live Editor task to solve a constrained least-squares problem.

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

$$x_1 + 2x_2 + 4x_3 = 7. \tag{12-16}$$

The function $f(x)$ is 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.

Set Up and Solve the Problem Using Optimize

Set up the problem with the `lsqlin` solver in the **Optimize** Live Editor task.

1. Create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.
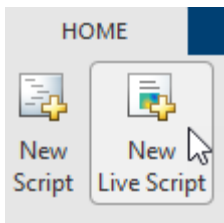
2. Insert an **Optimize** Live Editor task. Click the **Insert** tab and then, in the **Code** section, select **Task > Optimize**.
3. In the **Specify problem type** section of the task, select **Objective > Least squares** and **Constraints > Linear equality**.

   The task selects `lsqlin` as the recommended solver.
4. To get the data C and d into the MATLAB workspace, click the **Section Break** button on the **Insert** tab. In the new section, enter the following code.

   ```
   C = eye(3);
   d = zeros(3,1);
   ```
5. Set the linear equality constraint matrix and vector.

   ```
   Aeq = [1 2 4];
   beq = 7;
   ```
6. Run the section by pressing **Ctrl+Enter**. This places the variables into the workspace.
7. In the **Select problem data** section of the task, set the entries to their corresponding values.

Select problem data

Multiplier matrix (C) `C ▼` Constant vector (d) `d ▼` Initial point (x0) `select ▼`

Constraints Linear equality Aeq `Aeq ▼` *x = beq `beq ▼`

8 Run the solver by pressing **Ctrl+Enter**. View the exit message.

```
Minimum found that satisfies the constraints.

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

<stopping criteria details>
```

9 To find the solution, look at the top of the task.

Optimize

`solution`, `objectiveValue` = Minimize norm(**C***x - **d**)^2 using **lsqlin** solver

The solver returns the variables `solution` and `objectiveValue` to the MATLAB workspace.

10 Insert a section break below the task. Place these lines in the new section.

```
disp(solution)
disp(objectiveValue)
```

11 Run the section by pressing **Ctrl+Enter**.

```
disp(solution)

    0.3333
    0.6667
    1.3333
```

```
disp(objectiveValue)

    2.3333
```

See Also

Optimize | `lsqlin`

More About

- "Shortest Distance to a Plane" on page 12-38
- "Use Optimize Live Editor Task Effectively" on page 1-39

Jacobian Multiply Function with Linear Least Squares

Using a Jacobian multiply function, you can solve a least-squares problem of the form

$$\min_{x} \frac{1}{2} \| C \cdot x - d \|_2^2$$

such that `lb` $\leq$ `x` $\leq$ `ub`, for problems where C is very large, perhaps too large to be stored. For this technique, use the `'trust-region-reflective'` algorithm.

For example, consider a problem where C is a 2n-by-n matrix based on a circulant matrix. The rows of C are shifts of a row vector v. This example has the row vector v with elements of the form $(-1)^{k+1}/k$:

$v = [1, -1/2, 1/3, -1/4, ..., -1/n],$

where the elements are cyclically shifted.

$$C = \begin{bmatrix} 1 & -1/2 & 1/3 & \cdots & -1/n \\ -1/n & 1 & -1/2 & \cdots & 1/(n-1) \\ 1/(n-1) & -1/n & 1 & \cdots & -1/(n-2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1/2 & 1/3 & -1/4 & \cdots & 1 \\ 1 & -1/2 & 1/3 & \cdots & -1/n \\ -1/n & 1 & -1/2 & \cdots & 1/(n-1) \\ 1/(n-1) & -1/n & 1 & \cdots & -1/(n-2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1/2 & 1/3 & -1/4 & \cdots & 1 \end{bmatrix}.$$

This least-squares example considers the problem where

$d = [n - 1, n - 2, ..., -n],$

and the constraints are $-5 \leq x_i \leq 5$ for $i = 1, ..., n$.

For large enough n, the dense matrix C does not fit into computer memory ($n = 10,000$ is too large on one tested system).

A Jacobian multiply function has the following syntax.

`w = jmfcn(Jinfo,Y,flag)`

`Jinfo` is a matrix the same size as C, used as a preconditioner. If C is too large to fit into memory, `Jinfo` should be sparse. `Y` is a vector or matrix sized so that `C*Y` or `C'*Y` works as matrix multiplication. `flag` tells `jmfcn` which product to form:

- `flag > 0` $\Rightarrow$ `w = C*Y`
- `flag < 0` $\Rightarrow$ `w = C'*Y`
- `flag = 0` $\Rightarrow$ `w = C'*C*Y`

Because *C* is such a simply structured matrix, you can easily write a Jacobian multiply function in terms of the vector *v*; that is, without forming *C*. Each row of C*Y is the product of a circularly shifted version of *v* times Y. Use `circshift` to circularly shift *v*.

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),fliplr(v)];
```

```
temp = [circshift(temp,1,2),circshift(temp,1,2)]; % Now temp = C'(1,:)
```

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 helper function `lsqcirculant3` is a Jacobian multiply function that implements this procedure; it appears at the end of this example on page 12-0 .

The `dolsqJac3` helper function at the end of this example on page 12-0 sets up the vector *v* and calls the solver `lsqlin` using the `lsqcirculant3` Jacobian multiply function.

When n = 3000, *C* is an 18,000,000-element dense matrix. Determine the results of the `dolsqJac3` function for n = 3000 at selected values of *x*, and display the output structure.

```
[x,resnorm,residual,exitflag,output] = dolsqJac3(3000);
```

Local minimum possible.

lsqlin stopped because the relative change in function value is less than the function tolerance

```
disp(x(1))
```

```
    5.0000
```

```
disp(x(1500))
```

```
   -0.5201
```

```
disp(x(3000))
```

```
   -5.0000
```

```
disp(output)
```

```
        iterations: 16
         algorithm: 'trust-region-reflective'
     firstorderopt: 5.9351e-05
       cgiterations: 36
    constrviolation: []
       linearsolver: []
            message: 'Local minimum possible.↵↵lsqlin stopped because the relative change in func
```

Helper Functions

This code creates the `lsqcirculant3` helper function.

```
function w = lsqcirculant3(Jinfo,Y,flag,v)
% This function computes the Jacobian multiply function
```

```matlab
% 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 = circshift(temp,1,2); % Shift the circulant
        end
    end

    function a = Jnegative(q)
        % Calculate C'*q
        temp = fliplr(v);
        temp = circshift(temp,1,2); % Shift 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 = circshift(temp,1,2); % Shift the circulant
        end
    end
end
```

This code creates the `dolsqJac3` helper function.

```matlab
function [x,resnorm,residual,exitflag,output] = dolsqJac3(n)
%
r = 1:n-1; % Index for making vectors

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 it 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 used in this example.
```

```matlab
% Pass the vector v so that it does not need to be
% computed in the Jacobian multiply function.
options = optimoptions('lsqlin','Algorithm','trust-region-reflective',...
    'JacobianMultiplyFcn',@(Jinfo,Y,flag)lsqcirculant3(Jinfo,Y,flag,v));

lb = -5*ones(1,n);
ub = 5*ones(1,n);

[x,resnorm,residual,exitflag,output] = ...
    lsqlin(Jinfo,d,[],[],[],[],lb,ub,[],options);
end
```

See Also

```
circshift | fliplr
```

More About

- "Quadratic Minimization with Dense, Structured Hessian" on page 11-24

Large-Scale Constrained Linear Least-Squares, Solver-Based

This example shows how to recover a blurred image by solving a large-scale bound-constrained linear least-squares optimization problem. The example uses the solver-based approach. For the problem-based approach, see "Large-Scale Constrained Linear Least-Squares, Problem-Based" on page 12-44.

The Problem

Here is a photo of people sitting in a car having an interesting license plate.

```
load optdeblur
[m,n] = size(P);
mn = m*n;
imshow(P)
title(sprintf('Original Image, size %d-by-%d, %d pixels',m,n,mn))
```


Original Image, size 149-by-311, 46339 pixels

The problem is to take a blurred version of this photo and try to deblur it. The starting image is black and white, meaning it consists of pixel values from 0 through 1 in the m x n matrix P.

Add Motion

Simulate the effect of vertical motion blurring by averaging each pixel with the 5 pixels above and below. Construct a sparse matrix D to blur with a single matrix multiply.

```
blur = 5;  mindex = 1:mn;  nindex = 1:mn;
for i = 1:blur
  mindex=[mindex i+1:mn 1:mn-i];
  nindex=[nindex 1:mn-i i+1:mn];
end
D = sparse(mindex,nindex,1/(2*blur+1));
```

Draw a picture of D.

```
cla
axis off ij
xs = 31;
ys = 15;
xlim([0,xs+1]);
```

```
ylim([0,ys+1]);
[ix,iy] = meshgrid(1:(xs-1),1:(ys-1));
l = abs(ix-iy)<=5;
text(ix(l),iy(l),'x')
text(ix(~l),iy(~l),'0')
text(xs*ones(ys,1),1:ys,'...');
text(1:xs,ys*ones(xs,1),'...');
title('Blurring Operator D (x = 1/11)')
```


Blurring Operator D (x = 1/11)

Multiply the image P by the matrix D to create a blurred image G.

```
G = D*(P(:));
figure
imshow(reshape(G,m,n));
title('Blurred Image')
```


Blurred Image

The image is much less distinct; you can no longer read the license plate.

Deblurred Image

To deblur, suppose that you know the blurring operator D. How well can you remove the blur and recover the original image P?

The simplest approach is to solve a least squares problem for x:

$\min(\|Dx - G\|^2)$ subject to $0 \le x \le 1$.

This problem takes the blurring matrix D as given, and tries to find the x that makes Dx closest to $G = DP$. In order for the solution to represent sensible pixel values, restrict the solution to be from 0 through 1.

```
lb = zeros(mn,1);
ub = 1 + lb;
sol = lsqlin(D,G,[],[],[],[],lb,ub);
```

```
Minimum found that satisfies the constraints.

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

```
xpic = reshape(sol,m,n);
figure
imshow(xpic)
title('Deblurred Image')
```


Deblurred Image

The deblurred image is much clearer than the blurred image. You can once again read the license plate. However, the deblurred image has some artifacts, such as horizontal bands in the lower-right pavement region. Perhaps these artifacts can be removed by a regularization.

Regularization

Regularization is a way to smooth the solution. There are many regularization methods. For a simple approach, add a term to the objective function as follows:

$\min(\|(D + \varepsilon I)x - G\|^2)$ subject to $0 \le x \le 1$.

The term εI makes the resulting quadratic problem more stable. Take $\varepsilon = 0.02$ and solve the problem again.

```
addI = speye(mn);
sol2 = lsqlin(D+0.02*addI,G,[],[],[],[],lb,ub);
```

```
Minimum found that satisfies the constraints.

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

```
xpic2 = reshape(sol2,m,n);
figure
imshow(xpic2)
title('Deblurred Regularized Image')
```


Deblurred Regularized Image

Apparently, this simple regularization does not remove the artifacts.

See Also

More About

- "Large-Scale Constrained Linear Least-Squares, Problem-Based" on page 12-44

Shortest Distance to a Plane

The Problem

This example shows how to formulate a linear least squares problem using the problem-based approach.

The problem is to find the shortest distance from the origin (the point `[0,0,0]`) to the plane $x_1 + 2x_2 + 4x_3 = 7$. In other words, this problem is to minimize $f(x) = x_1^2 + x_2^2 + x_3^2$ subject to the constraint $x_1 + 2x_2 + 4x_3 = 7$. The function $f(x)$ is called the *objective function* and $x_1 + 2x_2 + 4x_3 = 7$ is an *equality constraint*. More complicated problems might contain other equality constraints, inequality constraints, and upper or lower bound constraints.

Set Up the Problem

To formulate this problem using the problem-based approach, create an optimization problem object called `pointtoplane`.

```
pointtoplane = optimproblem;
```

Create a problem variable x as a continuous variable with three components.

```
x = optimvar('x',3);
```

Create the objective function and put it in the `Objective` property of `pointtoplane`.

```
obj = sum(x.^2);
pointtoplane.Objective = obj;
```

Create the linear constraint and put it in the problem.

```
v = [1,2,4];
pointtoplane.Constraints = dot(x,v) == 7;
```

The problem formulation is complete. To check for errors, review the problem.

```
show(pointtoplane)

  OptimizationProblem :

    Solve for:
       x

    minimize :
       sum(x.^2)


    subject to :
       x(1) + 2*x(2) + 4*x(3) == 7
```

The formulation is correct.

Solve the Problem

Solve the problem by calling `solve`.

```
[sol,fval,exitflag,output] = solve(pointtoplane);
```

```
Solving problem using lsqlin.
```

```
Minimum found that satisfies the constraints.
```

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

```
disp(sol.x)
```

```
    0.3333
    0.6667
    1.3333
```

Verify the Solution

To verify the solution, solve the problem analytically. Recall that for any nonzero `t`, the vector `t*[1,2,4] = t*v` is perpendicular to the plane $x_1 + 2x_2 + 4x_3 = 7$. So the solution point `xopt` is `t*v` for the value of `t` that satisfies the equation `dot(t*v,v) = 7`.

```
t = 7/dot(v,v)
```

```
t = 0.3333
```

```
xopt = t*v
```

```
xopt = 1×3
```

```
    0.3333    0.6667    1.3333
```

Indeed, the vector `xopt` is equivalent to the point `sol.x` that `solve` finds.

See Also

More About

Nonnegative Linear Least Squares, Problem-Based

This example shows how to use several algorithms to solve a linear least squares problem with the bound constraint that the solution is nonnegative. A linear least squares problem has the form

$$\min_{x} \|Cx - d\|^2.$$

In this case, constrain the solution to be nonnegative, $x \geq 0$.

To begin, load the arrays C and d into your workspace.

```
load particle
```

View the size of each array.

```
sizec = size(C)
```

```
sizec = 1×2

     2000          400
```

```
sized = size(d)
```

```
sized = 1×2

     2000            1
```

Create an optimization variable x of the appropriate size for multiplication by C. Impose a lower bound of 0 on the elements of x.

```
x = optimvar('x',sizec(2),'LowerBound',0);
```

Create the objective function expression.

```
residual = C*x - d;
obj = sum(residual.^2);
```

Create an optimization problem called nonneglsq and include the objective function in the problem.

```
nonneglsq = optimproblem('Objective',obj);
```

Find the default solver for the problem.

```
opts = optimoptions(nonneglsq)
```

```
opts =
  lsqlin options:

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

   Set properties:
     No options set.

   Default properties:
```

```
             Algorithm: 'interior-point'
   ConstraintTolerance: 1.0000e-08
               Display: 'final'
          LinearSolver: 'auto'
         MaxIterations: 200
    OptimalityTolerance: 1.0000e-08
         StepTolerance: 1.0000e-12

   Show options not used by current Algorithm ('interior-point')
```

Solve the problem using the default solver.

```
[sol,fval,exitflag,output] = solve(nonneglsq);
```

Solving problem using lsqlin.

Minimum found that satisfies the constraints.

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

To see details of the optimization process, examine the output structure.

```
disp(output)
```

```
            message: '...'
          algorithm: 'interior-point'
      firstorderopt: 9.9673e-07
      constrviolation: 0
         iterations: 9
       linearsolver: 'sparse'
        cgiterations: []
              solver: 'lsqlin'
```

The output structure shows that the `lsqlin` solver uses a sparse internal linear solver for the interior-point algorithm and takes 9 iterations to arrive at a first-order optimality measure of about 1e-6.

Change Algorithm

The trust-region-reflective algorithm handles bound-constrained problems. See how well it performs on this problem.

```
opts.Algorithm = 'trust-region-reflective';
[sol2,fval2,exitflag2,output2] = solve(nonneglsq,'Options',opts);
```

Solving problem using lsqlin.

Local minimum possible.

lsqlin stopped because the relative change in function value is less than the function tolerance

```
disp(output2)
```

```
         iterations: 14
          algorithm: 'trust-region-reflective'
      firstorderopt: 5.2187e-08
```

```
           cgiterations: 54
         constrviolation: []
           linearsolver: []
                 message: 'Local minimum possible....'
                  solver: 'lsqlin'
```

This time, the solver takes more iterations and arrives at a solution with a lower (better) first-order optimality measure.

To get an even better first-order optimality measure, try setting the `SubproblemAlgorithm` option to `'factorization'`.

```
opts.SubproblemAlgorithm = 'factorization';
[sol3,fval3,exitflag3,output3] = solve(nonneglsq,'Options',opts);
```

Solving problem using lsqlin.

Optimal solution found.

```
disp(output3)
```

```
              iterations: 11
               algorithm: 'trust-region-reflective'
            firstorderopt: 1.3973e-14
              cgiterations: 0
           constrviolation: []
              linearsolver: []
                   message: 'Optimal solution found.'
                    solver: 'lsqlin'
```

Using this option brings the first-order optimality measure nearly to zero, which is the best possible.

Change Solver

The `lsqnonneg` solver is specially designed to handle nonnegative linear least squares. Try that solver.

```
[sol4,fval4,exitflag4,output4] = solve(nonneglsq,'Solver','lsqnonneg');
```

Solving problem using lsqnonneg.

```
disp(output4)
```

```
     iterations: 184
      algorithm: 'active-set'
        message: 'Optimization terminated.'
         solver: 'lsqnonneg'
```

`lsqnonneg` does not report a first-order optimality measure. Instead, investigate the residual norms, which are returned in the `fval` outputs. To see the lower-significance digits, subtract 22.5794 from each residual norm.

```
t = table(fval - 22.5794, fval2 - 22.5794, fval3 - 22.5794, fval4 - 22.5794,...
    'VariableNames',{'default','trust-region-reflective','factorization','lsqnonneg'})
```

t=1×4 table

default	trust-region-reflective	factorization	lsqnonneg

```
   5.0804e-05          4.9179e-05          4.9179e-05       4.9179e-05
```

The default solver has a slightly higher (worse) residual norm than the other three, whose residual norms are indistinguishable at this level of display precision. To see which is lowest, subtract the `lsqnonneg` result from the two results.

```
disp([fval2 - fval4,fval3 - fval4])

   1.0e-12 *

   0.7105    0.6999
```

The `lsqnonneg` residual norm is the smallest by a nearly negligible amount. However, `lsqnonneg` takes the most iterations to converge.

See Also

More About

- "Nonnegative Linear Least Squares, Solver-Based" on page 12-25
- "Problem-Based Optimization Workflow" on page 10-2

Large-Scale Constrained Linear Least-Squares, Problem-Based

This example shows how to recover a blurred image by solving a large-scale bound-constrained linear least-squares optimization problem. The example uses the problem-based approach. For the solver-based approach, see "Large-Scale Constrained Linear Least-Squares, Solver-Based" on page 12-34.

The Problem

Here is a photo of people sitting in a car having an interesting license plate.

```
load optdeblur
[m,n] = size(P);
mn = m*n;
figure
imshow(P);
colormap(gray);
axis off image;
title([int2str(m) ' x ' int2str(n) ' (' int2str(mn) ') pixels'])
```


The problem is to take a blurred version of this photo and try to deblur it. The starting image is black and white, meaning it consists of pixel values from 0 through 1 in the m x n matrix P.

Add Motion

Simulate the effect of vertical motion blurring by averaging each pixel with the 5 pixels above and below.Construct a sparse matrix D to blur with a single matrix multiply.

```
blur = 5;
mindex = 1:mn;
nindex = 1:mn;
for i = 1:blur
  mindex=[mindex i+1:mn 1:mn-i];
  nindex=[nindex 1:mn-i i+1:mn];
end
D = sparse(mindex,nindex,1/(2*blur+1));
```

Draw a picture of D.

```
cla
axis off ij
```

```
xs = 31;
ys = 15;
xlim([0,xs+1]);
ylim([0,ys+1]);
[ix,iy] = meshgrid(1:(xs-1),1:(ys-1));
l = abs(ix-iy) <= blur;
text(ix(l),iy(l),'x')
text(ix(~l),iy(~l),'0')
text(xs*ones(ys,1),1:ys,'...');
text(1:xs,ys*ones(xs,1),'...');
title('Blurring Operator D (x = 1/11)')
```


Blurring Operator D (x = 1/11)

Multiply the image P by the matrix D to create a blurred image G.

```
G = D*(P(:));
figure
imshow(reshape(G,m,n));
```


The image is much less distinct; you can no longer read the license plate.

Deblurred Image

To deblur, suppose that you know the blurring operator D. How well can you remove the blur and recover the original image P?

The simplest approach is to solve a least squares problem for *x*:

$\min(\|Dx - G\|^2)$ subject to $0 \leq x \leq 1$.

This problem takes the blurring matrix *D* as given, and tries to find the *x* that makes *Dx* closest to *G* = *DP*. In order for the solution to represent sensible pixel values, restrict the solution to be from 0 through 1.

```
x = optimvar('x',mn,'LowerBound',0,'UpperBound',1);
expr = D*x-G;
objec = expr'*expr;
blurprob = optimproblem('Objective',objec);
sol = solve(blurprob);
```

```
Solving problem using quadprog.

Minimum found that satisfies the constraints.

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

```
xpic = reshape(sol.x,m,n);
figure
imshow(xpic)
title('Deblurred Image')
```


Deblurred Image

The deblurred image is much clearer than the blurred image. You can once again read the license plate. However, the deblurred image has some artifacts, such as horizontal bands in the lower-right pavement region. Perhaps these artifacts can be removed by a regularization.

Regularization

Regularization is a way to smooth the solution. There are many regularization methods. For a simple approach, add a term to the objective function as follows:

$\min(\|(D + \varepsilon I)x - G\|^2)$ subject to $0 \leq x \leq 1$.

The term εI makes the resulting quadratic problem more stable. Take $\varepsilon = 0.02$ and solve the problem again.

```
addI = speye(mn);
expr2 = (D + 0.02*addI)*x - G;
objec2 = expr2'*expr2;
blurprob2 = optimproblem('Objective',objec2);
sol2 = solve(blurprob2);
```

```
Solving problem using quadprog.

Minimum found that satisfies the constraints.

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

```
xpic2 = reshape(sol2.x,m,n);
figure
imshow(xpic2)
title('Deblurred Regularized Image')
```


Deblurred Regularized Image

Apparently, this simple regularization does not remove the artifacts.

See Also

More About

- "Large-Scale Constrained Linear Least-Squares, Solver-Based" on page 12-34
- "Problem-Based Optimization Workflow" on page 10-2

Nonlinear Curve Fitting with lsqcurvefit

lsqcurvefit enables you to fit parameterized nonlinear functions to data easily. You can also use lsqnonlin; 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 for the problem.

```
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

$$\text{ydata}_i = a_1 + a_2 \exp(-a_3 \, \text{xdata}_i) + \varepsilon_i.$$

The code generates xdata from 100 independent samples of an exponential distribution with mean 2. The code generates ydata from its defining equation 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.

```
a0 = [2;2;2];
```

Call lsqcurvefit to find the best-fitting parameters $\hat{a}_i$.

```
[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 value of the function tolerance.
```

Examine the resulting parameters.

```
disp(ahat)
```

```
    1.0169
    3.1444
    2.1596
```

The fitted values ahat are within 8% of a = [1;3;2].

If you have the Statistics and Machine Learning Toolbox™ software, use the nlparci function to generate confidence intervals for the ahat estimate.

See Also
`lsqcurvefit` | `nlparci`

More About

- "Nonlinear Data-Fitting" on page 12-10
- "Nonlinear Least Squares Without and Including Jacobian" on page 12-22

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, least-squares solvers and `fsolve` can work on both real-valued and complex-valued data for unconstrained problems. The objective function must be analytic in the complex function sense.

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. The data model is analytic, so you can use it in a complex solution.

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`.

```
opts = optimoptions(@lsqnonlin,'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 =

   2.1582 + 0.1351i
   2.7399 + 3.8012i
  -0.5338 + 0.4660i


resnorm =

  100.9933
```

```
exitflag =

     3


ans =

    0.0018
```

`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 =

    2.1582 + 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 include bounds, or simply to stay completely within real values, you can split the real and complex parts of the coefficients into separate variables. For this problem, split the coefficients as follows:

$$\begin{aligned}
y &= v_1 + iv_2 + (v_3 + iv_4)\exp\left((v_5 + iv_6)x\right) \\
&= \left(v_1 + v_3\exp(v_5 x)\cos(v_6 x) - v_4\exp(v_5 x)\sin(v_6 x)\right) \\
&\quad + i\left(v_2 + v_4\exp(v_5 x)\cos(v_6 x) + v_3\exp(v_5 x)\sin(v_6 x)\right).
\end{aligned}$$

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
```

```
Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.


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.

See Also

More About

- "Complex Numbers in Optimization Toolbox Solvers" on page 2-14

Fit an Ordinary Differential Equation (ODE)

This example shows how to fit parameters of an ODE to data in two ways. The first shows a straightforward fit of a constant-speed circular path to a portion of a solution of the Lorenz system, a famous ODE with sensitive dependence on initial parameters. The second shows how to modify the parameters of the Lorenz system to fit a constant-speed circular path. You can use the appropriate approach for your application as a model for fitting a differential equation to data.

Lorenz System: Definition and Numerical Solution

The Lorenz system is a system of ordinary differential equations (see Lorenz system). For real constants $\sigma, \ \rho, \ \beta$, the system is

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z \,.$$

Lorenz's values of the parameters for a sensitive system are $\sigma = 10, \ \beta = 8/3, \ \rho = 28$. Start the system from $[x(0),y(0),z(0)] = [10,20,10]$ and view the evolution of the system from time 0 through 100.

```
sigma = 10;
beta = 8/3;
rho = 28;
f = @(t,a) [-sigma*a(1) + sigma*a(2); rho*a(1) - a(2) - a(1)*a(3); -beta*a(3) + a(1)*a(2)];
xt0 = [10,20,10];
[tspan,a] = ode45(f,[0 100],xt0);     % Runge-Kutta 4th/5th order ODE solver
figure
plot3(a(:,1),a(:,2),a(:,3))
view([-10.0 -2.0])
```
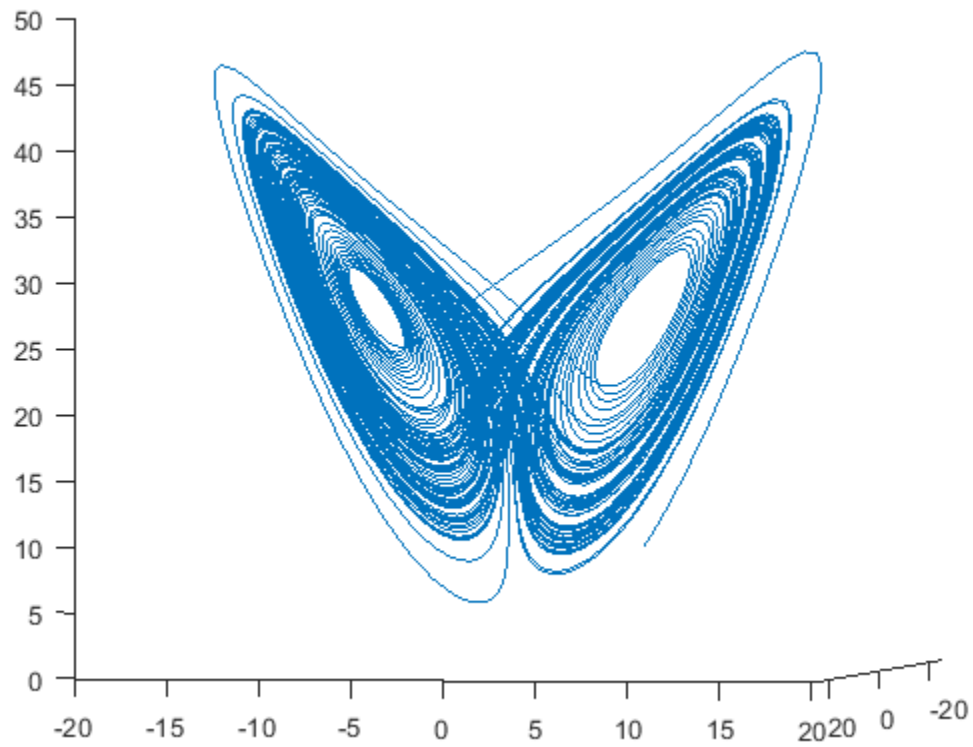
The evolution is quite complicated. But over a small time interval, it looks somewhat like uniform circular motion. Plot the solution over the time interval [0,1/10].

```
[tspan,a] = ode45(f,[0 1/10],xt0);      % Runge-Kutta 4th/5th order ODE solver
figure
plot3(a(:,1),a(:,2),a(:,3))
view([-30 -70])
```
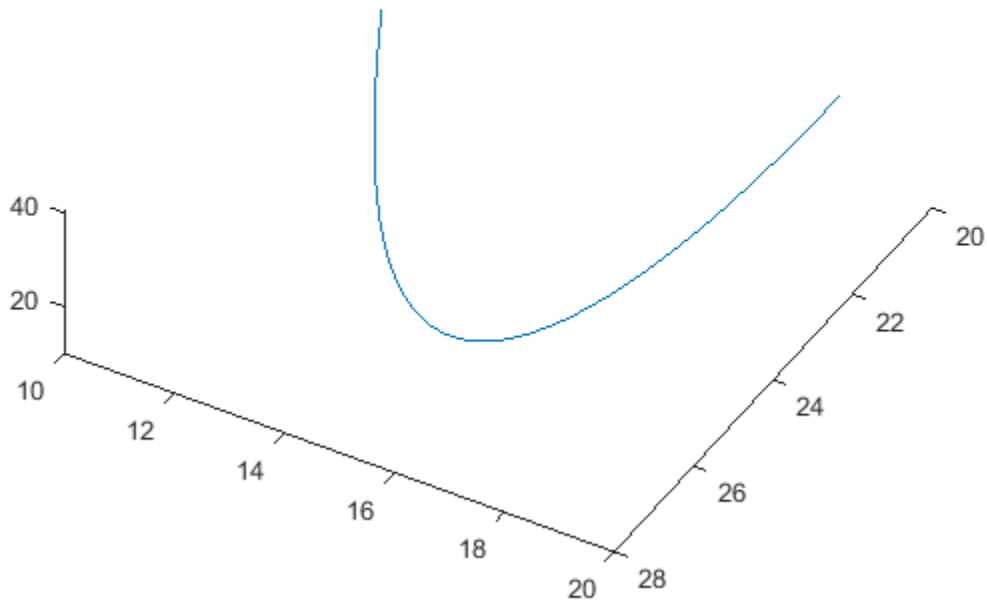
Fit a Circular Path to the ODE Solution

The equations of a circular path have several parameters:

- Angle $\theta(1)$ of the path from the x-y plane
- Angle $\theta(2)$ of the plane from a tilt along the x-axis
- Radius R
- Speed V
- Shift $t0$ from time 0
- 3-D shift in space *delta*

In terms of these parameters, determine the position of the circular path for times `xdata`.

```
type fitlorenzfn
```

```
function f = fitlorenzfn(x,xdata)

theta = x(1:2);
R = x(3);
V = x(4);
t0 = x(5);
delta = x(6:8);
f = zeros(length(xdata),3);
f(:,3) = R*sin(theta(1))*sin(V*(xdata - t0)) + delta(3);
f(:,1) = R*cos(V*(xdata - t0))*cos(theta(2)) ...
```

```
    - R*sin(V*(xdata - t0))*cos(theta(1))*sin(theta(2)) + delta(1);
f(:,2) = R*sin(V*(xdata - t0))*cos(theta(1))*cos(theta(2)) ...
    - R*cos(V*(xdata - t0))*sin(theta(2)) + delta(2);
```

To find the best-fitting circular path to the Lorenz system at times given in the ODE solution, use `lsqcurvefit`. In order to keep the parameters in reasonable limits, put bounds on the various parameters.

```
lb = [-pi/2,-pi,5,-15,-pi,-40,-40,-40];
ub = [pi/2,pi,60,15,pi,40,40,40];
theta0 = [0;0];
R0 = 20;
V0 = 1;
t0 = 0;
delta0 = zeros(3,1);
x0 = [theta0;R0;V0;t0;delta0];
[xbest,resnorm,residual] = lsqcurvefit(@fitlorenzfn,x0,tspan,a,lb,ub);
```

```
Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.
```

Plot the best-fitting circular points at the times from the ODE solution together with the solution of the Lorenz system.

```
soln = a + residual;
hold on
plot3(soln(:,1),soln(:,2),soln(:,3),'r')
legend('ODE Solution','Circular Arc')
hold off
```
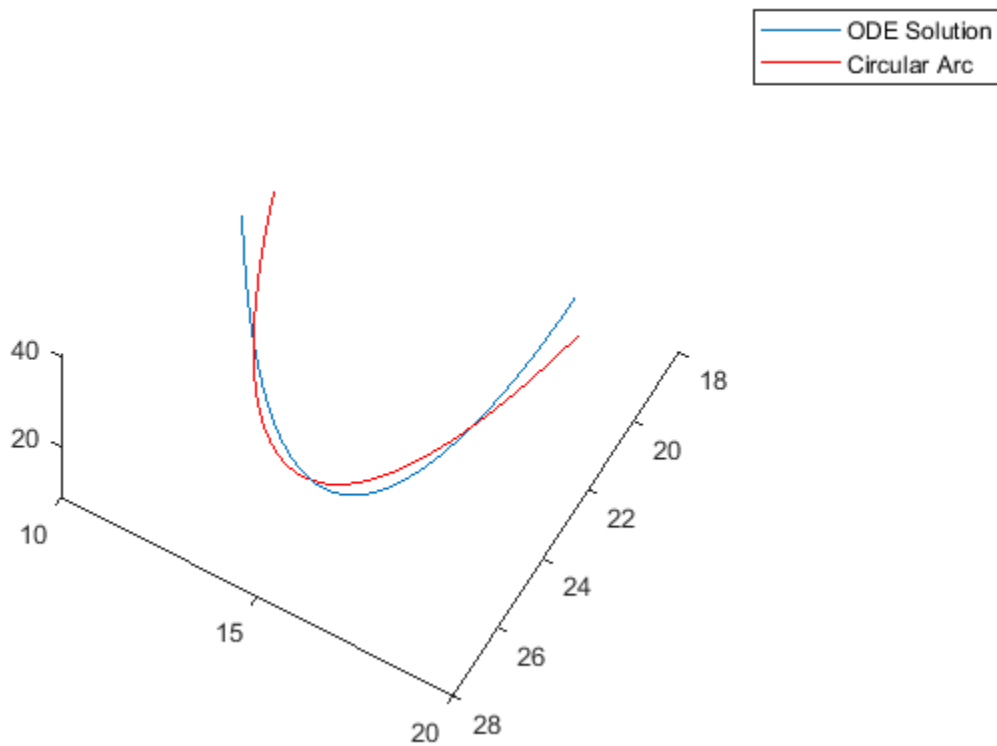
```
figure
plot3(a(:,1),a(:,2),a(:,3),'b.','MarkerSize',10)
hold on
plot3(soln(:,1),soln(:,2),soln(:,3),'rx','MarkerSize',10)
legend('ODE Solution','Circular Arc')
hold off
```
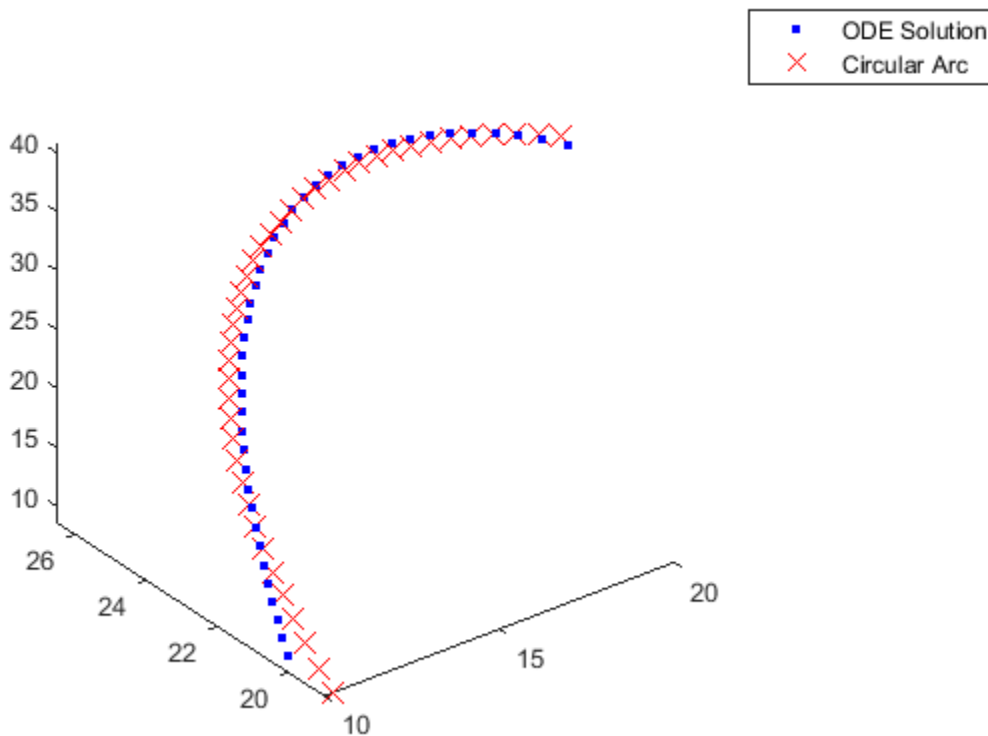
Fit the ODE to the Circular Arc

Now modify the parameters σ, β, and ρ to best fit the circular arc. For an even better fit, allow the initial point [10,20,10] to change as well.

To do so, write a function file `paramfun` that takes the parameters of the ODE fit and calculates the trajectory over the times `t`.

```
type paramfun
```

```
function pos = paramfun(x,tspan)

sigma = x(1);
beta = x(2);
rho = x(3);
xt0 = x(4:6);
f = @(t,a) [-sigma*a(1) + sigma*a(2); rho*a(1) - a(2) - a(1)*a(3); -beta*a(3) + a(1)*a(2)];
[~,pos] = ode45(f,tspan,xt0);
```

To find the best parameters, use `lsqcurvefit` to minimize the differences between the new calculated ODE trajectory and the circular arc `soln`.

```
xt0 = zeros(1,6);
xt0(1) = sigma;
xt0(2) = beta;
xt0(3) = rho;
xt0(4:6) = soln(1,:);
[pbest,presnorm,presidual,exitflag,output] = lsqcurvefit(@paramfun,xt0,tspan,soln);
```

```
Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.
```

Determine how much this optimization changed the parameters.

```
fprintf('New parameters: %f, %f, %f',pbest(1:3))
```

```
New parameters: 9.132446, 2.854998, 27.937986
```

```
fprintf('Original parameters: %f, %f, %f',[sigma,beta,rho])
```

```
Original parameters: 10.000000, 2.666667, 28.000000
```

The parameters `sigma` and `beta` changed by about 10%.

Plot the modified solution.

```
figure
hold on
odesl = presidual + soln;
plot3(odesl(:,1),odesl(:,2),odesl(:,3),'b')
plot3(soln(:,1),soln(:,2),soln(:,3),'r')
legend('ODE Solution','Circular Arc')
view([-30 -70])
hold off
```
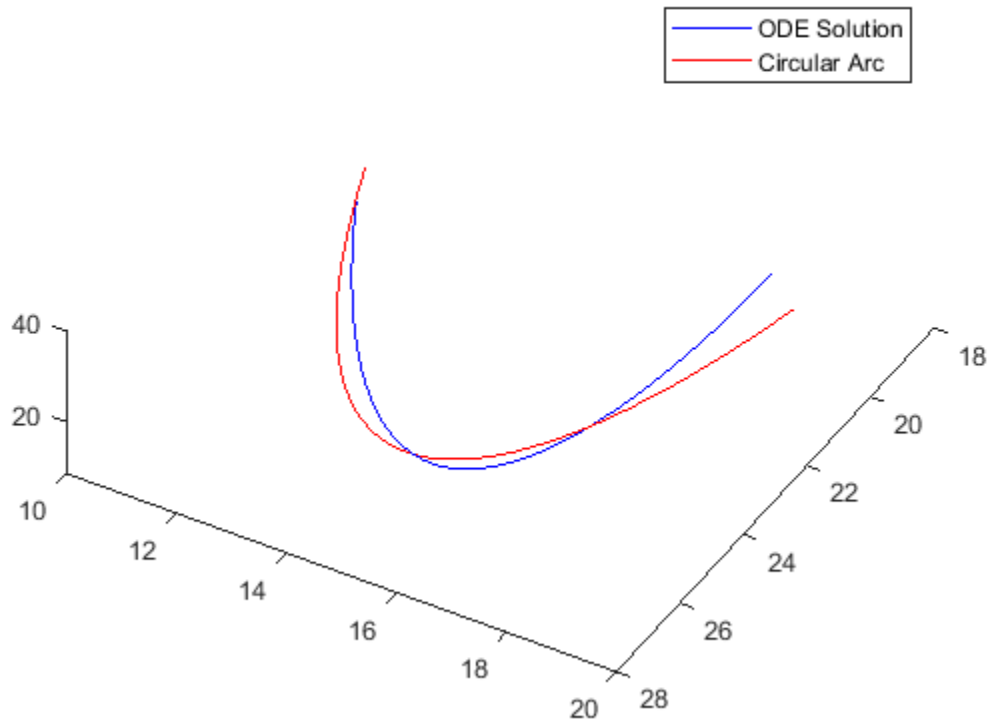
Problems in Fitting ODEs

As described in "Optimizing a Simulation or Ordinary Differential Equation" on page 4-26, an optimizer can have trouble due to the inherent noise in numerical ODE solutions. If you suspect that your solution is not ideal, perhaps because the exit message or exit flag indicates a potential inaccuracy, then try changing the finite differencing. In this example, use a larger finite difference step size and central finite differences.

```
options = optimoptions('lsqcurvefit','FiniteDifferenceStepSize',1e-4,...
    'FiniteDifferenceType','central');
[pbest2,presnorm2,presidual2,exitflag2,output2] = ...
    lsqcurvefit(@paramfun,xt0,tspan,soln,[],[],options);
```

```
Local minimum possible.

lsqcurvefit stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.
```

In this case, using these finite differencing options does not improve the solution.

```
disp([presnorm,presnorm2])
```

```
   20.0637   20.0637
```

See Also

More About

*   "Optimizing a Simulation or Ordinary Differential Equation" on page 4-26

Nonlinear Least-Squares, Problem-Based

This example shows how to perform nonlinear least-squares curve fitting using the "Problem-Based Optimization Workflow" on page 10-2.

Model

The model equation for this problem is

$$y(t) = A_1 \exp(r_1 t) + A_2 \exp(r_2 t),$$

where A_1, A_2, r_1, and r_2 are the unknown parameters, y is the response, and t is time. The problem requires data for times `tdata` and (noisy) response measurements `ydata`. The goal is to find the best A and r, meaning those values that minimize

$$\sum_{t \,\in\, \text{tdata}} (y(t) - \text{ydata})^2 .$$

Sample Data

Typically, you have data for a problem. In this case, generate artificial noisy data for the problem. Use `A = [1,2]` and `r = [-1,-3]` as the underlying values, and use 200 random values from 0 to 3 as the time data. Plot the resulting data points.

```
rng default % For reproducibility
A = [1,2];
r = [-1,-3];
tdata = 3*rand(200,1);
tdata = sort(tdata); % Increasing times for easier plotting
noisedata = 0.05*randn(size(tdata)); % Artificial noise
ydata = A(1)*exp(r(1)*tdata) + A(2)*exp(r(2)*tdata) + noisedata;
plot(tdata,ydata,'r*')
xlabel 't'
ylabel 'Response'
```
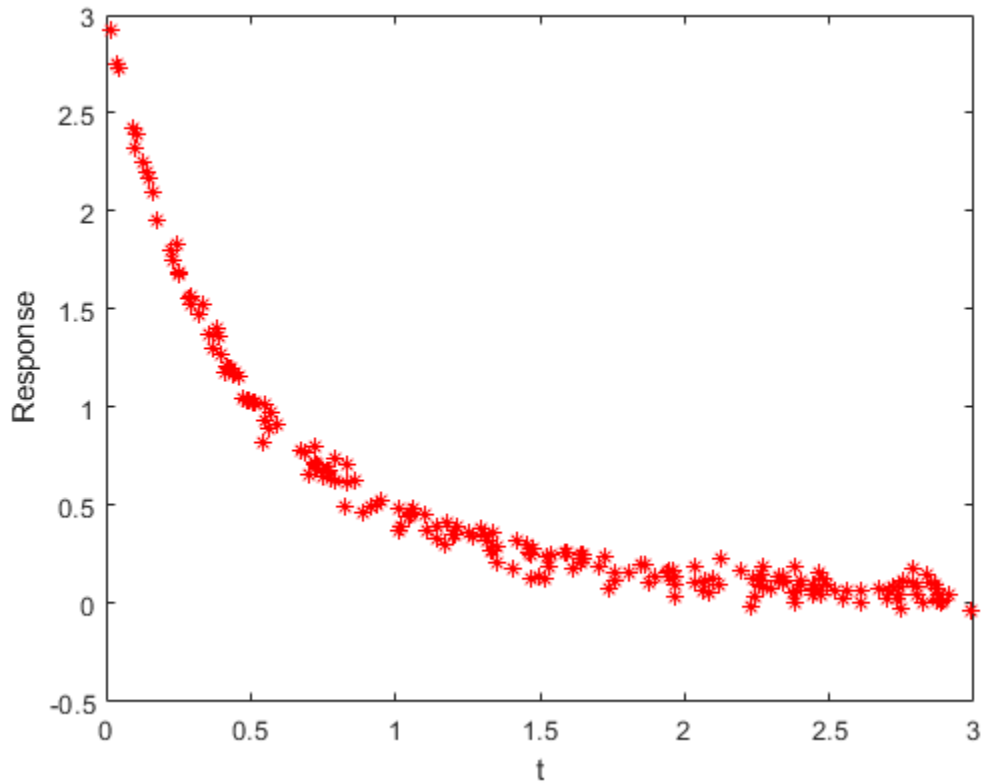
The data are noisy. Therefore, the solution probably will not match the original parameters A and r very well.

Problem-Based Approach

To find the best-fitting parameters A and r, first define optimization variables with those names.

```
A = optimvar('A',2);
r = optimvar('r',2);
```

Create an expression for the objective function, which is the sum of squares to minimize.

```
fun = A(1)*exp(r(1)*tdata) + A(2)*exp(r(2)*tdata);
obj = sum((fun - ydata).^2);
```

Create an optimization problem with the objective function obj.

```
lsqproblem = optimproblem("Objective",obj);
```

For the problem-based approach, specify the initial point as a structure, with the variable names as the fields of the structure. Specify the initial A = [1/2,3/2] and the initial r = [-1/2,-3/2].

```
x0.A = [1/2,3/2];
x0.r = [-1/2,-3/2];
```

Review the problem formulation.

```
show(lsqproblem)
```

```
   OptimizationProblem :

     Solve for:
       A, r

     minimize :
       sum(arg6)

       where:

         arg5 = extraParams{3};
         arg6 = (((A(1) .* exp((r(1) .* extraParams{1}))) + (A(2) .* exp((r(2) .* extraParams{2}

       extraParams
```

Problem-Based Solution

Solve the problem.

```
[sol,fval] = solve(lsqproblem,x0)
```

```
Solving problem using lsqnonlin.
```

```
Local minimum found.
```

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

```
<stopping criteria details>
```

```
sol = struct with fields:
    A: [2×1 double]
    r: [2×1 double]
```

```
fval = 0.4724
```

Plot the resulting solution and the original data.

```
figure
responsedata = evaluate(fun,sol);
plot(tdata,ydata,'r*',tdata,responsedata,'b-')
legend('Original Data','Fitted Curve')
xlabel 't'
ylabel 'Response'
title("Fitted Response")
```

Fitted Response
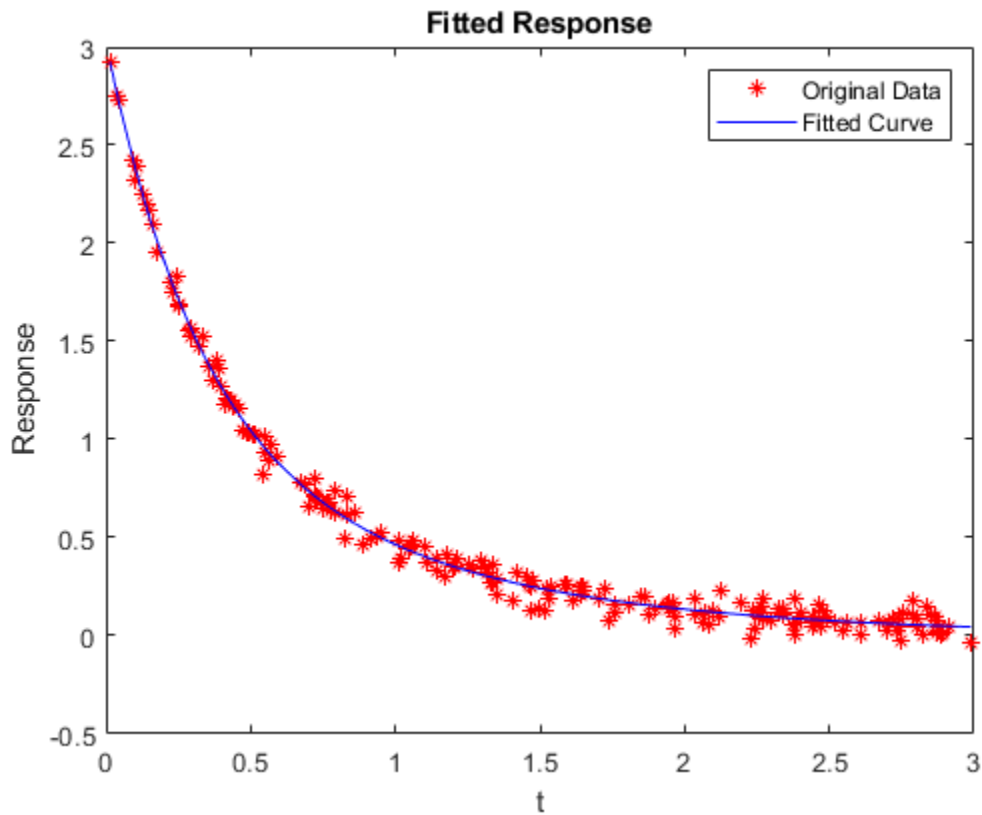
The plot shows that the fitted data matches the original noisy data fairly well.

See how closely the fitted parameters match the original parameters `A = [1,2]` and `r = [-1,-3]`.

```
disp(sol.A)
```

```
    1.1615
    1.8629
```

```
disp(sol.r)
```

```
   -1.0882
   -3.2256
```

The fitted parameters are off by about 15% in `A` and 8% in `r`.

Unsupported Functions Require `fcn2optimexpr`

If your objective function is not composed of elementary functions, you must convert the function to an optimization expression using `fcn2optimexpr`. See "Convert Nonlinear Function to Optimization Expression" on page 7-8. For the present example:

```
fun = @(A,r) A(1)*exp(r(1)*tdata) + A(2)*exp(r(2)*tdata);
response = fcn2optimexpr(fun,A,r);
obj = sum((response - ydata).^2);
```

The remainder of the steps in solving the problem are the same. The only other difference is in the plotting routine, where you call `response` instead of `fun`:

```
responsedata = evaluate(response,sol);
```

For the list of supported functions, see "Supported Operations on Optimization Variables and Expressions" on page 10-43.

See Also
solve

More About

• "Problem-Based Optimization Workflow" on page 10-2

Fit ODE, Problem-Based

This example shows how to find parameters that optimize an ordinary differential equation (ODE) in the least-squares sense, using the problem-based approach.

Problem

The problem is a multistep reaction model involving several substances, some of which react with each other to produce different substances.

For this problem, the true reaction rates are unknown. So, you need to observe the reactions and infer the rates. Assume that you can measure the substances for a set of times t. From these observations, fit the best set of reaction rates to the measurements.

Model

The model has six substances, C_1 through C_6, that react as follows:

- One C_1 and one C_2 react to form one C_3 at rate r_1
- One C_3 and one C_4 react to form one C_5 at rate r_2
- One C_3 and one C_4 react to form one C_6 at rate r_3
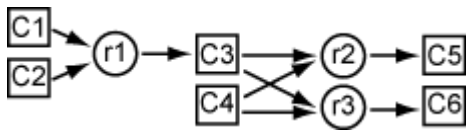
The reaction rate is proportional to the product of the quantities of the required substances. So, if y_i represents the quantity of substance C_i, then the reaction rate to produce C_3 is $r_1 y_1 y_2$. Similarly, the reaction rate to produce C_5 is $r_2 y_3 y_4$, and the reaction rate to produce C_6 is $r_3 y_3 y_4$.

In other words, the differential equation controlling the evolution of the system is

$$\frac{dy}{dt} = \begin{bmatrix} -r_1 y_1 y_2 \\ -r_1 y_1 y_2 \\ -r_2 y_3 y_4 + r_1 y_1 y_2 - r_3 y_3 y_4 \\ -r_2 y_3 y_4 - r_3 y_3 y_4 \\ r_2 y_3 y_4 \\ r_3 y_3 y_4 \end{bmatrix}.$$

Start the differential equation at time 0 at the point $y(0) = [1, 1, 0, 1, 0, 0]$. These initial values ensure that all of the substances react completely, causing C_1 through C_4 to approach zero as time increases.

Express Model in MATLAB

The `diffun` function implements the differential equations in a form ready for solution by `ode45`.

```
type diffun
```

```
function dydt = diffun(~,y,r)
dydt = zeros(6,1);
```

```
s12 = y(1)*y(2);
s34 = y(3)*y(4);

dydt(1) = -r(1)*s12;
dydt(2) = -r(1)*s12;
dydt(3) = -r(2)*s34 + r(1)*s12 - r(3)*s34;
dydt(4) = -r(2)*s34 - r(3)*s34;
dydt(5) = r(2)*s34;
dydt(6) = r(3)*s34;
end
```

The true reaction rates are $r_1 = 2.5$, $r_2 = 1.2$, and $r_3 = 0.45$. Compute the evolution of the system for times zero through five by calling ode45.

```
rtrue = [2.5 1.2 0.45];
y0 = [1 1 0 1 0 0];
tspan = linspace(0,5);
soltrue = ode45(@(t,y)diffun(t,y,rtrue),tspan,y0);
yvalstrue = deval(soltrue,tspan);
for i = 1:6
    subplot(3,2,i)
    plot(tspan,yvalstrue(i,:))
    title(['y(',num2str(i),')'])
end
```
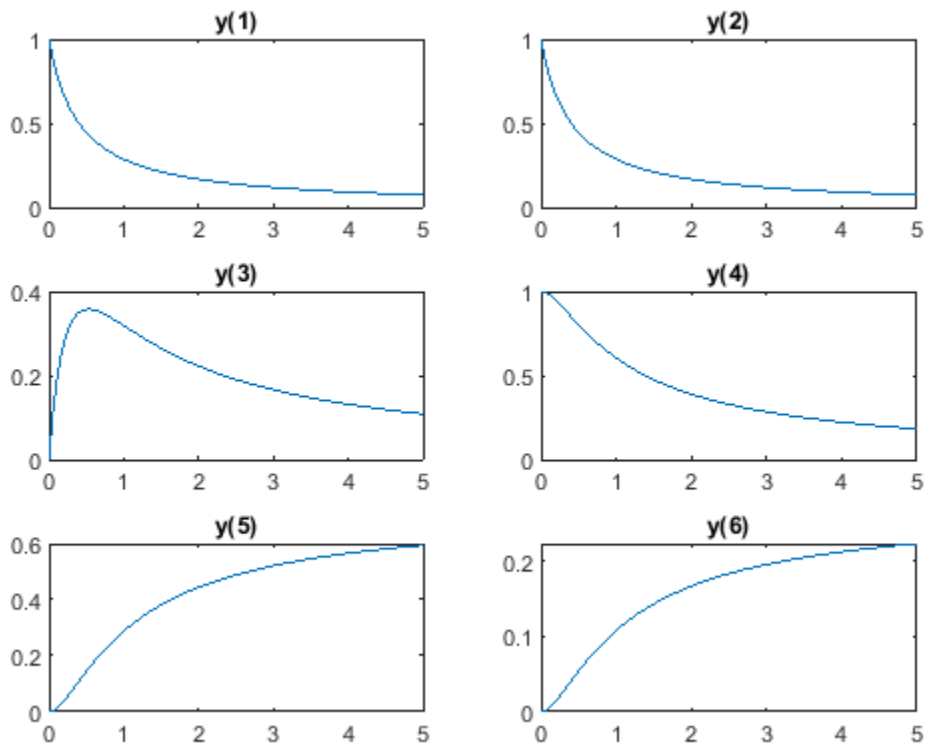
Optimization Problem

To prepare the problem for solution in the problem-based approach, create a three-element optimization variable r that has a lower bound of 0.1 and an upper bound of 10.

```
r = optimvar('r',3,"LowerBound",0.1,"UpperBound",10);
```

The objective function for this problem is the sum of squares of the differences between the ODE solution with parameters r and the solution with the true parameters yvals. To express this objective function, first write a MATLAB function that computes the ODE solution using parameters r. This function is the RtoODE function.

```
type RtoODE
```

```
function solpts = RtoODE(r,tspan,y0)
sol = ode45(@(t,y)diffun(t,y,r),tspan,y0);
solpts = deval(sol,tspan);
end
```

To use RtoODE in an objective function, convert the function to an optimization expression by using fcn2optimexpr. See "Convert Nonlinear Function to Optimization Expression" on page 7-8.

```
myfcn = fcn2optimexpr(@RtoODE,r,tspan,y0);
```

Express the objective function as the sum of squared differences between the ODE solution and the solution with true parameters.

```
obj = sum(sum((myfcn - yvalstrue).^2));
```

Create an optimization problem with the objective function obj.

```
prob = optimproblem("Objective",obj);
```

View the problem by calling show.

```
show(prob)
```

```
  OptimizationProblem :

    Solve for:
       r

    minimize :
       sum(sum((RtoODE(r, extraParams{1}, extraParams{2})
     - extraParams{3}).^2, 1))

        extraParams{1}:

       Columns 1 through 7

             0    0.0505    0.1010    0.1515    0.2020    0.2525    0.3030

       Columns 8 through 14

        0.3535    0.4040    0.4545    0.5051    0.5556    0.6061    0.6566

       Columns 15 through 21
```

```
   0.7071    0.7576    0.8081    0.8586    0.9091    0.9596    1.0101

Columns 22 through 28

   1.0606    1.1111    1.1616    1.2121    1.2626    1.3131    1.3636

Columns 29 through 35

   1.4141    1.4646    1.5152    1.5657    1.6162    1.6667    1.7172

Columns 36 through 42

   1.7677    1.8182    1.8687    1.9192    1.9697    2.0202    2.0707

Columns 43 through 49

   2.1212    2.1717    2.2222    2.2727    2.3232    2.3737    2.4242

Columns 50 through 56

   2.4747    2.5253    2.5758    2.6263    2.6768    2.7273    2.7778

Columns 57 through 63

   2.8283    2.8788    2.9293    2.9798    3.0303    3.0808    3.1313

Columns 64 through 70

   3.1818    3.2323    3.2828    3.3333    3.3838    3.4343    3.4848

Columns 71 through 77

   3.5354    3.5859    3.6364    3.6869    3.7374    3.7879    3.8384

Columns 78 through 84

   3.8889    3.9394    3.9899    4.0404    4.0909    4.1414    4.1919

Columns 85 through 91

   4.2424    4.2929    4.3434    4.3939    4.4444    4.4949    4.5455

Columns 92 through 98

   4.5960    4.6465    4.6970    4.7475    4.7980    4.8485    4.8990

Columns 99 through 100

   4.9495    5.0000

extraParams{2}:

   1    1    0    1    0    0

extraParams{3}:

Columns 1 through 7
```

```
    1.0000    0.8879    0.7984    0.7253    0.6644    0.6130    0.5690
    1.0000    0.8879    0.7984    0.7253    0.6644    0.6130    0.5690
         0    0.1074    0.1847    0.2404    0.2805    0.3089    0.3287
    1.0000    0.9953    0.9831    0.9657    0.9449    0.9219    0.8977
         0    0.0034    0.0123    0.0249    0.0401    0.0568    0.0744
         0    0.0013    0.0046    0.0094    0.0150    0.0213    0.0279

Columns 8 through 14

    0.5308    0.4975    0.4681    0.4420    0.4186    0.3976    0.3786
    0.5308    0.4975    0.4681    0.4420    0.4186    0.3976    0.3786
    0.3421    0.3506    0.3554    0.3574    0.3573    0.3556    0.3527
    0.8729    0.8481    0.8235    0.7994    0.7759    0.7532    0.7313
    0.0924    0.1105    0.1284    0.1459    0.1630    0.1795    0.1954
    0.0347    0.0414    0.0481    0.0547    0.0611    0.0673    0.0733

Columns 15 through 21

    0.3613    0.3456    0.3311    0.3178    0.3056    0.2942    0.2837
    0.3613    0.3456    0.3311    0.3178    0.3056    0.2942    0.2837
    0.3489    0.3444    0.3395    0.3342    0.3287    0.3230    0.3173
    0.7102    0.6900    0.6706    0.6520    0.6343    0.6173    0.6010
    0.2108    0.2255    0.2396    0.2531    0.2660    0.2783    0.2902
    0.0790    0.0846    0.0898    0.0949    0.0997    0.1044    0.1088

Columns 22 through 28

    0.2739    0.2647    0.2562    0.2481    0.2406    0.2335    0.2268
    0.2739    0.2647    0.2562    0.2481    0.2406    0.2335    0.2268
    0.3116    0.3059    0.3002    0.2946    0.2891    0.2837    0.2784
    0.5855    0.5706    0.5564    0.5428    0.5297    0.5172    0.5052
    0.3015    0.3123    0.3226    0.3325    0.3420    0.3511    0.3598
    0.1131    0.1171    0.1210    0.1247    0.1283    0.1317    0.1349

Columns 29 through 35

    0.2205    0.2146    0.2089    0.2035    0.1984    0.1936    0.1890
    0.2205    0.2146    0.2089    0.2035    0.1984    0.1936    0.1890
    0.2732    0.2682    0.2633    0.2585    0.2538    0.2493    0.2449
    0.4938    0.4827    0.4722    0.4620    0.4523    0.4429    0.4339
    0.3682    0.3762    0.3839    0.3913    0.3984    0.4052    0.4117
    0.1381    0.1411    0.1440    0.1467    0.1494    0.1519    0.1544

Columns 36 through 42

    0.1846    0.1804    0.1763    0.1725    0.1688    0.1653    0.1619
    0.1846    0.1804    0.1763    0.1725    0.1688    0.1653    0.1619
    0.2406    0.2364    0.2324    0.2285    0.2246    0.2209    0.2173
    0.4252    0.4168    0.4087    0.4010    0.3935    0.3862    0.3792
    0.4181    0.4241    0.4300    0.4357    0.4411    0.4464    0.4515
    0.1568    0.1591    0.1613    0.1634    0.1654    0.1674    0.1693

Columns 43 through 49

    0.1587    0.1556    0.1526    0.1497    0.1469    0.1442    0.1416
    0.1587    0.1556    0.1526    0.1497    0.1469    0.1442    0.1416
    0.2138    0.2104    0.2071    0.2039    0.2007    0.1977    0.1947
    0.3725    0.3660    0.3596    0.3535    0.3476    0.3419    0.3364
```

```
    0.4564    0.4611    0.4657    0.4702    0.4744    0.4786    0.4826
    0.1711    0.1729    0.1746    0.1763    0.1779    0.1795    0.1810

Columns 50 through 56

    0.1392    0.1368    0.1344    0.1322    0.1300    0.1279    0.1259
    0.1392    0.1368    0.1344    0.1322    0.1300    0.1279    0.1259
    0.1918    0.1890    0.1863    0.1836    0.1810    0.1785    0.1761
    0.3310    0.3258    0.3207    0.3158    0.3111    0.3064    0.3019
    0.4866    0.4903    0.4940    0.4976    0.5010    0.5044    0.5077
    0.1825    0.1839    0.1853    0.1866    0.1879    0.1892    0.1904

Columns 57 through 63

    0.1239    0.1220    0.1202    0.1184    0.1166    0.1149    0.1133
    0.1239    0.1220    0.1202    0.1184    0.1166    0.1149    0.1133
    0.1737    0.1713    0.1690    0.1668    0.1646    0.1625    0.1605
    0.2976    0.2933    0.2892    0.2852    0.2813    0.2775    0.2737
    0.5109    0.5139    0.5169    0.5199    0.5227    0.5255    0.5282
    0.1916    0.1927    0.1939    0.1950    0.1960    0.1971    0.1981

Columns 64 through 70

    0.1117    0.1101    0.1086    0.1072    0.1057    0.1043    0.1030
    0.1117    0.1101    0.1086    0.1072    0.1057    0.1043    0.1030
    0.1584    0.1565    0.1546    0.1527    0.1508    0.1491    0.1473
    0.2701    0.2666    0.2632    0.2598    0.2566    0.2534    0.2503
    0.5308    0.5334    0.5359    0.5383    0.5407    0.5430    0.5453
    0.1991    0.2000    0.2010    0.2019    0.2028    0.2036    0.2045

Columns 71 through 77

    0.1017    0.1004    0.0991    0.0979    0.0967    0.0955    0.0944
    0.1017    0.1004    0.0991    0.0979    0.0967    0.0955    0.0944
    0.1456    0.1439    0.1423    0.1407    0.1391    0.1376    0.1361
    0.2472    0.2443    0.2414    0.2385    0.2358    0.2331    0.2304
    0.5475    0.5496    0.5517    0.5538    0.5558    0.5578    0.5597
    0.2053    0.2061    0.2069    0.2077    0.2084    0.2092    0.2099

Columns 78 through 84

    0.0933    0.0922    0.0911    0.0901    0.0891    0.0881    0.0871
    0.0933    0.0922    0.0911    0.0901    0.0891    0.0881    0.0871
    0.1346    0.1331    0.1317    0.1303    0.1290    0.1277    0.1264
    0.2279    0.2253    0.2229    0.2204    0.2181    0.2157    0.2135
    0.5616    0.5634    0.5652    0.5670    0.5687    0.5704    0.5720
    0.2106    0.2113    0.2119    0.2126    0.2133    0.2139    0.2145

Columns 85 through 91

    0.0862    0.0852    0.0843    0.0834    0.0826    0.0817    0.0809
    0.0862    0.0852    0.0843    0.0834    0.0826    0.0817    0.0809
    0.1251    0.1238    0.1226    0.1214    0.1202    0.1191    0.1179
    0.2112    0.2091    0.2069    0.2048    0.2028    0.2008    0.1988
    0.5736    0.5752    0.5768    0.5783    0.5798    0.5813    0.5827
    0.2151    0.2157    0.2163    0.2169    0.2174    0.2180    0.2185

Columns 92 through 98
```

```
        0.0801    0.0793    0.0785    0.0777    0.0770    0.0762    0.0755
        0.0801    0.0793    0.0785    0.0777    0.0770    0.0762    0.0755
        0.1168    0.1157    0.1146    0.1136    0.1125    0.1115    0.1105
        0.1969    0.1950    0.1931    0.1913    0.1895    0.1877    0.1860
        0.5841    0.5855    0.5868    0.5882    0.5895    0.5907    0.5920
        0.2190    0.2196    0.2201    0.2206    0.2210    0.2215    0.2220

    Columns 99 through 100

        0.0748    0.0741
        0.0748    0.0741
        0.1095    0.1086
        0.1843    0.1826
        0.5932    0.5944
        0.2225    0.2229



    variable bounds:
        0.1 <= r(1) <= 10
        0.1 <= r(2) <= 10
        0.1 <= r(3) <= 10
```

Solve Problem

To find the best-fitting parameters r, give an initial guess r0 for the solver and call `solve`.

```
r0.r = [1 1 1];
[rsol,sumsq] = solve(prob,r0)

Solving problem using lsqnonlin.

Local minimum found.

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

rsol = struct with fields:
    r: [3x1 double]


sumsq = 3.8660e-15
```

The sum of squared differences is essentially zero, meaning the solver found parameters that cause the ODE solution to match the solution with true parameters. So, as expected, the solution contains the true parameters.

```
disp(rsol.r)

    2.5000
    1.2000
    0.4500

disp(rtrue)

    2.5000    1.2000    0.4500
```

Limited Observations

Suppose that you cannot observe all the components of y, but only the final outputs y(5) and y(6). Can you obtain the values of all the reaction rates based on this limited information?

To find out, modify the function RtoODE to return only the fifth and sixth ODE outputs. The modified ODE solver is in RtoODE2.

type RtoODE2

```
function solpts = RtoODE2(r,tspan,y0)
solpts = RtoODE(r,tspan,y0);
solpts = solpts([5,6],:); % Just y(5) and y(6)
end
```

The RtoODE2 function simply calls RtoODE and then takes the final two rows of the output.

Create a new optimization expression from RtoODE2 and the optimization variable r, the time span data tspan, and the initial point y0.

myfcn2 = fcn2optimexpr(@RtoODE2,r,tspan,y0);

Modify the comparison data to include outputs 5 and 6 only.

yvals2 = yvalstrue([5,6],:);

Create a new objective and new optimization problem from the optimization expression myfcn2 and the comparison data yvals2.

```
obj2 = sum(sum((myfcn2 - yvals2).^2));
prob2 = optimproblem("Objective",obj2);
```

Solve the problem based on this limited set of observations.

[rsol2,sumsq2] = solve(prob2,r0)

Solving problem using lsqnonlin.

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to its initial value is less than the value of the function tolerance.

rsol2 = struct with fields:
    r: [3x1 double]

sumsq2 = 2.1616e-05

Once again, the returned sum of squares is essentially zero. Does this mean that the solver found the correct reaction rates?

disp(rsol2.r)

    1.7811
    1.5730
    0.5899

disp(rtrue)

```
    2.5000    1.2000    0.4500
```

No; in this case, the new rates are quite different from the true rates. However, a plot of the new ODE solution compared to the true values shows that y(5) and y(6) match the true values.

```
figure
plot(tspan,yvals2(1,:),'b-')
hold on
ss2 = RtoODE2(rsol2.r,tspan,y0);
plot(tspan,ss2(1,:),'r--')
plot(tspan,yvals2(2,:),'c-')
plot(tspan,ss2(2,:),'m--')
legend('True y(5)','New y(5)','True y(6)','New y(6)','Location','northwest')
hold off
```
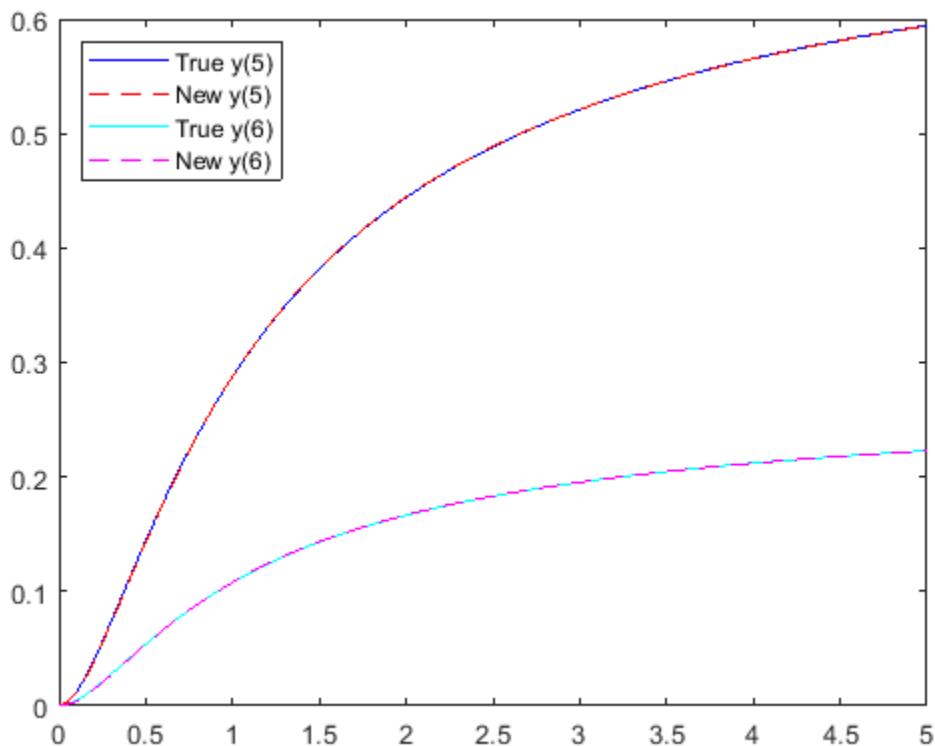

To identify the correct reaction rates for this problem, you must have data for more observations than y(5) and y(6).

Plot all the components of the solution with the new parameters, and plot the solution with the true parameters.

```
figure
yvals2 = RtoODE(rsol2.r,tspan,y0);
for i = 1:6
    subplot(3,2,i)
    plot(tspan,yvalstrue(i,:),'b-',tspan,yvals2(i,:),'r--')
    legend('True','New','Location','best')
```

```
        title(['y(',num2str(i),')'])
end
```
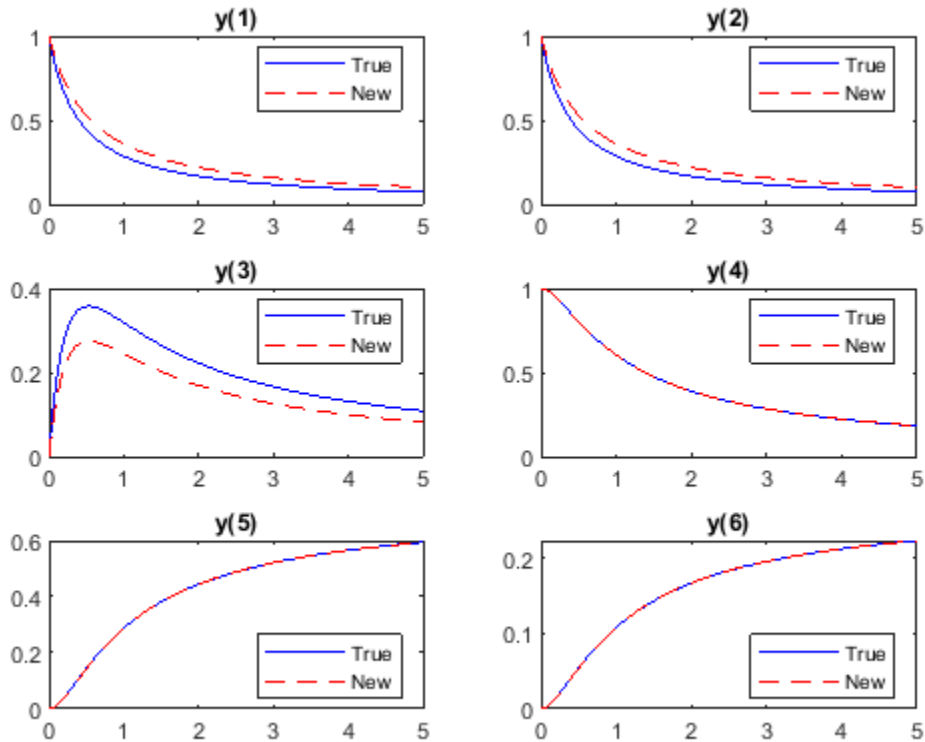

With the new parameters, substances C_1 and C_2 drain more slowly, and substance C_3 does not accumulate as much. But substances C_4, C_5, and C_6 have exactly the same evolution with both the new parameters and the true parameters.

See Also

`fcn2optimexpr` | `ode45` | `solve`

More About

- "Problem-Based Optimization Workflow" on page 10-2

Nonlinear Data-Fitting Using Several Problem-Based Approaches

The general advice for least-squares problem setup is to formulate the problem in a way that allows `solve` to recognize that the problem has a least-squares form. When you do that, `solve` internally calls `lsqnonlin`, which is efficient at solving least-squares problems. See "Write Objective Function for Problem-Based Least Squares" on page 12-85.

This example shows the efficiency of a least-squares solver by comparing the performance of `lsqnonlin` with that of `fminunc` on the same problem. Additionally, the example shows added benefits that you can obtain by explicitly recognizing and handling separately the linear parts of a problem.

Problem Setup

Consider the following data:

```
Data = ...
  [0.0000    5.8955
   0.1000    3.5639
   0.2000    2.5173
   0.3000    1.9790
   0.4000    1.8990
   0.5000    1.3938
   0.6000    1.1359
   0.7000    1.0096
   0.8000    1.0343
   0.9000    0.8435
   1.0000    0.6856
   1.1000    0.6100
   1.2000    0.5392
   1.3000    0.3946
   1.4000    0.3903
   1.5000    0.5474
   1.6000    0.3459
   1.7000    0.1370
   1.8000    0.2211
   1.9000    0.1704
   2.0000    0.2636];
```

Plot the data points.

```
t = Data(:,1);
y = Data(:,2);
plot(t,y,'ro')
title('Data points')
```
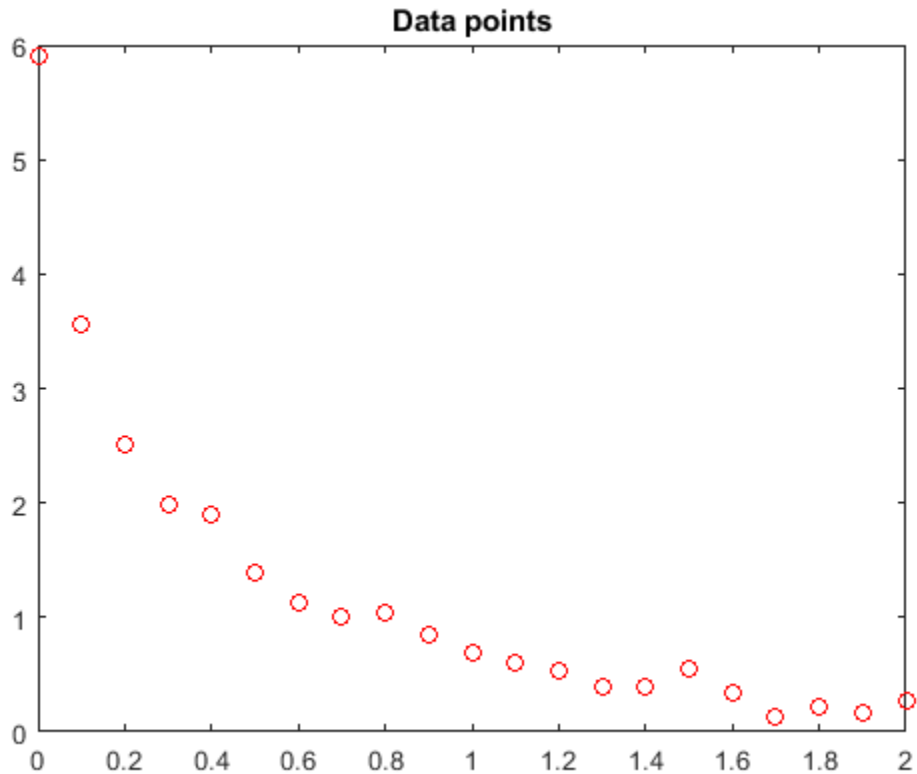
The problem is to fit the function

```
y = c(1)*exp(-lam(1)*t) + c(2)*exp(-lam(2)*t)
```

to the data.

Solution Approach Using Default Solver

To begin, define optimization variables corresponding to the equation.

```
c = optimvar('c',2);
lam = optimvar('lam',2);
```

Arbitrarily set the initial point x0 as follows: c(1) = 1, c(2) = 1, lam(1) = 1, and lam(2) = 0:

```
x0.c = [1,1];
x0.lam = [1,0];
```

Create a function that computes the value of the response at times t when the parameters are c and lam.

```
diffun = c(1)*exp(-lam(1)*t) + c(2)*exp(-lam(2)*t);
```

Convert `diffun` to an optimization expression that sums the squares of the differences between the function and the data y.

```
diffexpr = sum((diffun - y).^2);
```

Create an optimization problem having `diffexpr` as the objective function.

```
ssqprob = optimproblem('Objective',diffexpr);
```

Solve the problem using the default solver.

```
[sol,fval,exitflag,output] = solve(ssqprob,x0)
```

```
Solving problem using lsqnonlin.

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

sol = struct with fields:
      c: [2x1 double]
    lam: [2x1 double]


fval = 0.1477

exitflag =
    FunctionChangeBelowTolerance


output = struct with fields:
    firstorderopt: 7.8874e-06
       iterations: 6
        funcCount: 35
     cgiterations: 0
        algorithm: 'trust-region-reflective'
         stepsize: 0.0096
          message: '...'
           solver: 'lsqnonlin'
```

Plot the resulting curve based on the returned solution values `sol.c` and `sol.lam`.

```
resp = evaluate(diffun,sol);
hold on
plot(t,resp)
hold off
```

Data points
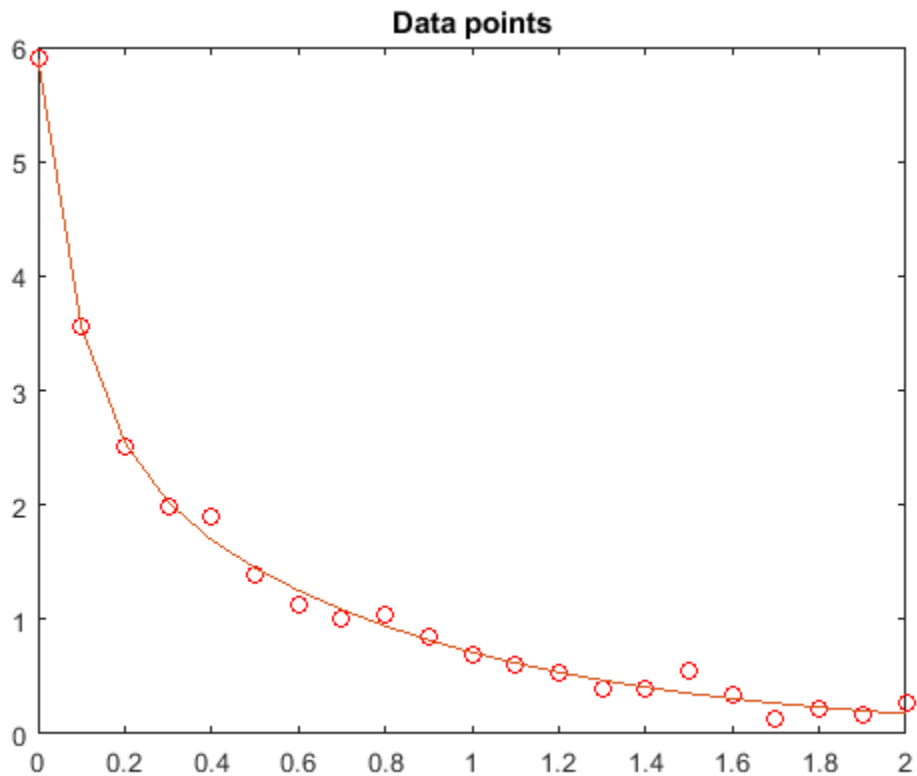
The fit looks to be as good as possible.

Solution Approach Using `fminunc`

To solve the problem using the `fminunc` solver, set the `'Solver'` option to `'fminunc'` when calling `solve`.

```
[xunc,fvalunc,exitflagunc,outputunc] = solve(ssqprob,x0,'Solver',"fminunc")
```

```
Solving problem using fminunc.

Local minimum found.

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

xunc = struct with fields:
      c: [2x1 double]
    lam: [2x1 double]


fvalunc = 0.1477

exitflagunc =
    OptimalSolution


outputunc = struct with fields:
      iterations: 30
```

```
      funcCount: 185
       stepsize: 0.0017
   lssteplength: 1
   firstorderopt: 2.9562e-05
      algorithm: 'quasi-newton'
        message: '...'
         solver: "fminunc"
```

Notice that `fminunc` found the same solution as `lsqcurvefit`, but took many more function evaluations to do so. The parameters for `fminunc` are in the opposite order as those for `lsqcurvefit`; the larger `lam` is `lam(2)`, not `lam(1)`. This is not surprising, the order of variables is arbitrary.

```
fprintf(['There were %d iterations using fminunc,' ...
    ' and %d using lsqcurvefit.\n'], ...
    outputunc.iterations,output.iterations)
```

There were 30 iterations using fminunc, and 6 using lsqcurvefit.

```
fprintf(['There were %d function evaluations using fminunc,' ...
    ' and %d using lsqcurvefit.'], ...
    outputunc.funcCount,output.funcCount)
```

There were 185 function evaluations using fminunc, and 35 using lsqcurvefit.

Splitting the Linear and Nonlinear Problems

Notice that the fitting problem is linear in the parameters `c(1)` and `c(2)`. This means for any values of `lam(1)` and `lam(2)`, you can use the backslash operator to find the values of `c(1)` and `c(2)` that solve the least-squares problem.

Rework the problem as a two-dimensional problem, searching for the best values of `lam(1)` and `lam(2)`. The values of `c(1)` and `c(2)` are calculated at each step using the backslash operator as described above. To do so, use the `fitvector` function, which performs the backslash operation to obtain `c(1)` and `c(2)` at each solver iteration.

```
type fitvector
```

```
function yEst = fitvector(lam,xdata,ydata)
%FITVECTOR  Used by DATDEMO to return value of fitting function.
%   yEst = FITVECTOR(lam,xdata) returns the value of the fitting function, y
%   (defined below), at the data points xdata with parameters set to lam.
%   yEst is returned as a N-by-1 column vector, where N is the number of
%   data points.
%
%   FITVECTOR assumes the fitting function, y, takes the form
%
%     y =  c(1)*exp(-lam(1)*t) + ... + c(n)*exp(-lam(n)*t)
%
%   with n linear parameters c, and n nonlinear parameters lam.
%
%   To solve for the linear parameters c, we build a matrix A
%   where the j-th column of A is exp(-lam(j)*xdata) (xdata is a vector).
%   Then we solve A*c = ydata for the linear least-squares solution c,
%   where ydata is the observed values of y.

A = zeros(length(xdata),length(lam));  % build A matrix
```

```
for j = 1:length(lam)
    A(:,j) = exp(-lam(j)*xdata);
end
c = A\ydata; % solve A*c = y for linear parameters c
yEst = A*c; % return the estimated response based on c
```

Solve the problem using `solve` starting from a two-dimensional initial point `x02.lam = [1,0]`. To do so, first convert the `fitvector` function to an optimization expression using `fcn2optimexpr`. See "Convert Nonlinear Function to Optimization Expression" on page 7-8. To avoid a warning, give the output size of the resulting expression. Create a new optimization problem with objective as the sum of squared differences between the converted `fitvector` function and the data `y`.

```
x02.lam = x0.lam;
F2 = fcn2optimexpr(@(x) fitvector(x,t,y),lam,'OutputSize',[length(t),1]);
ssqprob2 = optimproblem('Objective',sum((F2 - y).^2));
[sol2,fval2,exitflag2,output2] = solve(ssqprob2,x02)
```

```
Solving problem using lsqnonlin.

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

sol2 = struct with fields:
    lam: [2x1 double]


fval2 = 0.1477

exitflag2 =
    FunctionChangeBelowTolerance


output2 = struct with fields:
    firstorderopt: 4.4018e-06
        iterations: 10
         funcCount: 33
      cgiterations: 0
         algorithm: 'trust-region-reflective'
          stepsize: 0.0080
           message: '...'
            solver: 'lsqnonlin'
```

The efficiency of the two-dimensional solution is similar to that of the four-dimensional solution:

```
fprintf(['There were %d function evaluations using the 2-d ' ...
    'formulation, and %d using the 4-d formulation.'], ...
    output2.funcCount,output.funcCount)
```

```
There were 33 function evaluations using the 2-d formulation, and 35 using the 4-d formulation.
```

Split Problem is More Robust to Initial Guess

Choosing a bad starting point for the original four-parameter problem leads to a local solution that is not global. Choosing a starting point with the same bad `lam(1)` and `lam(2)` values for the split two-parameter problem leads to the global solution. To show this, rerun the original problem with a start

point that leads to a relatively bad local solution, and compare the resulting fit with the global solution.

```
x0bad.c = [5 1];
x0bad.lam = [1 0];
[solbad,fvalbad,exitflagbad,outputbad] = solve(ssqprob,x0bad)

Solving problem using lsqnonlin.

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the value of the function tolerance.

solbad = struct with fields:
      c: [2x1 double]
    lam: [2x1 double]


fvalbad = 2.2173

exitflagbad =
    FunctionChangeBelowTolerance


outputbad = struct with fields:
    firstorderopt: 0.0057
       iterations: 32
        funcCount: 165
      cgiterations: 0
        algorithm: 'trust-region-reflective'
         stepsize: 0.0021
          message: '...'
           solver: 'lsqnonlin'


respbad = evaluate(diffun,solbad);
hold on
plot(t,respbad,'g')
legend('Data','Global fit','Bad local fit','Location','NE')
hold off
```
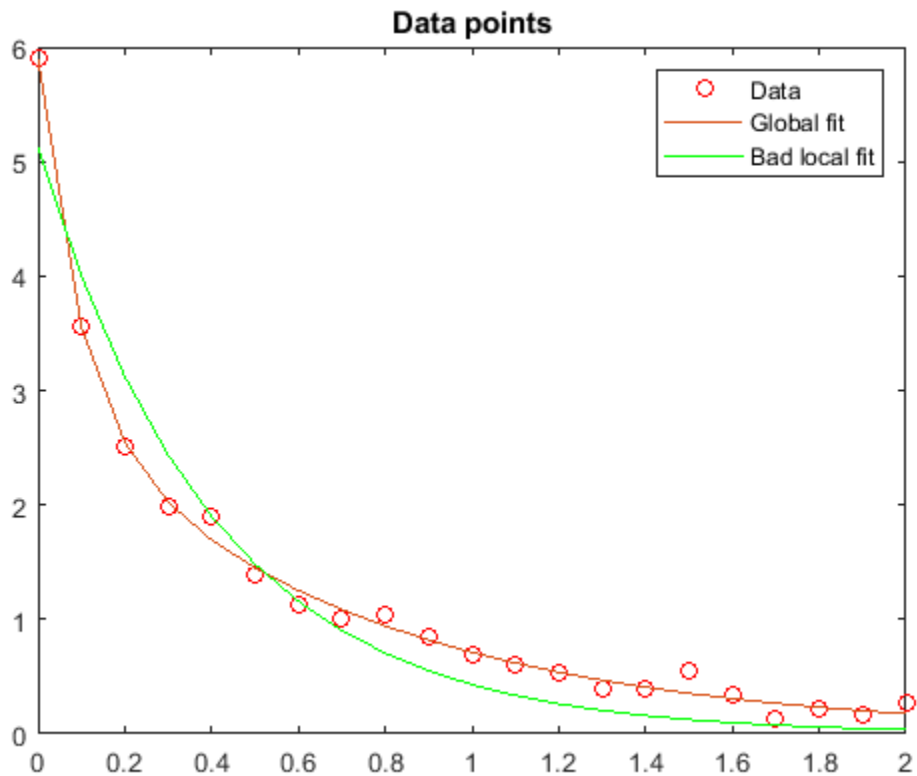
Data points


```
fprintf(['The residual norm at the good ending point is %f,' ...
    ' and the residual norm at the bad ending point is %f.'], ...
    fval,fvalbad)
```

The residual norm at the good ending point is 0.147723, and the residual norm at the bad ending p

See Also
`fcn2optimexpr | solve`

More About
• "Nonlinear Data-Fitting" on page 12-10
• "Problem-Based Optimization Workflow" on page 10-2

Write Objective Function for Problem-Based Least Squares

To specify an objective function for problem-based least squares, write the objective explicitly as a sum of squares. By explicitly using a least-squares formulation, you obtain the most appropriate and efficient solver for your problem. For example,

```matlab
t = randn(10,1); % Data for the example
x = optimvar('x',10);

obj = sum((x - t).^2); % Explicit sum of squares

prob = optimproblem("Objective",obj);
% Check to see the default solver
opts = optimoptions(prob)

opts =

  lsqlin options:
...
```

In contrast, expressing the objective as a mathematically equivalent expression gives a problem that the software interprets as a general quadratic problem.

```matlab
obj2 = (x - t)'*(x - t); % Equivalent to a sum of squares,
                         % but not interpreted as a sum of squares
prob2 = optimproblem("Objective",obj2);
% Check to see the default solver
opts = optimoptions(prob2)

opts =

  quadprog options:
...
```

Similarly, write nonlinear least-squares as explicit sums of squares of optimization expressions.

```matlab
t = linspace(0,5); % Data for the example
A = optimvar('A');
r = optimvar('r');
expr = fcn2optimexpr(@(A,r)A*exp(r*t),A,r);
ydata = 3*exp(-2*t) + 0.1*randn(size(t));

obj3 = sum((expr - ydata).^2); % Explicit sum of squares

prob3 = optimproblem("Objective",obj3);
% Check to see the default solver
opts = optimoptions(prob3)

opts =

  lsqnonlin options:
...
```

The most general form that the software interprets as a least-squares problem is a sum of expressions R_n of this form:

$$R_n = a_n + k_1 \sum \left(k_2 \sum \left(k_3 \sum \left(...k_j e_n^2 \right) \right) \right)$$

- e_n is any expression. If multidimensional, e_n should be squared term-by-term using `.^2`.
- a_n is a scalar numeric value.
- The k_j are positive scalar numeric values.

Each expression R_n must evaluate to a scalar, not a multidimensional value. For example,

```
x = optimvar('x',10,3,4);
y = optimvar('y',10,2);
t = randn(10,3,4); % Data for example
u = randn(10,2); % Data for example
a = randn; % Coefficient
k = abs(randn(5,1)); % Positive coefficients
% Explicit sums of squares:
R1 = a + k(1)*sum(k(2)*sum(k(3)*sum((x - t).^2,3)));
R2 = k(4)*sum(k(5)*sum((y - u).^2,2));
R3 = 1 + (fcn2optimexpr(@cos,x(1)))^2;
prob = optimproblem('Objective',R1 + R2 + R3);
options = optimoptions(prob)

options =

  lsqnonlin options:
...
```

See Also

More About

- "Problem-Based Optimization Workflow" on page 10-2

Code Generation in Nonlinear Least Squares: Background

What Is Code Generation?

Code generation is the conversion of MATLAB code to C code using MATLAB Coder. Code generation requires a MATLAB Coder license.

Typically, you use code generation to deploy code on hardware that is not running MATLAB. For example, you can deploy code on a robot, using `lsqnonlin` for optimizing movement or planning.

For an example, see "Generate Code for lsqcurvefit or lsqnonlin" on page 12-89. For code generation in other optimization solvers, see "Generate Code for fmincon" on page 6-127, "Generate Code for quadprog" on page 11-60, or "Generate Code for fsolve" on page 13-38.

Requirements for Code Generation

- `lsqcurvefit` and `lsqnonlin` support code generation using either the `codegen` function or the MATLAB Coder app. You must have a MATLAB Coder license to generate code.
- The target hardware must support standard double-precision floating-point computations. You cannot generate code for single-precision or fixed-point computations.
- Code generation targets do not use the same math kernel libraries as MATLAB solvers. Therefore, code generation solutions can vary from solver solutions, especially for poorly conditioned problems.
- All code for generation must be MATLAB code. In particular, you cannot use a custom black-box function as an objective function for `lsqcurvefit` or `lsqnonlin`. You can use `coder.ceval` to evaluate a custom function coded in C or C++. However, the custom function must be called in a MATLAB function.
- `lsqcurvefit` and `lsqnonlin` do not support the `problem` argument for code generation.

  ```
  [x,fval] = lsqnonlin(problem) % Not supported
  ```
- You must specify the objective function by using function handles, not strings or character names.

  ```
  x = lsqnonlin(@fun,x0,lb,ub,options) % Supported
  % Not supported: lsqnonlin('fun',...) or lsqnonlin("fun",...)
  ```
- All input matrices `lb` and `ub` must be full, not sparse. You can convert sparse matrices to full by using the `full` function.
- The `lb` and `ub` arguments must have the same number of entries as the `x0` argument or must be empty `[]`.
- For advanced code optimization involving embedded processors, you also need an Embedded Coder license.
- You must include options for `lsqcurvefit` or `lsqnonlin` and specify them using `optimoptions`. The options must include the `Algorithm` option, set to `'levenberg-marquardt'`.

  ```
  options = optimoptions('lsqnonlin','Algorithm','levenberg-marquardt');
  [x,fval,exitflag] = lsqnonlin(fun,x0,lb,ub,options);
  ```
- Code generation supports these options:

  - `Algorithm` — Must be `'levenberg-marquardt'`

- FiniteDifferenceStepSize
- FiniteDifferenceType
- FunctionTolerance
- MaxFunctionEvaluations
- MaxIterations
- SpecifyObjectiveGradient
- StepTolerance
- TypicalX

- Generated code has limited error checking for options. The recommended way to update an option is to use `optimoptions`, not dot notation.

```
opts = optimoptions('lsqnonlin','Algorithm','levenberg-marquardt');
opts = optimoptions(opts,'MaxIterations',1e4); % Recommended
opts.MaxIterations = 1e4; % Not recommended
```

- Do not load options from a file. Doing so can cause code generation to fail. Instead, create options in your code.

- Usually, if you specify an option that is not supported, the option is silently ignored during code generation. However, if you specify a plot function or output function by using dot notation, code generation can issue an error. For reliability, specify only supported options.

- Because output functions and plot functions are not supported, solvers do not return the exit flag –1.

Generated Code Not Multithreaded

By default, generated code for use outside the MATLAB environment uses linear algebra libraries that are not multithreaded. Therefore, this code can run significantly slower than code in the MATLAB environment.

If your target hardware has multiple cores, you can achieve better performance by using custom multithreaded LAPACK and BLAS libraries. To incorporate these libraries in your generated code, see "Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls" (MATLAB Coder).

See Also
codegen | lsqcurvefit | lsqnonlin | optimoptions

More About
- "Generate Code for lsqcurvefit or lsqnonlin" on page 12-89
- "Static Memory Allocation for fmincon Code Generation" on page 6-131
- "Optimization Code Generation for Real-Time Applications" on page 6-133

Generate Code for lsqcurvefit or lsqnonlin

This example shows how to generate C code for nonlinear least squares.

Data and Model for Least Squares

In this example, the vector `xdata` represents 100 data points, and the vector `ydata` represents the associated measurements. The modeled relationship between `xdata` and `ydata` is

$$\text{ydata}_i = a_1 + a_2 \exp(-a_3 \text{xdata}_i) + \varepsilon_i.$$

Generate the data for the problem.

```
rng(5489,'twister') % For reproducibility
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)+...
    0.5*randn(100,1))).*xdata);
```

The code generates `xdata` from 100 independent samples of an exponential distribution with mean 2. The code generates `ydata` from its defining equation using `a = [1;3;2]`, perturbed by adding normal deviates with standard deviations `[0.1;0.5;0.5]`.

Solve Generating Code for lsqcurvefit

Solver Approach

The goal is to find parameters for the model $\hat{a}_i$, $i = 1, 2, 3$ that best fit the data.

To fit the parameters to the data using `lsqcurvefit`, you need to define a fitting function. For `lsqcurvefit`, the fitting function takes a parameter vector `a` and the data `xdata` and returns a prediction of the response, which should be equal to `ydata` with no noise and a perfect model. 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.

```
a0 = [2;2;2];
```

Call `lsqcurvefit` to find the best-fitting parameters $\hat{a}_i$.

```
[ahat,resnorm,residual,exitflag,output,lambda,jacobian] =...
    lsqcurvefit(predicted,a0,xdata,ydata);
```

Code Generation Approach

To solve the same problem using code generation, complete the following steps.

1 Write a function that incorporates all of the previous steps: generate data, create a fitting function, create an initial point, and call `lsqcurvefit`.

```
function [x,res] = solvelsqcurve
rng(5489,'twister') % For reproducibility
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)+...
            0.5*randn(100,1))).*xdata);
        predicted = @(a,xdata) a(1)*ones(100,1)+a(2)*exp(-a(3)*xdata);
        options = optimoptions('lsqcurvefit','Algorithm','levenberg-marquardt','Display','off');
        a0 = [2;2;2];
        lb = [];
        ub = [];
        [x,res] = lsqcurvefit(predicted,a0,xdata,ydata,lb,ub,options);
    end
```

2   Create a configuration for code generation. In this case, use `'mex'`.

```
    cfg = coder.config('mex');
```

3   Generate code for the `solvelsqcurve` function.

```
    codegen -config cfg solvelsqcurve
```

4   Test the generated code by running the generated file, which is named
`solvelsqcurve_mex.mexw64` or similar.

```
    [x,res] = solvelsqcurve_mex

    x =

        1.0169
        3.1444
        2.1596


    res =

        7.4101
```

Solve Generating Code for lsqnonlin

Solver Approach

The goal is to find parameters for the model $\widehat{a}_i$, $i = 1, 2, 3$ that best fit the data.

To fit the parameters to the data using `lsqnonlin`, you need to define a fitting function. For
`lsqnonlin`, the fitting function takes a parameter vector `a`, the data `xdata`, and the data `ydata`. The
fitting function returns the difference between the prediction of a response and the data `ydata`,
which should equal 0 with no noise and a perfect model. Define the fitting function `predicted` as an
anonymous function.

```
predicted = @(a)(a(1)*ones(100,1)+a(2)*exp(-a(3)*xdata) - ydata)
```

To fit the model to the data, `lsqnonlin` needs an initial estimate `a0` of the parameters.

```
a0 = [2;2;2];
```

Call `lsqnonlin` to find the best-fitting parameters $\widehat{a}_i$.

```
[ahat,resnorm,residual,exitflag,output,lambda,jacobian] =...
    lsqnonlin(predicted,a0);
```

Code Generation Approach

To solve the same problem using code generation, complete the following steps.

1 Write a function that incorporates all of the previous steps: generate data, create a fitting function, create an initial point, and call `lsqnonlin`.

```
function [x,res] = solvelsqnon
rng(5489,'twister') % For reproducibility
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)+...
    0.5*randn(100,1))).*xdata);
predicted = @(a) (a(1)*ones(100,1)+a(2)*exp(-a(3)*xdata) - ydata);
options = optimoptions('lsqnonlin','Algorithm','levenberg-marquardt','Display','off');
a0 = [2;2;2];
lb = [];
ub = [];
[x,res] = lsqnonlin(predicted,a0,lb,ub,options);
end
```

2 Create a configuration for code generation. In this case, use `'mex'`.

```
cfg = coder.config('mex');
```

3 Generate code for the `solvelsqnon` function.

```
codegen -config cfg solvelsqnon
```

4 Test the generated code by running the generated file, which is named `solvelsqnon_mex.mexw64` or similar.

```
[x,res] = solvelsqnon_mex

x =

    1.0169
    3.1444
    2.1596


res =

    7.4101
```

The solution is identical to the one generated by `solvelsqcurve_mex` because the solvers have identical underlying algorithms. So, you can use the solver you find most convenient.

See Also
codegen | lsqcurvefit | lsqnonlin | optimoptions

More About
- "Code Generation in Nonlinear Least Squares: Background" on page 12-87
- "Static Memory Allocation for fmincon Code Generation" on page 6-131
- "Optimization Code Generation for Real-Time Applications" on page 6-133

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 in 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 a system of equations by minimizing the sum of squares of the components. If the sum of squares is zero, the system of equations is solved. `fsolve` has three algorithms:

- Trust-region
- Trust-region-dogleg
- Levenberg-Marquardt

All algorithms are large scale; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-10.

The `fzero` function solves a single one-dimensional equation.

The `mldivide` function solves a system of linear equations.

Trust-Region 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 that the current point is x in n-space and you want to improve by moving to a point with a lower function value. To do so, the algorithm approximates 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. The solver computes a trial step s by minimizing (or approximately minimizing) over N. The trust-region subproblem is

$$\min_s \{ q(s),\ s \in N \}.$$

The solver updates the current point to $x + s$ if $f(x + s) < f(x)$; otherwise, the current point remains unchanged and the solver shrinks N (the trust region) and repeats the trial step computation.

The key questions in defining a specific trust-region approach to minimizing $f(x)$ are how to choose and compute the approximation q (defined at the current point x), how to choose and modify the trust region N, and how accurately to solve the trust-region subproblem.

In 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

$$\min\left\{\frac{1}{2}s^T H s + s^T g \ \text{ such that } \ \|Ds\| \le \Delta\right\},$$ (13-1)

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. To solve "Equation 13-1", an algorithm (see [48]) can compute all eigenvalues of H and then apply a Newton process to the secular equation

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

Such an algorithm provides an accurate solution to "Equation 13-1". However, this requires time proportional to several factorizations of H. Therefore, trust-region problems require a different approach. Several approximation and heuristic strategies, based on "Equation 13-1", have been proposed in the literature ([42] and [50]). Optimization Toolbox solvers follow an approximation approach that restricts the trust-region subproblem to a two-dimensional subspace S ([39] and [42]). After the solver computes the subspace S, the work to solve "Equation 13-1" is trivial because, in the subspace, the problem is only two-dimensional. The dominant work now shifts to the determination of the subspace.

The solver determines the two-dimensional subspace S with the aid of a preconditioned conjugate gradient method (described in the next section). 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, that is, 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).

The process of unconstrained minimization using the trust-region approach is now easy to specify:

1 Formulate the two-dimensional trust-region subproblem.

2 Solve "Equation 13-1" to determine the trial step s.

3 If $f(x + s) < f(x)$, then $x = x + s$.

4 Adjust Δ.

The solver repeats these four steps until convergence, adjusting he trust-region dimension Δ according to standard rules. In particular, the solver decreases the trust-region size if it does not accept the trial step, when $f(x + s) \ge f(x)$. See [46] and [49] for a discussion of this aspect.

Optimization Toolbox solvers treat important 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.

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 it encounters a direction of negative (or zero) curvature, that is, $d^T H d \leq 0$. The PCG output direction p is either a direction of negative curvature or an approximate solution to the Newton system $Hp = -g$. In either case, p helps to 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 Algorithm

Another approach is to solve a linear system of equations to find the search direction. Newton's method specifies 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 be problematic. $J(x_k)$ might be singular, in which case the Newton step d_k is not even defined. Also, the exact Newton step d_k can be expensive to compute. In addition, Newton's method might 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) handles the case when $J(x_k)$ is singular and improves robustness when the starting point is far from the solution. To use a trust-region strategy, you need a merit function 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

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

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

$$\min_d m(d) = \frac{1}{2}\|M(x_k + d)\|_2^2 = \frac{1}{2}\|F(x_k) + J(x_k)d\|_2^2$$

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

(13-2)

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

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

(13-3)

such that $\|D\cdot d\| \le \Delta$. You can solve this subproblem efficiently 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 the trust-region-dogleg algorithm is the use of the Powell dogleg procedure for computing the step d, which minimizes "Equation 13-3". For a detailed description, see Powell [34].

The algorithm constructs the step d 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 minimizes "Equation 13-2".

The Gauss-Newton step is calculated by solving

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

using the MATLAB `mldivide` (matrix left division) operator.

The algorithm chooses the step d so that

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

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

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

Levenberg-Marquardt Method

The Levenberg-Marquardt algorithm ([25], and [27]) uses a search direction that is a solution of the linear set of equations

$$\left(J(x_k)^T J(x_k) + \lambda_k I\right)d_k = -J(x_k)^T F(x_k),$$

(13-4)

or, optionally, of the equations

$$\left(J(x_k)^T J(x_k) + \lambda_k \text{diag}\left(J(x_k)^T J(x_k)\right)\right)d_k = -J(x_k)^T F(x_k),$$

(13-5)

where the scalar λ_k controls both the magnitude and direction of d_k. Set the `fsolve` option `ScaleProblem` to `'none'` to use "Equation 13-4", or set this option to `'jacobian'` to use "Equation 13-5".

When λ_k is zero, the direction d_k is the Gauss-Newton method. As λ_k tends towards infinity, d_k tends towards the steepest descent direction, with magnitude tending towards zero. The implication is that, for some sufficiently large λ_k, the term $F(x_k + d_k) < F(x_k)$ holds true. Therefore, the algorithm can control the term λ_k to ensure descent despite second-order terms, which restrict the efficiency of the Gauss-Newton method. The Levenberg-Marquardt algorithm, therefore, uses a search direction that is a cross between the Gauss-Newton direction and the steepest descent direction. For more details, see "Levenberg-Marquardt Method" on page 12-6 in the least squares documentation.

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 specify an initial interval instead of an initial point, `fzero` checks to make sure that $f(x)$ 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.

\ Algorithm

The \ algorithm is described in the MATLAB arithmetic operators section for `mldivide`.

See Also
`fsolve` | `fzero`

More About
- "Systems of Nonlinear Equations"

Solve Nonlinear System Without and Including Jacobian

This example shows the reduction in function evaluations when you provide derivatives for a system of nonlinear equations. As explained in "Writing Vector and Matrix Objective Functions" on page 2-26, the Jacobian $J(x)$ of a system of equations $F(x)$ is $J_{ij}(x) = \dfrac{\partial F_i(x)}{\partial x_j}$. Provide this derivative as the second output of your objective function.

For example, the multirosenbrock function is an n-dimensional generalization of Rosenbrock's function (see "Solve a Constrained Nonlinear Problem, Problem-Based" on page 1-5) for any positive even value of n:

$$F(1) = 1 - x_1$$
$$F(2) = 10\left(x_2 - x_1^2\right)$$
$$F(3) = 1 - x_3$$
$$F(4) = 10\left(x_4 - x_3^2\right)$$
$$\vdots$$
$$F(n-1) = 1 - x_{n-1}$$
$$F(n) = 10\left(x_n - x_{n-1}^2\right).$$

The solution of the equation system $F(x) = 0$ is the point $x_i = 1$, $i = 1...n$.

For this objective function, all of the Jacobian terms $J_{ij}(x)$ are zero except for the terms where i and j differ by at most one. For odd values of $i < n$, the nonzero terms are

$$J_{ii}(x) = -1$$
$$J_{(i+1)i} = -20x_i$$
$$J_{(i+1)(i+1)} = 10.$$

The `multirosenbrock` helper function at the end of this example on page 13-0   creates the objective function $F(x)$ and its Jacobian $J(x)$.

Solve the system of equations starting from the point $x_i = -1.9$ for odd values of $i < n$, and $x_i = 2$ for even values of i. Choose $n = 64$.

```
n = 64;
x0(1:n,1) = -1.9;
x0(2:2:n,1) = 2;
[x,F,exitflag,output,JAC] = fsolve(@multirosenbrock,x0);
```

```
Equation solved.

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

Examine the distance of the computed solution x from the true solution, and the number of function evaluations that `fsolve` takes to compute the solution.

```
disp(norm(x-ones(size(x))))
```

```
        0
```

```
disp(output.funcCount)
```

```
        1043
```

`fsolve` finds the solution, and takes over 1000 function evaluations to do so.

Solve the system of equations again, this time using the Jacobian. To do so, set the `'SpecifyObjectiveGradient'` option to `true`.

```
opts = optimoptions('fsolve','SpecifyObjectiveGradient',true);
[x2,F2,exitflag2,output2,JAC2] = fsolve(@multirosenbrock,x0,opts);
```

```
Equation solved.
```

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

Again, examine the distance of the computed solution `x2` from the true solution, and the number of function evaluations that `fsolve` takes to compute the solution.

```
disp(norm(x2-ones(size(x2))))
```

```
        0
```

```
disp(output2.funcCount)
```

```
        21
```

`fsolve` returns the same solution as before, but takes about 20 function evaluations to do so, rather than about 1000. In general, using the Jacobian can save computation and can provide increased robustness, although this example does not show the robustness improvement.

This code creates the `multirosenbrock` helper function.

```matlab
function [F,J] = multirosenbrock(x)
% 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
end
```

See Also
`fsolve`

More About

- "Systems of Nonlinear Equations"

Large Sparse System of Nonlinear Equations with Jacobian

This example shows how to use features of the `fsolve` solver to solve large sparse systems of equations effectively. The example uses the objective function, defined for a system of n equations,

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

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

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

The equations to solve are $F_i(x) = 0$, $1 \le i \le n$. The example uses $n = 1000$.

This objective function is simple enough that you can calculate its Jacobian analytically. As explained in "Writing Vector and Matrix Objective Functions" on page 2-26, the Jacobian $J(x)$ of a system of equations $F(x)$ is $J_{ij}(x) = \dfrac{\partial F_i(x)}{\partial x_j}$. Provide this derivative as the second output of your objective function. The `nlsf1` helper function at the end of this example on page 13-0   creates the objective function $F(x)$ and its Jacobian $J(x)$.

Solve the system of equations using the default options, which call the `'trust-region-dogleg'` algorithm. Start from the point `xstart(i) = -1`.

```
n = 1000;
xstart = -ones(n,1);
fun = @nlsf1;
[x,fval,exitflag,output] = fsolve(fun,xstart);
```

```
Equation solved.

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

Display the solution quality and the number of function evaluations taken.

```
disp(norm(fval))
```

```
   2.8577e-13
```

```
disp(output.funcCount)
```

```
      7007
```

The equation is solved accurately, but `fsolve` takes thousands of function evaluations to do so.

Solve the equation using the Jacobian in both the default and `'trust-region'` algorithms.

```
options = optimoptions('fsolve','SpecifyObjectiveGradient',true);
[x2,fval2,exitflag2,output2] = fsolve(fun,xstart,options);
```

```
Equation solved.

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

```
options.Algorithm = 'trust-region';
[x3,fval3,exitflag3,output3] = fsolve(fun,xstart,options);
```

```
Equation solved.

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

```
disp([norm(fval2),norm(fval3)])
```

```
   1.0e-08 *

   0.0000    0.1065
```

```
disp([output2.funcCount,output3.funcCount])
```

```
     7     5
```

Both algorithms take a tiny fraction of the number of function evaluations compared to the number without using the Jacobian. The default algorithm takes a few more function evaluations than the `'trust-region'` algorithm, but the default algorithm reaches a more accurate answer.

See whether setting the `'PrecondBandWidth'` option to 1 changes the `'trust-region'` answer or efficiency. This setting causes `fsolve` to use a tridiagonal preconditioner, which should be effective for this tridiagonal system of equations.

```
options.PrecondBandWidth = 1;
[x4,fval4,exitflag4,output4] = fsolve(fun,xstart,options);
```

```
Equation solved, inaccuracy possible.

fsolve stopped because the vector of function values is near zero, as measured by the value
of the function tolerance. However, the last step was ineffective.
```

```
disp(norm(fval4))
```

```
   3.1185e-05
```

```
disp(output4.funcCount)
```

```
     6
```

```
disp(output4.cgiterations)
```

```
     8
```

This option setting causes `fsolve` to give a slightly less accurate answer as measured by the norm of the residual. The number of function evaluations increased slightly, from 5 to 6. The solver had fewer than ten conjugate gradient iterations as part of its solution process.

See how well `fsolve` performs with a diagonal preconditioner.

```
options.PrecondBandWidth = 0;
[x5,fval5,exitflag5,output5] = fsolve(fun,xstart,options);
```

```
Equation solved, inaccuracy possible.

fsolve stopped because the vector of function values is near zero, as measured by the value
of the function tolerance. However, the last step was ineffective.
```

```
disp(norm(fval5))

    2.0057e-05
```

```
disp(output5.funcCount)

     6
```

```
disp(output5.cgiterations)

    19
```

The residual norm is slightly lower this time, and the number of function evaluations is unchanged. The number of conjugate gradient iterations increases from 8 to 19, indicating that this setting causes the solver to do more work.

Solve the equations using the `'levenberg-marquardt'` algorithm.

```
options = optimoptions('fsolve','SpecifyObjectiveGradient',true,'Algorithm','levenberg-marquardt'
[x6,fval6,exitflag6,output6] = fsolve(fun,xstart,options);

Equation solved.

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

```
disp(norm(fval6))

    7.5854e-15
```

```
disp(output6.funcCount)

     6
```

This algorithm gives the lowest residual norm and uses only one more than the lowest number of function evaluations.

Summarize the results.

```
t = table([norm(fval);norm(fval2);norm(fval3);norm(fval4);norm(fval5);norm(fval6)],...
    [output.funcCount;output2.funcCount;output3.funcCount;output4.funcCount;output5.funcCount;ou
    'VariableNames',["Residual" "Fevals"],...
    'RowNames',["Default" "Default+Jacobian" "Trust-Region+Jacobian" "Trust-Region+Jacobian,BW=1"
```

```
t=6×2 table
```

	Residual	Fevals
Default	2.8577e-13	7007
Default+Jacobian	2.5886e-13	7
Trust-Region+Jacobian	1.0646e-09	5
Trust-Region+Jacobian,BW=1	3.1185e-05	6
Trust-Region+Jacobian,BW=0	2.0057e-05	6
Levenberg-Marquardt+Jacobian	7.5854e-15	6

This code creates the `nlsf1` function.

```
function [F,J] = nlsf1(x)
% Evaluate the vector function
```

```matlab
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
end
```

See Also

`fsolve`

More About

- "Large System of Nonlinear Equations with Jacobian Sparsity Pattern" on page 13-14
- "Systems of Nonlinear Equations"

Large System of Nonlinear Equations with Jacobian Sparsity Pattern

This example shows how to supply a Jacobian Sparsity Pattern to solve large a sparse system of equations effectively. The example uses the objective function, defined for a system of n equations,

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

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

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

The equations to solve are $F_i(x) = 0$, $1 \le i \le n$. The example uses $n = 1000$. The `nlsf1a` helper function at the end of this example on page 13-0   implements the objective function $F(x)$.

In the example "Large Sparse System of Nonlinear Equations with Jacobian" on page 13-10, which solves the same system, the objective function has an explicit Jacobian. However, sometimes you cannot compute the Jacobian explicitly, and yet you can determine which elements of the Jacobian are nonzero. In this example, the Jacobian is tridiagonal, because the only variables that appear in the definition of $F(i)$ are x_{i-1}, x_i, and x_{i+1}. So the only nonzero parts of the Jacobian are on the main diagonal and its two neighboring diagonals. The Jacobian sparsity pattern is a matrix whose nonzero elements correspond to (potentially) nonzero elements in the Jacobian.

Create a sparse n-by-n tridiagonal matrix of ones representing the Jacobian sparsity pattern. (For an explanation of this code, see `spdiags`.)

```
n = 1000;
e = ones(n,1);
Jstr = spdiags([e e e],-1:1,n,n);
```

View the structure of `Jstr`.

```
spy(Jstr)
```

nz = 2998
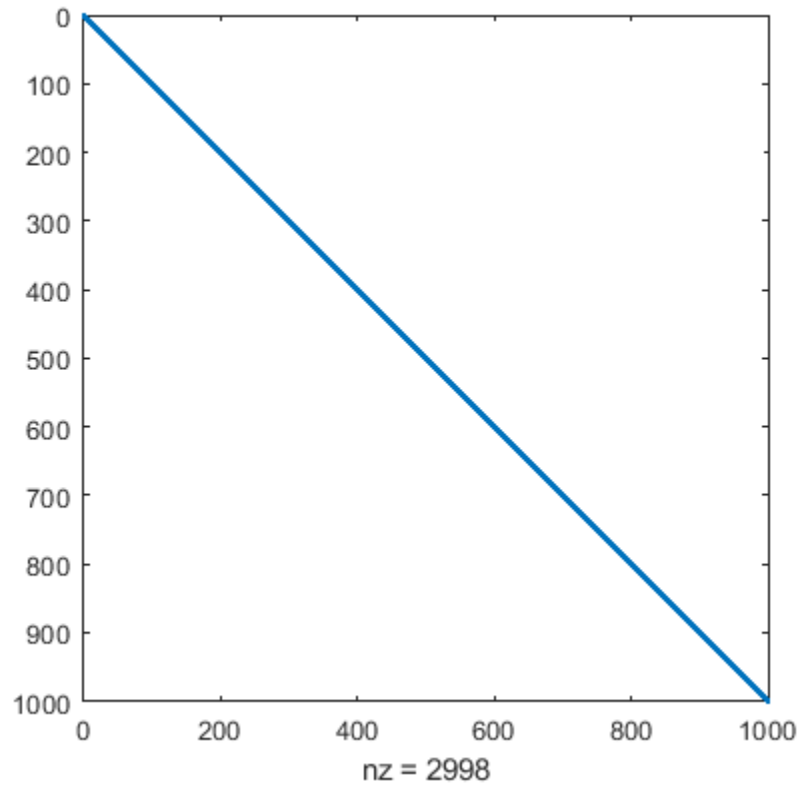
Set `fsolve` options to use the `'trust-region'` algorithm, which is the only `fsolve` algorithm that can make use of a Jacobian sparsity pattern.

```
options = optimoptions('fsolve','Algorithm','trust-region','JacobPattern',Jstr);
```

Set the initial point `x0` to a vector of –1 entries.

```
x0 = -1*ones(n,1);
```

Solve the problem.

```
[x,fval,exitflag,output] = fsolve(@nlsf1a,x0,options);
```

```
Equation solved.

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

Examine the resulting residual norm and the number of function evaluations that `fsolve` takes.

```
disp(norm(fval))
```

```
   1.0523e-09
```

```
disp(output.funcCount)
```

```
   25
```

The residual norm is near zero, indicating that `fsolve` solves the problem correctly. The number of function evaluations is fairly small, just a few dozen. Compare this number of function evaluations to the number without a supplied Jacobian sparsity pattern.

```
options = optimoptions('fsolve','Algorithm','trust-region');
[x,fval,exitflag,output] = fsolve(@nlsf1a,x0,options);

Equation solved.

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

```
disp(norm(fval))

   1.0659e-09
```

```
disp(output.funcCount)

      5005
```

The solver reaches essentially the same solution as before, but takes thousands of function evaluations to do so.

This code creates the `nlsf1a` function.

```
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;
end
```

See Also

`fsolve`

More About

- "Large Sparse System of Nonlinear Equations with Jacobian" on page 13-10
- "Systems of Nonlinear Equations"

Nonlinear Systems with Constraints

Solve Equations with Inequality Constraints

`fsolve` solves a system of nonlinear equations. However, it does not allow you to include any constraints, even bound constraints. So how can you solve a system of nonlinear equations when you have constraints?

A solution that satisfies your constraints is not guaranteed to exist. In fact, the problem might not have any solution, even one that does not satisfy your constraints. However techniques exist to 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. The equations have four solutions:

$x = (-1, -2)$
$x = (10, -2)$
$x = (-1, 20)$
$x = (10, 20).$

Only one solution satisfies the constraints, namely $x = (10, 20)$.

The `fbnd` helper function at the end of this example on page 13-0 calculates $F(x)$ numerically.

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 then run `fsolve` starting at each x0.

For this example, to look for a solution to , 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
```

List solutions that satisfy the constraints.

```
idx = soln(:,1) >= 0 & soln(:,2) >= 0;
disp(soln(idx,:))
```

```
  10.0000    20.0000
  10.0000    20.0000
  10.0000    20.0000
  10.0000    20.0000
  10.0000    20.0000
```

Use Different Algorithms

`fsolve` has three 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 = 1×2

  -1.0000    -2.0000
```

```
opts.Algorithm = 'trust-region';
x2 = fsolve(@fbnd,x0,opts)
```

```
x2 = 1×2

  -1.0000    20.0000
```

```
opts.Algorithm = 'levenberg-marquardt';
x3 = fsolve(@fbnd,x0,opts)
```

```
x3 = 1×2

   0.9523     8.9941
```

Here, all three algorithms find different solutions for the same initial point. None happen to satisfy the constraints. The reported "solution" `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 in a vector function $F(x)$. Therefore, it attempts to solve the equation $F(x) = 0$. Also, `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)
```

```
Local minimum found.

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

```
x = 2×1

   10.0000
   20.0000
```

```
res = 2.4783e-25
```

In this case, `lsqnonlin` converged to the solution satisfying the constraints. You can use `lsqnonlin` with the Global Optimization Toolbox `MultiStart` solver to search over many initial points automatically. See "MultiStart Using lsqcurvefit or lsqnonlin" (Global Optimization Toolbox).

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.

The `fminconstr` helper function at the end of this example on page 13-0   implements the nonlinear constraints. 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 = 2×1

   10.0000
   20.0000
```

In this case, `fmincon` solves the problem from the start point.

Helper Functions

This code creates the `fbnd` helper function.

```
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));
end
```

This code creates the `fminconstr` helper function.

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

c = []; % No nonlinear inequality
ceq = fbnd(x); % fsolve objective is fmincon nonlinear equality constraints
end
```

See Also

fmincon | fsolve | lsqnonlin

More About

- "Nonlinear System of Equations with Constraints, Problem-Based" on page 13-32
- "Systems of Nonlinear Equations"
- "When the Solver Fails" on page 4-3

Solve Nonlinear System of Equations, Problem-Based

To solve the nonlinear system of equations

$$\exp(-\exp(-(x_1 + x_2))) = x_2\left(1 + x_1^2\right)$$

$$x_1\cos(x_2) + x_2\sin(x_1) = \frac{1}{2}$$

using the problem-based approach, first define x as a two-element optimization variable.

```
x = optimvar('x',2);
```

Create the first equation as an optimization equality expression.

```
eq1 = exp(-exp(-(x(1) + x(2)))) == x(2)*(1 + x(1)^2);
```

Similarly, create the second equation as an optimization equality expression.

```
eq2 = x(1)*cos(x(2)) + x(2)*sin(x(1)) == 1/2;
```

Create an equation problem, and place the equations in the problem.

```
prob = eqnproblem;
prob.Equations.eq1 = eq1;
prob.Equations.eq2 = eq2;
```

Review the problem.

```
show(prob)
```

```
  EquationProblem :

    Solve for:
       x


  eq1:
       exp(-exp(-(x(1) + x(2)))) == (x(2) .* (1 + x(1).^2))

  eq2:
       ((x(1) .* cos(x(2))) + (x(2) .* sin(x(1)))) == 0.5
```

Solve the problem starting from the point [0,0]. For the problem-based approach, specify the initial point as a structure, with the variable names as the fields of the structure. For this problem, there is only one variable, x.

```
x0.x = [0 0];
[sol,fval,exitflag] = solve(prob,x0)
```

```
Solving problem using fsolve.

Equation solved.

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

```
sol = struct with fields:
    x: [2x1 double]


fval = struct with fields:
    eq1: -2.4069e-07
    eq2: -3.8253e-08


exitflag =
    EquationSolved
```

View the solution point.

```
disp(sol.x)

    0.3532
    0.6061
```

Unsupported Functions Require `fcn2optimexpr`

If your equation functions are not composed of elementary functions, you must convert the functions to optimization expressions using `fcn2optimexpr`. For the present example:

```
ls1 = fcn2optimexpr(@(x)exp(-exp(-(x(1)+x(2)))),x);
eq1 = ls1 == x(2)*(1 + x(1)^2);
ls2 = fcn2optimexpr(@(x)x(1)*cos(x(2))+x(2)*sin(x(1)),x);
eq2 = ls2 == 1/2;
```

See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8.

See Also
`fcn2optimexpr` | `solve`

More About
*   "Convert Nonlinear Function to Optimization Expression" on page 7-8
*   "Systems of Nonlinear Equations"
*   "Problem-Based Workflow for Solving Equations" on page 10-4

Solve Nonlinear System of Polynomials, Problem-Based

When x is a 2-by-2 matrix, the equation

$$x^3 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

is a system of polynomial equations. Here, x^3 means $x*x*x$ using matrix multiplication. You can easily formulate and solve this system using the problem-based approach.

First, define the variable x as a 2-by-2 matrix variable.

```
x = optimvar('x',2,2);
```

Define the equation to be solved in terms of x.

```
eqn = x^3 == [1 2;3 4];
```

Create an equation problem with this equation.

```
prob = eqnproblem('Equations',eqn);
```

Solve the problem starting from the point [1 1;1 1].

```
x0.x = ones(2);
sol = solve(prob,x0)
```

```
Solving problem using fsolve.

Equation solved.

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

sol = struct with fields:
    x: [2x2 double]
```

Examine the solution.

```
disp(sol.x)
```

```
   -0.1291    0.8602
    1.2903    1.1612
```

Display the cube of the solution.

```
sol.x^3
```

```
ans = 2×2

    1.0000    2.0000
    3.0000    4.0000
```

See Also
`solve`

More About
- "Systems of Nonlinear Equations"
- "Problem-Based Workflow for Solving Equations" on page 10-4

Follow Equation Solution as a Parameter Changes

This example shows how to solve an equation repeatedly as a parameter changes by starting subsequent solutions from the previous solution point. Often, this process leads to efficient solutions. However, a solution can sometimes disappear, requiring a start from a new point or points.

Parameterized Scalar Equation

The parameterized equation to solve is

$\sinh(x) - 3x = a$,

where a is a numeric parameter that goes from 0 to 5. At $a = 0$, one solution to this equation is $x = 0$. When a is not too large in absolute value, the equation has three solutions. To visualize the equation, create the left side of the equation as an anonymous function. Plot the function.

```
fun = @(x)sinh(x) - 3*x;
t = linspace(-3.5,3.5);
plot(t,fun(t),t,zeros(size(t)),'k-')
xlabel('x')
ylabel('fun(x)')
```
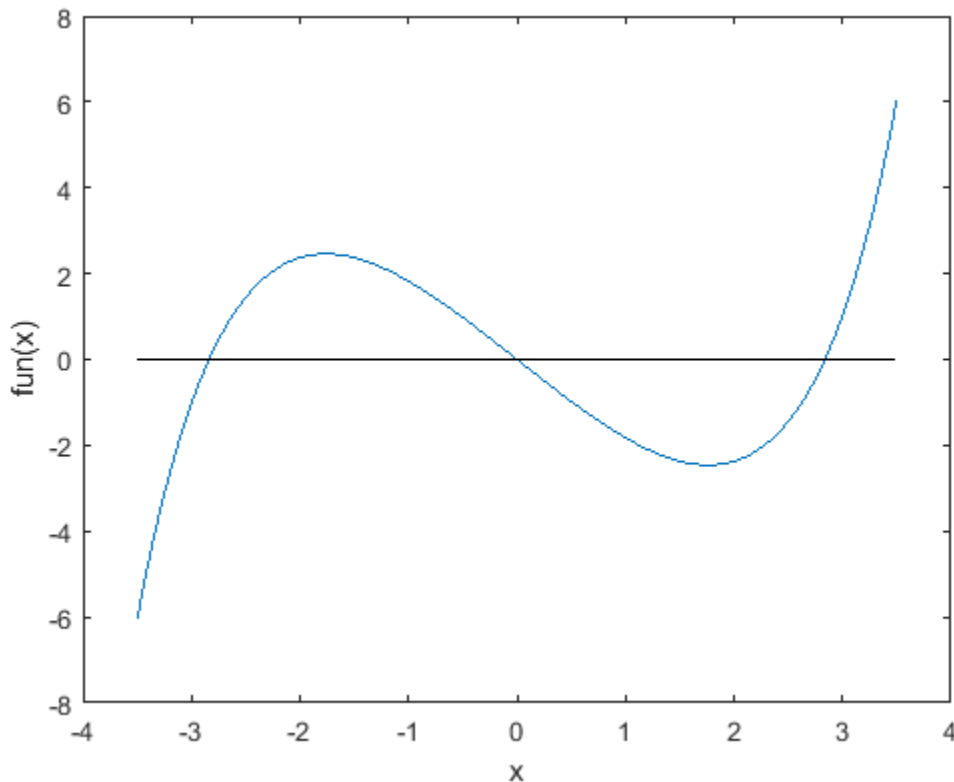

When a is too large or too small, there is only one solution.

Problem-Based Setup

To create an objective function for the problem-based approach, create an optimization expression `expr` in an optimization variable `x`.

```
x = optimvar('x');
expr = sinh(x) - 3*x;
```

Create and Plot Solutions

Starting from the initial solution $x = 0$ at $a = 0$, find solutions for 100 values of a from 0 through 5. Because `fun` is a scalar nonlinear function, `solve` calls `fzero` as the solver.

Set up the problem object, options, and data structures for holding solution statistics.

```
prob = eqnproblem;
options = optimset('Display','off');
sols = zeros(100,1);
fevals = sols;
as = linspace(0,5);
```

Solve the equation in a loop, starting from the first solution `sols(1) = 0`.

```
for i = 2:length(as)
    x0.x = sols(i-1); % Start from previous solution
    prob.Equations = expr == as(i);
    [sol,~,~,output] = solve(prob,x0,'Options',options);
    sols(i) = sol.x;
    fevals(i) = output.funcCount;
end
```

Plot the solution as a function of the parameter `a` and the number of function evaluations taken to reach the solution.

```
subplot(2,1,1)
plot(as,sols,'ko')
xlabel 'a'
ylabel('Solution(x)')
subplot(2,1,2)
plot(fevals,'k*')
xlabel('Iteration Number')
ylabel('Fevals')
```
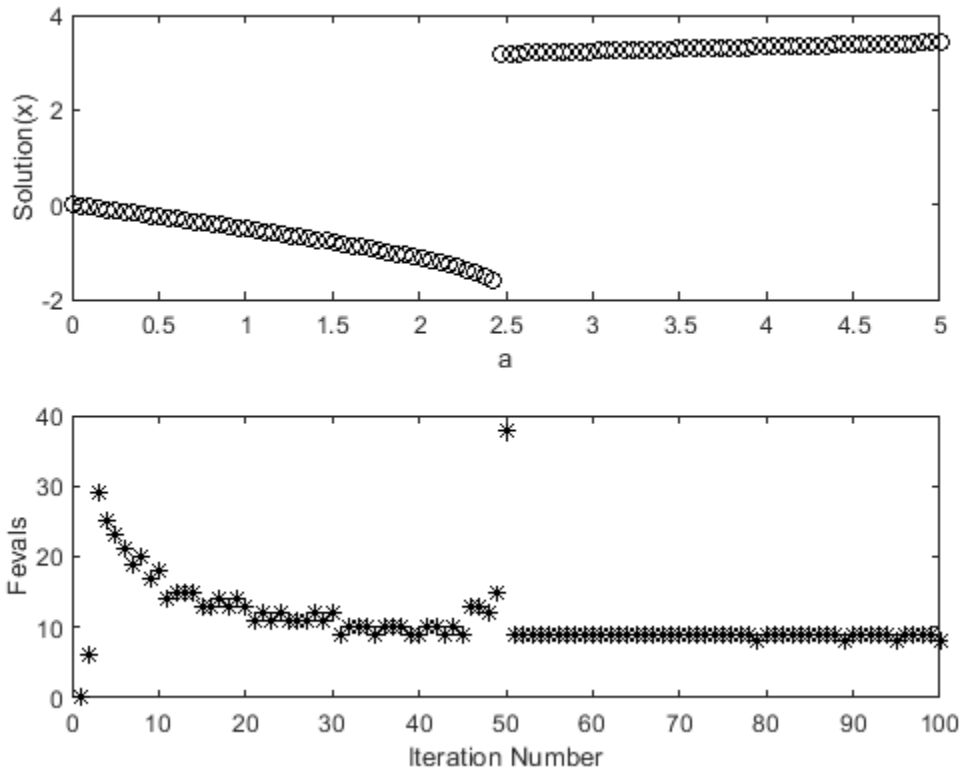
A jump in the solution occurs near $a = 2.5$. At the same point, the number of function evaluations to reach a solution increases from near 15 to near 40. To understand why, examine a more detailed plot of the function. Plot the function and every seventh solution point.

```
figure
t = linspace(-3.5,3.5);
plot(t,fun(t));
hold on
plot([-3.5,min(sols)],[2.5,2.5],'k--')
legend('fun','Maximum a','Location','north','autoupdate','off')
for a0 = 7:7:100
    plot(sols(a0),as(a0),'ro')
    if mod(a0,2) == 1
        text(sols(a0) + 0.15,as(a0) + 0.15,num2str(a0/7))
    else
        text(sols(a0) - 0.3,as(a0) + 0.05,num2str(a0/7))
    end
end
plot(t,zeros(size(t)),'k-')
hold off
```
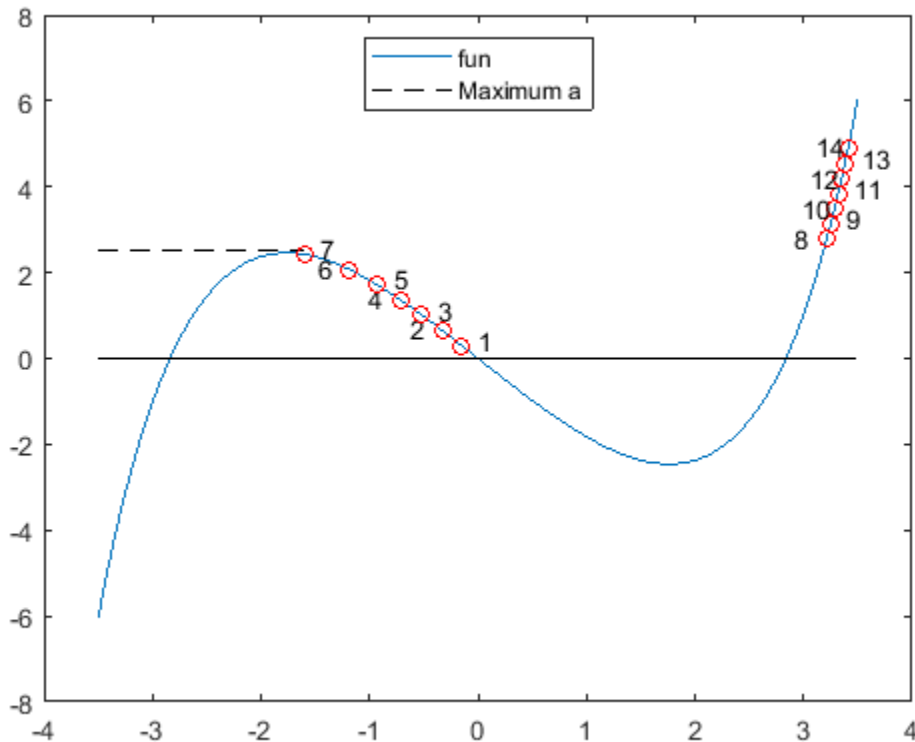
As *a* increases, at first the solutions move to the left. However, when *a* is above 2.5, there is no longer a solution near the previous solution. `fzero` requires extra function evaluations to search for a solution, and finds a solution near $x = 3$. After that, the solution values move slowly to the right as *a* increases further. The solver requires only about 10 function evaluations for each subsequent solution.

Choose Different Solver

The `fsolve` solver can be more efficient than `fzero`. However, `fsolve` can become stuck in a local minimum and fail to solve the equation.

Set up the problem object, options, and data structures for holding solution statistics.

```
probfsolve = eqnproblem;
sols = zeros(100,1);
fevals = sols;
infeas = sols;
asfsolve = linspace(0,5);
```

Solve the equation in a loop, starting from the first solution `sols(1) = 0`.

```
for i = 2:length(as)
    x0.x = sols(i-1); % Start from previous solution
    probfsolve.Equations = expr == asfsolve(i);
    [sol,fval,~,output] = solve(probfsolve,x0,'Options',options,'Solver','fsolve');
    sols(i) = sol.x;
    fevals(i) = output.funcCount;
```

```
    infeas(i) = fval;
end
```

Plot the solution as a function of the parameter `a` and the number of function evaluations taken to reach the solution.

```
subplot(2,1,1)
plot(asfsolve,sols,'ko',asfsolve,infeas,'r-')
xlabel 'a'
legend('Solution','Error of Solution','Location','best')
subplot(2,1,2)
plot(fevals,'k*')
xlabel('Iteration Number')
ylabel('Fevals')
```
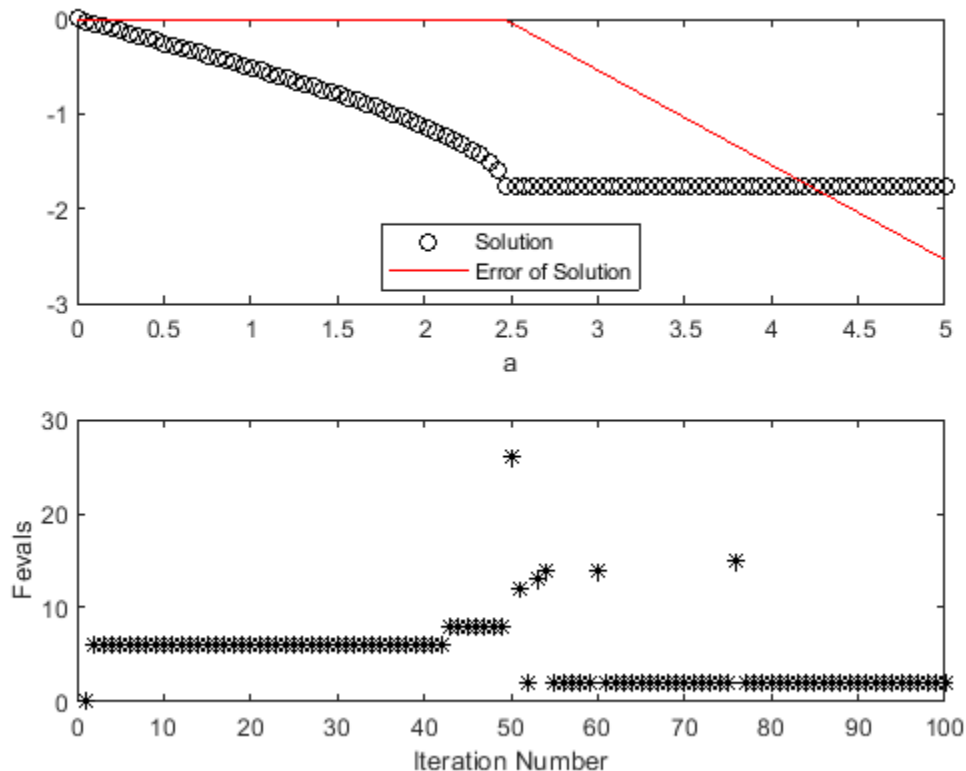

`fsolve` is somewhat more efficient than `fzero`, requiring about 7 or 8 function evaluations per iteration. Again, when the solver finds no solution near the previous value, the solver requires many more function evaluations to search for a solution. This time, the search is unsuccessful. Subsequent iterations require few function evaluations for the most part, but fail to find a solution. The `Error of Solution` plot shows the function value $fun(x) - a$.

To try to overcome a local minimum not being a solution, search again from a different start point when `fsolve` returns with a negative exit flag. Set up the problem object, options, and data structures for holding solution statistics.

```
rng default % For reproducibility
sols = zeros(100,1);
```

```
fevals = sols;
asfsolve = linspace(0,5);
```

Solve the equation in a loop, starting from the first solution `sols(1) = 0`.

```
for i = 2:length(as)
    x0.x = sols(i-1); % Start from previous solution
    probfsolve.Equations = expr == asfsolve(i);
    [sol,~,exitflag,output] = solve(probfsolve,x0,'Options',options,'Solver','fsolve');

    while exitflag <= 0 % If fsolve fails to find a solution
        x0.x = 5*randn; % Try again from a new start point
        fevals(i) = fevals(i) + output.funcCount;
        [sol,~,exitflag,output] = solve(probfsolve,x0,'Options',options,'Solver','fsolve');
    end

    sols(i) = sol.x;
    fevals(i) = fevals(i) + output.funcCount;
end
```

Plot the solution as a function of the parameter `a` and the number of function evaluations taken to reach the solution.

```
subplot(2,1,1)
plot(asfsolve,sols,'ko')
xlabel 'a'
ylabel('Solution(x)')
subplot(2,1,2)
plot(fevals,'k*')
xlabel('Iteration Number')
ylabel('Fevals')
```
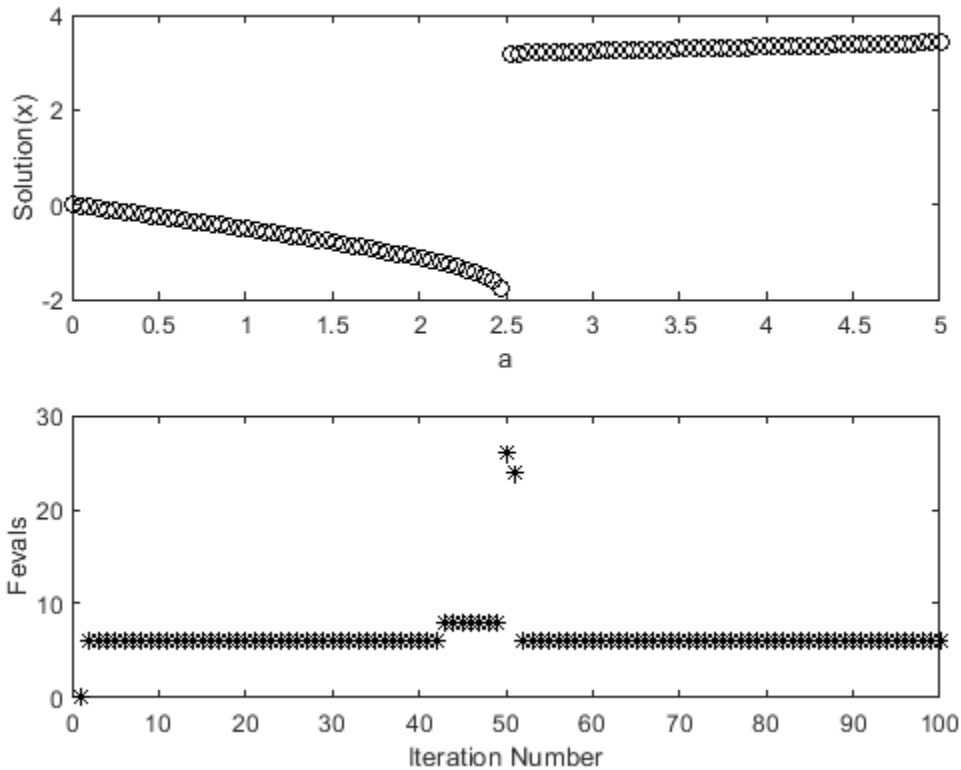
This time, `fsolve` recovers from the poor initial point near $a = 2.5$ and obtains a solution similar to the one obtained by `fzero`. The number of function evaluations for each iteration is typically 8, increasing to about 30 at the point where the solution jumps.

Convert Objective Function Using `fcn2optimexpr`

For some objective functions or software versions, you must convert nonlinear functions to optimization expressions by using `fcn2optimexpr`. See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8. For this example, convert the original function `fun` used for plotting to the optimization expression `expr`:

```
expr = fcn2optimexpr(fun,x);
```

The remainder of the example is exactly the same after this change to the definition of `expr`.

See Also
`fsolve` | `fzero` | `solve`

More About
*   "Systems of Nonlinear Equations"
*   "Problem-Based Workflow for Solving Equations" on page 10-4

Nonlinear System of Equations with Constraints, Problem-Based

This example shows how to attempt to solve a nonlinear system of equations with constraints by using the problem-based approach.

Bound Constraints

When your problem has only bound constraints, the process for solving the problem is straightforward. For example, to find the solution with positive components to the system of equations

$$(x_1 + 1)(10 - x_1)\frac{1 + x_2^2}{1 + x_2^2 + x_2} = 0$$

$$(x_2 + 2)(20 - x_2)\frac{1 + x_1^2}{1 + x_1^2 + x_1} = 0,$$

simply create optimization variables with lower bounds of 0. (These equations have four solutions: where $x_1 = -1$ or $x_1 = 10$, and where $x_2 = -2$ or $x_2 = 20$.)

```
x = optimvar('x',2,"LowerBound",0);
expr1 = (x(1) + 1)*(10 - x(1))*((1 + x(2)^2))/(1 + x(2)^2 + x(2));
expr2 = (x(2) + 2)*(20 - x(2))*((1 + x(1)^2))/(1 + x(1)^2 + x(1));
eqn1 = expr1 == 0;
eqn2 = expr2 == 0;
prob = eqnproblem;
prob.Equations.eqn1 = eqn1;
prob.Equations.eqn2 = eqn2;
x0.x = [15,15];
[sol,fval,exitflag] = solve(prob,x0)
```

Equation problem has bound constraints. Reformulating as a least squares problem.

Solving problem using lsqnonlin.

Equation solved.

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

sol = *struct with fields:*
    x: [2x1 double]

fval = *struct with fields:*
    eqn1: 0
    eqn2: 0

exitflag =
    EquationSolved

View the solution.

```
sol.x
```

ans = *2×1*

```
    10
    20
```

General Constraints

When your problem has general constraints, formulate the problem as an optimization problem, not an equation problem. Set the equations as equality constraints. For example, to solve the preceding equations subject to the nonlinear inequality constraint $\|x\|^2 \leq 10$, remove the bounds on x and formulate the problem as an optimization problem with no objective function.

```
x.LowerBound = [];
circlecons = x(1)^2 + x(2)^2 <= 10;
prob2 = optimproblem;
prob2.Constraints.circlecons = circlecons;
prob2.Constraints.eqn1 = eqn1;
prob2.Constraints.eqn2 = eqn2;
[sol2,fval2,exitflag2] = solve(prob2,x0)
```

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol2 = *struct with fields:*
    x: [2x1 double]

fval2 = 0

exitflag2 =
    OptimalSolution

View the solution.

```
sol2.x
```

ans = *2×1*

```
   -1.0000
   -2.0000
```

General Constraints Using Least Squares Objective

You can also formulate the problem by setting the objective function as a sum of squares, and the general constraints as a constraint. This alternative formulation gives a mathematically equivalent problem, but can result in a different solution because the change in formulation leads the solver to different iterations.

```
prob3 = optimproblem;
prob3.Objective = expr1^2 + expr2^2;
prob3.Constraints.circlecons = circlecons;
[sol3,fval3,exitflag3] = solve(prob3,x0)

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol3 = struct with fields:
    x: [2x1 double]


fval3 = 8.0569e-16

exitflag3 =
    OptimalSolution
```

View the solution.

```
sol3.x

ans = 2×1

   -1.0000
   -2.0000
```

In this case, the least squares objective leads to the same solution as the previous formulation, which uses only constraints.

More About Solving Equations with Constraints

Generally, `solve` attempts to solve a nonlinear system of equations by minimizing the sum of squares of the equation components. In other words, if `LHS(i)` is the left-side expression for equation `i`, and `RHS(i)` is the right-side expression, then `solve` attempts to minimize `sum((LHS – RHS).^2)`.

In contrast, when attempting to satisfy nonlinear constraint expressions, `solve` generally uses `fmincon`, and tries to satisfy the constraints by using different strategies.

In both cases, the solver can fail to solve the equations. For strategies you can use to attempt to find a solution when the solver fails, see "fsolve Could Not Solve Equation" on page 4-8.

See Also
```
solve
```

More About
- "Nonlinear Systems with Constraints" on page 13-17
- "When the Solver Fails" on page 4-3

- "Problem-Based Workflow for Solving Equations" on page 10-4

Code Generation in Nonlinear Equation Solving: Background

What Is Code Generation?

Code generation is the conversion of MATLAB code to C code using MATLAB Coder. Code generation requires a MATLAB Coder license.

Typically, you use code generation to deploy code on hardware that is not running MATLAB. For example, you can deploy code on a robot, using `fsolve` for optimizing movement or planning.

For an example, see "Generate Code for fsolve" on page 13-38. For code generation in other optimization solvers, see "Generate Code for fmincon" on page 6-127, "Generate Code for quadprog" on page 11-60, or "Generate Code for lsqcurvefit or lsqnonlin" on page 12-89.

Requirements for Code Generation

- `fsolve` supports code generation using either the `codegen` function or the MATLAB Coder app. You must have a MATLAB Coder license to generate code.
- The target hardware must support standard double-precision floating-point computations. You cannot generate code for single-precision or fixed-point computations.
- Code generation targets do not use the same math kernel libraries as MATLAB solvers. Therefore, code generation solutions can vary from solver solutions, especially for poorly conditioned problems.
- All code for generation must be MATLAB code. In particular, you cannot use a custom black-box function as an objective function for `fsolve`. You can use `coder.ceval` to evaluate a custom function coded in C or C++. However, the custom function must be called in a MATLAB function.
- `fsolve` does not support the `problem` argument for code generation.

  ```
  [x,fval] = fsolve(problem) % Not supported
  ```
- You must specify the objective function by using function handles, not strings or character names.

  ```
  x = fsolve(@fun,x0,options) % Supported
  % Not supported: fsolve('fun',...) or fsolve("fun",...)
  ```
- For advanced code optimization involving embedded processors, you also need an Embedded Coder license.
- You must include options for `fsolve` and specify them using `optimoptions`. The options must include the `Algorithm` option, set to `'levenberg-marquardt'`.

  ```
  options = optimoptions('fsolve','Algorithm','levenberg-marquardt');
  [x,fval,exitflag] = fsolve(fun,x0,options);
  ```
- Code generation supports these options:

  - `Algorithm` — Must be `'levenberg-marquardt'`
  - `FiniteDifferenceStepSize`
  - `FiniteDifferenceType`
  - `FunctionTolerance`
  - `MaxFunctionEvaluations`
  - `MaxIterations`

- SpecifyObjectiveGradient
- StepTolerance
- TypicalX

- Generated code has limited error checking for options. The recommended way to update an option is to use `optimoptions`, not dot notation.

```
opts = optimoptions('fsolve','Algorithm','levenberg-marquardt');
opts = optimoptions(opts,'MaxIterations',1e4); % Recommended
opts.MaxIterations = 1e4; % Not recommended
```

- Do not load options from a file. Doing so can cause code generation to fail. Instead, create options in your code.

- Usually, if you specify an option that is not supported, the option is silently ignored during code generation. However, if you specify a plot function or output function by using dot notation, code generation can issue an error. For reliability, specify only supported options.

- Because output functions and plot functions are not supported, solvers do not return the exit flag –1.

Generated Code Not Multithreaded

By default, generated code for use outside the MATLAB environment uses linear algebra libraries that are not multithreaded. Therefore, this code can run significantly slower than code in the MATLAB environment.

If your target hardware has multiple cores, you can achieve better performance by using custom multithreaded LAPACK and BLAS libraries. To incorporate these libraries in your generated code, see "Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls" (MATLAB Coder).

See Also
codegen | fsolve | optimoptions

More About
- "Generate Code for fsolve" on page 13-38
- "Static Memory Allocation for fmincon Code Generation" on page 6-131
- "Optimization Code Generation for Real-Time Applications" on page 6-133

Generate Code for fsolve

This example shows how to generate C code for solving systems of nonlinear equations with `fsolve`.

Equation to Solve

The system of nonlinear equations to solve is

$$e^{-e^{-(x_1 + x_2)}} = x_2\left(1 + x_1^2\right)$$

$$x_1\cos(x_2) + x_2\sin(x_1) = \frac{1}{2}.$$

Convert the equations to the form $F(x) = 0$.

$$e^{-e^{-(x_1 + x_2)}} - x_2\left(1 + x_1^2\right) = 0$$

$$x_1\cos(x_2) + x_2\sin(x_1) - \frac{1}{2} = 0.$$

Code Generation Steps

1 Write a function that computes the left side of the two equations. For code generation, your program must allocate all arrays when they are created, and must not change their sizes after creation.

```
function F = root2d(x)
F = zeros(2,1); % Allocate return array
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;
end
```

2 Write a function that sets up the problem and calls `fsolve`. The function must refer to `root2d` as a function handle, not as a name.

```
function [x,fval] = solveroot
options = optimoptions('fsolve','Algorithm','levenberg-marquardt','Display','off');
fun = @root2d;
rng default
x0 = rand(2,1);
[x,fval] = fsolve(fun,x0,options);
end
```

3 Create a configuration for code generation. In this case, use `'mex'`.

```
cfg = coder.config('mex');
```

4 Generate code for the `solveroot` function.

```
codegen -config cfg solveroot
```

5 Test the generated code by running the generated file, which is named `solveroot_mex.mexw64` or similar.

```
[x,fval] = solveroot_mex

x =

    0.3532
```

```
    0.6061


fval =

   1.0e-14 *

   -0.1998
   -0.1887
```

See Also
codegen | fsolve | optimoptions

More About

- "Code Generation in Nonlinear Equation Solving: Background" on page 13-36
- "Static Memory Allocation for fmincon Code Generation" on page 6-131
- "Optimization Code Generation for Real-Time Applications" on page 6-133

Parallel Computing for Optimization

What Is Parallel Computing in Optimization Toolbox?

In this section...
"Parallel Optimization Functionality" on page 14-2
"Parallel Estimation of Gradients" on page 14-2
"Nested Parallel Functions" on page 14-3

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 following Optimization Toolbox solvers can automatically distribute the numerical estimation of gradients of objective functions and nonlinear constraint functions to multiple processors:

- `fmincon`
- `fminunc`
- `fgoalattain`
- `fminimax`
- `fsolve`
- `lsqcurvefit`
- `lsqnonlin`

These solvers use parallel gradient estimation under the following conditions:

- You have a license for Parallel Computing Toolbox software.
- The option `SpecifyObjectiveGradient` is set to `false`, or, if there is a nonlinear constraint function, the option `SpecifyConstraintGradient` is set to `false`. Since `false` is the default value of these options, you don't have to set them; just don't set them both to `true`.
- 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 solver subroutine can compute in parallel automatically: 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

$$\nabla f(x) \approx \left[\frac{f(x + \Delta_1 e_1) - f(x)}{\Delta_1}, \frac{f(x + \Delta_2 e_2) - f(x)}{\Delta_2}, ..., \frac{f(x + \Delta_n e_n) - f(x)}{\Delta_n} \right],$$

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

$$\nabla f(x) \approx \left[\frac{f(x + \Delta_1 e_1) - f(x - \Delta_1 e_1)}{2\Delta_1}, ..., \frac{f(x + \Delta_n e_n) - f(x - \Delta_n e_n)}{2\Delta_n} \right].$$

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 `FiniteDifferenceType` option to `'central'`. To use forward finite differences, set the `FiniteDifferenceType` 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.

Note The documentation recommends not to use `parfor` or `parfeval` when calling Simulink; see "Using sim function within parfor" (Simulink). Therefore, you might encounter issues when optimizing a Simulink simulation in parallel using a solver's built-in parallel functionality.

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 14-2, are satisfied. "When parfor Runs In Parallel" on page 14-4 shows three cases:

1. The outermost loop is `parfor`. Only that loop runs in parallel.
2. The outermost `parfor` loop is in `fmincon`. Only `fmincon` runs in parallel.
3. The outermost `parfor` loop is in `userfcn`. `userfcn` can use `parfor` in parallel.

Bold indicates the function that runs in parallel

① **parfor (i=1:10)** ← Only the outermost parfor loop runs in parallel
    x(i) = fmincon(@userfcn,...)
    ...
end

② for (i=1:10) ← If UseParallel = true fmincon runs in parallel
    x(i) = **fmincon**(@userfcn,...)
    ...
end

③ for (i=1:10) ← If UseParallel = false userfcn can use parfor in parallel
    x(i) = fmincon(**@userfcn**,...)
    ...
end

When parfor Runs In Parallel

See Also

"Using Parallel Computing in Optimization Toolbox" on page 14-5 | "Improving Performance with Parallel Computing" on page 14-13 | "Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™" on page 14-8

Using Parallel Computing in Optimization Toolbox

In this section...
"Using Parallel Computing with Multicore Processors" on page 14-5
"Using Parallel Computing with a Multiprocessor Network" on page 14-5
"Testing Parallel Computations" on page 14-6

Using Parallel Computing with Multicore Processors

If you have a multicore processor, you can increase processing speed by 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 "Get Started with Parallel Computing Toolbox" (Parallel Computing Toolbox).

Suppose you have a dual-core processor, and want to use parallel computing. Enter this code at the command line.

```
parpool
```

MATLAB starts a pool of workers using the multicore processor. If you previously set a nondefault cluster profile, you can enforce multicore (local) computing by entering this code.

```
parpool('local')
```

Note Depending on your preferences, MATLAB can start a parallel pool automatically. To enable this feature, select **Parallel > Parallel Preferences** in the **Environment** group on the **Home** tab, and then select **Automatically create a parallel pool**.

Set solver options to use parallel computing.

```
options = optimoptions('solvername','UseParallel',true);
```

When you run an applicable solver with `options`, applicable solvers automatically use parallel computing.

To stop computing optimizations in parallel, set `UseParallel` to `false`. To halt all parallel computation, enter this code.

```
delete(gcp)
```

Note The documentation recommends not to use `parfor` or `parfeval` when calling Simulink; see "Using sim function within parfor" (Simulink). Therefore, you might encounter issues when optimizing a Simulink simulation in parallel using a solver's built-in parallel functionality.

Using Parallel Computing with a Multiprocessor Network

If you have multiple processors on a network, use Parallel Computing Toolbox functions and MATLAB Parallel Server™ software to establish parallel computation.

Make sure your system is configured properly for parallel computing. Check with your systems administrator, or refer to the Parallel Computing Toolbox documentation.

1   Perform a basic check by entering this code, where `prof` is your cluster profile.

```
parpool(prof)
```

2   Workers must be able to access your objective function file and, if applicable, your nonlinear constraint function file. Complete one of these steps to ensure access:

- Distribute the files to the workers using the `parpool AttachedFiles` argument. In this example, `objfun.m` is your objective function file, and `constrfun.m` is your nonlinear constraint function file.

```
parpool('AttachedFiles',{'objfun.m','constrfun.m'});
```

Workers access their own copies of the files.

- Give a network file path to your objective or constraint function files.

```
pctRunOnAll('addpath network_file_path')
```

Workers access the function files over the network.

3   Check whether a file is on the path of every worker.

```
pctRunOnAll('which filename')
```

If any worker does not have a path to the file, it reports

```
filename not found.
```

Set solver options to specify using parallel computing. The argument `'solvername'` represents a nonlinear solver that supports parallel evaluation.

```
options = optimoptions('solvername','UseParallel',true);
```

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`. To halt all parallel computation, enter this code.

```
delete(gcp)
```

Note The documentation recommends not to use `parfor` or `parfeval` when calling Simulink; see "Using sim function within parfor" (Simulink). Therefore, you might encounter issues when optimizing a Simulink simulation in parallel using a solver's built-in parallel functionality.

Testing Parallel Computations

Follow these steps to test whether your problem runs correctly in parallel.

1   Try your problem without parallel computation to ensure that it runs serially. Make sure this test is successful (gives correct results) before going to the next test.

2   Set `UseParallel` to `true`, and ensure that no parallel pool exists by entering `delete(gcp)`. To make sure that MATLAB does not create a parallel pool, select **Parallel > Parallel Preferences**

in the **Environment** group on the **Home** tab, and then clear **Automatically create a parallel pool**. Your problem runs `parfor` serially, with loop iterations in reverse order from a `for` loop. Make sure this test 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, this test does not increase processing speed. This testing is simply to verify the correctness of the computations.

Remember to call your solver using an `options` argument to test or use parallel functionality.

See Also

More About

- "What Is Parallel Computing in Optimization Toolbox?" on page 14-2
- "Improving Performance with Parallel Computing" on page 14-13
- "Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™" on page 14-8

Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™

This example shows 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.

```
function f = expensive_objfun(x)
%EXPENSIVE_OBJFUN An expensive objective function used in optimparfor example.

%   Copyright 2007-2013 The MathWorks, Inc.

% 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 2007-2013 The MathWorks, Inc.

% 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 time.

```
startPoint = [-1 1 1 -1];
options = optimoptions('fmincon','Display','iter','Algorithm','interior-point');
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);
```

```
                                            First-order      Norm of
 Iter F-count            f(x)   Feasibility   optimality       step
    0       5    1.839397e+00    1.500e+00    3.211e+00
```

```
 1        11      -9.760099e-01      3.708e+00      7.902e-01      2.362e+00
 2        16      -1.480976e+00      0.000e+00      8.344e-01      1.069e+00
 3        21      -2.601599e+00      0.000e+00      8.390e-01      1.218e+00
 4        29      -2.823630e+00      0.000e+00      2.598e+00      1.118e+00
 5        34      -3.905339e+00      0.000e+00      1.210e+00      7.302e-01
 6        39      -6.212992e+00      3.934e-01      7.372e-01      2.405e+00
 7        44      -5.948762e+00      0.000e+00      1.784e+00      1.905e+00
 8        49      -6.940062e+00      1.233e-02      7.668e-01      7.553e-01
 9        54      -6.973887e+00      0.000e+00      2.549e-01      3.920e-01
10        59      -7.142993e+00      0.000e+00      1.903e-01      4.735e-01
11        64      -7.155325e+00      0.000e+00      1.365e-01      2.626e-01
12        69      -7.179122e+00      0.000e+00      6.336e-02      9.115e-02
13        74      -7.180116e+00      0.000e+00      1.069e-03      4.670e-02
14        79      -7.180409e+00      0.000e+00      7.799e-04      2.815e-03
15        84      -7.180410e+00      0.000e+00      6.189e-06      3.122e-04

Local minimum found that satisfies the constraints.

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

Serial FMINCON optimization takes 17.0722 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. Pass empty matrices [] 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.

```matlab
rng default % for reproducibility
try
    gaAvailable = false;
    nvar = 4;
    gaoptions = optimoptions('ga','MaxGenerations',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     Func-count         f(x)            f(x)       Generations
    1             100           -5.546e+05       1.483e+15        0
    2             150           -5.581e+17      -1.116e+16        0
    3             200           -7.556e+17       6.679e+22        0
    4             250           -7.556e+17      -7.195e+16        1
    5             300           -9.381e+27      -1.876e+26        0
    6             350           -9.673e+27      -7.497e+26        0
    7             400           -4.511e+36      -9.403e+34        0
    8             450           -5.111e+36      -3.011e+35        0
```

```
 9              500      -7.671e+36       9.346e+37          0
10              550       -1.52e+43      -3.113e+41          0
11              600      -2.273e+45       -4.67e+43          0
12              650      -2.589e+47      -6.281e+45          0
13              700      -2.589e+47      -1.015e+46          1
14              750      -8.149e+47      -5.855e+46          0
15              800      -9.503e+47       -1.29e+47          0
Optimization terminated: maximum number of generations exceeded.
Serial GA optimization takes 80.2351 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
```

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);
```

```
                                          First-order      Norm of
 Iter F-count           f(x)  Feasibility   optimality         step
    0       5    1.839397e+00    1.500e+00    3.211e+00
    1      11   -9.760099e-01    3.708e+00    7.902e-01    2.362e+00
    2      16   -1.480976e+00    0.000e+00    8.344e-01    1.069e+00
    3      21   -2.601599e+00    0.000e+00    8.390e-01    1.218e+00
    4      29   -2.823630e+00    0.000e+00    2.598e+00    1.118e+00
    5      34   -3.905339e+00    0.000e+00    1.210e+00    7.302e-01
    6      39   -6.212992e+00    3.934e-01    7.372e-01    2.405e+00
    7      44   -5.948762e+00    0.000e+00    1.784e+00    1.905e+00
    8      49   -6.940062e+00    1.233e-02    7.668e-01    7.553e-01
    9      54   -6.973887e+00    0.000e+00    2.549e-01    3.920e-01
   10      59   -7.142993e+00    0.000e+00    1.903e-01    4.735e-01
   11      64   -7.155325e+00    0.000e+00    1.365e-01    2.626e-01
   12      69   -7.179122e+00    0.000e+00    6.336e-02    9.115e-02
   13      74   -7.180116e+00    0.000e+00    1.069e-03    4.670e-02
   14      79   -7.180409e+00    0.000e+00    7.799e-04    2.815e-03
   15      84   -7.180410e+00    0.000e+00    6.189e-06    3.122e-04
```

```
Local minimum found that satisfies the constraints.

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

Parallel FMINCON optimization takes 8.11945 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.

```matlab
rng default % to get the same evaluations as the previous run
if gaAvailable
    gaoptions = optimoptions(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
```

```
                              Best           Mean        Stall
Generation     Func-count      f(x)           f(x)     Generations
    1             100        -5.546e+05      1.483e+15       0
    2             150        -5.581e+17     -1.116e+16       0
    3             200        -7.556e+17      6.679e+22       0
    4             250        -7.556e+17     -7.195e+16       1
    5             300        -9.381e+27     -1.876e+26       0
    6             350        -9.673e+27     -7.497e+26       0
    7             400        -4.511e+36     -9.403e+34       0
    8             450        -5.111e+36     -3.011e+35       0
    9             500        -7.671e+36      9.346e+37       0
   10             550         -1.52e+43     -3.113e+41       0
   11             600        -2.273e+45      -4.67e+43       0
   12             650        -2.589e+47     -6.281e+45       0
   13             700        -2.589e+47     -1.015e+46       1
   14             750        -8.149e+47     -5.855e+46       0
   15             800        -9.503e+47      -1.29e+47       0
Optimization terminated: maximum number of generations exceeded.
Parallel GA optimization takes 15.6984 seconds.
```

Compare Serial and Parallel Time

```matlab
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];
```

14-11

```
ax.XTickLabel = {'Serial' 'Parallel'};
axis([0 1 0 ceil(max([X Y]))])
title('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.

See Also

More About

- "What Is Parallel Computing in Optimization Toolbox?" on page 14-2
- "Using Parallel Computing in Optimization Toolbox" on page 14-5
- "Improving Performance with Parallel Computing" on page 14-13

Improving Performance with Parallel Computing

In this section...
"Factors That Affect Speed" on page 14-13
"Factors That Affect Results" on page 14-13
"Searching for Global Optima" on page 14-14

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 14-3. `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 "Parallel for-Loops (parfor)" (Parallel Computing Toolbox).

- 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. Solvers can throw errors such as size mismatches.

- 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" (Global Optimization Toolbox) and "Parallel MultiStart" (Global Optimization Toolbox).

See Also

More About

- "What Is Parallel Computing in Optimization Toolbox?" on page 14-2
- "Using Parallel Computing in Optimization Toolbox" on page 14-5
- "Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™" on page 14-8

Argument and Options Reference

Function 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 the same length as the number of objectives.	`fgoalattain`
`H`	The matrix of coefficients for the quadratic terms in the quadratic equation `x'*H*x+f'*x`. H must be symmetric.	`quadprog`
`lb, ub`	Lower and upper bound vectors (or matrices). The arguments are normally the same size as `x`. However, if `lb` has fewer elements than `x`, say only `m`, then only the first `m` elements in `x` are bounded below; upper bounds in `ub` can be defined in the same manner. You can also specify unbounded variables using `-Inf` (for lower bounds) or `Inf` (for upper bounds). For example, if `lb(i) = -Inf`, the variable `x(i)` is unbounded below.	`fgoalattain`, `fmincon`, `fminimax`, `fseminf`, `linprog`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`

Argument	Description	Used by Functions
nonlcon	The function that computes the nonlinear inequality and equality constraints. "Passing Extra Parameters" on page 2-57 explains how to parametrize 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 15-6 or the individual function reference pages.	All functions
seminfcon	The function that computes the nonlinear inequality and equality constraints *and* the semi-infinite constraints. seminfcon is the name of a function file or MEX-file. "Passing Extra Parameters" on page 2-57 explains how to parametrize 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 and linprog
x1, x2	The interval over which the function is minimized.	fminbnd

See Also

More About
- "Function Output Arguments" on page 15-4

Function Output Arguments

Argument	Description	Used by Functions
attainfactor	The attainment factor at the solution x.	fgoalattain
exitflag	An integer identifying the reason the optimization algorithm terminated. See the function reference pages for descriptions of exitflag specific to each function, 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.	All functions
fval	The value of the objective function fun at the solution x.	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, linprog, quadprog
grad	The value of the gradient of fun at the solution x. 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 Output" on page 3-24.	fmincon, fminunc
jacobian	The value of the Jacobian of fun at the solution x. If fun does not compute the Jacobian, jacobian is a finite-differencing approximation of the Jacobian.	lsqcurvefit, lsqnonlin, fsolve
lambda	The Lagrange multipliers at the solution x, see "Lagrange Multiplier Structures" on page 3-22. 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

Argument	Description	Used by Functions
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 3-21. 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

See Also

More About

- "Function Input Arguments" on page 15-2

Optimization Options Reference

In this section...
"Optimization Options" on page 15-6
"Hidden Options" on page 15-17

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')`

`optimoptions` "hides" some options, meaning it does not display their values. Those options do not appear in this table. Instead, they appear in "Hidden Options" on page 15-17.

Optimization Options

Option Name	Description	Used by Functions	Restrictions
AbsoluteGapTolerance	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 AbsoluteGapTolerance: U − L <= AbsoluteGapTolerance.	intlinprog	optimoptions only
AbsoluteMaxObjectiveCount	Number of $F(x)$ to minimize the worst case absolute values.	fminimax	
Algorithm	Chooses the algorithm used by the solver.	fmincon, fminunc, fsolve, linprog, lsqcurvefit, lsqlin, lsqnonlin, quadprog	
BranchRule	Rule for choosing the component for branching: • 'maxpscost' — The fractional component with maximum pseudocost. See "Branch and Bound" on page 9-48. • 'strongpscost' — The fractional component with maximum pseudocost, with a careful estimate of pseudocost. See "Branch and Bound" on page 9-48. • 'reliability' — The fractional component with maximum pseudocost, with an even more careful estimate of pseudocost than in 'strongpscost'. See "Branch and Bound" on page 9-48. • '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

Option Name	Description	Used by Functions	Restrictions
CheckGradients	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	optimoptions only. For optimset, use DerivativeCheck
ConstraintTolerance	Tolerance on the constraint violation.	fgoalattain, fmincon, fminimax, fseminf, intlinprog, linprog, lsqlin, quadprog	optimoptions only. For optimset, use TolCon
CutGeneration	Level of cut generation (see "Cut Generation" on page 9-45): • 'none' — No cuts. Makes CutMaxIterations irrelevant. • 'basic' — Normal cut generation. • 'intermediate' — Use more cut types. • 'advanced' — Use most cut types.	intlinprog	optimoptions only
CutMaxIterations	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'.	intlinprog	optimoptions only

Option Name	Description	Used by Functions	Restrictions
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. • `'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.	All. See the individual function reference pages for the values that apply.	
EqualityGoalCount	Specify the number of objectives required for the objective `fun` to equal the set goal. Reorder your objectives, if necessary, to have `fgoalattain` achieve the first `EqualityGoalCount` objectives exactly.	fgoalattain	optimoptions only. For optimset, use GoalsExactAchieve

Option Name	Description	Used by Functions	Restrictions

`FiniteDifferenceStepSize`	Scalar or vector step size factor for finite differences. When you set `FiniteDifferenceStepSize` to a vector v, the forward finite differences `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 `FiniteDifferenceStepSize` 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`	optimoptions only. For optimset, use `FinDiffRelStep`

`FiniteDifferenceType`	Finite differences, used to estimate gradients, are either `'forward'` (the default), or `'central'` (centered), which takes twice as many function evaluations but should be more accurate. `'central'` differences might violate bounds during their evaluation in `fmincon` interior-point evaluations if the `HonorBounds` option is set to `false`.	`fgoalattain`, `fmincon`, `fminimax`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`	optimoptions only. For optimset, use `FinDiffType`
`FunctionTolerance`	Termination tolerance on the function value.	`fgoalattain`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`, `quadprog`	optimoptions only. For optimset, use `TolFun`
`HessianApproximation`	Method of Hessian approximation: `'bfgs'`, `'lbfgs'`, `{'lbfgs',Positive Integer}`, or `'finite-difference'`. Ignored when `HessianFcn` or `HessianMultiplyFcn` is nonempty.	`fmincon`	optimoptions only. For optimset, use `Hessian`
`HessianFcn`	User-supplied Hessian, specified as a function handle (see "Including Hessians" on page 2-21).	`fmincon`, `fminunc`	optimoptions only. For optimset, use `HessFcn`

Option Name	Description	Used by Functions	Restrictions
`HessianMultiplyFcn`	User-supplied Hessian multiply function, specified as a function handle. Ignored when `HessianFcn` is nonempty.	`fmincon`, `fminunc`, `quadprog`	`optimoptions` only. For `optimset`, use `HessMult`
`Heuristics`	Algorithm for searching for feasible points (see "Heuristics for Finding Feasible Solutions" on page 9-46): • `'basic'` • `'intermediate'` • `'advanced'` • `'rss'` • `'rins'` • `'round'` • `'diving'` • `'rss-diving'` • `'rins-diving'` • `'round-diving'` • `'none'`	`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 9-46.	`intlinprog`	`optimoptions` only
`HonorBounds`	The default `true` ensures that bound constraints are satisfied at every iteration. Turn off by setting to `false`.	`fmincon`	`optimoptions` only. For `optimset`, use `AlwaysHonorConstraints`
`IntegerPreprocess`	Types of integer preprocessing (see "Mixed-Integer Program Preprocessing" on page 9-44): • `'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

15-11

Option Name	Description	Used by Functions	Restrictions
`IntegerTolerance`	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. `IntegerTolerance` is not a stopping criterion.	`intlinprog`	`optimoptions` only
`JacobianMultiplyFcn`	User-defined Jacobian multiply function, specified as a function handle. Ignored unless `SpecifyObjectiveGradient` is `true` for `fsolve`, `lsqcurvefit`, and `lsqnonlin`.	`fsolve`, `lsqcurvefit`, `lsqlin`, `lsqnonlin`	
`LinearSolver`	Type of internal linear solver in algorithm: • `'auto'` — Use `'sparse'` if the passed quadratic matrix is sparse (`H` for `quadprog`, `C` for `lsqlin`), `'dense'` otherwise. • `'sparse'` — Use sparse linear algebra. • `'dense'` — Use dense linear algebra.	`lsqlin` `'interior-point'` algorithm and `quadprog` `'interior-point-convex'` algorithm	
`LPMaxIterations`	Strictly positive integer, the maximum number of simplex algorithm iterations per node during the branch-and-bound process.	`intlinprog`	`optimoptions` only
`LPOptimalityTolerance`	Nonnegative real where reduced costs must exceed `LPOptimalityTolerance` for a variable to be taken into the basis.	`intlinprog`	`optimoptions` only
`MaxFunctionEvaluations`	Maximum number of function evaluations allowed.	`fgoalattain`, `fminbnd`, `fmincon`, `fminimax`, `fminsearch`, `fminunc`, `fseminf`, `fsolve`, `lsqcurvefit`, `lsqnonlin`	`optimoptions` only. For `optimset`, use `MaxFunEvals`
`MaxIterations`	Maximum number of iterations allowed.	All but `fzero` and `lsqnonneg`	`optimoptions` only. For `optimset`, use `MaxIter`
`MaxFeasiblePoints`	Strictly positive integer. `intlinprog` stops if it finds `MaxFeasiblePoints` integer feasible points.	`intlinprog`	`optimoptions` only

Option Name	Description	Used by Functions	Restrictions
MaxNodes	Strictly positive integer that is the maximum number of nodes the solver explores in its branch-and-bound process.	intlinprog	
MaxTime	Maximum amount of time in seconds allowed for the algorithm.	intlinprog, linprog	
NodeSelection	Choose the node to explore next. • 'simplebestproj' — Best projection. See "Branch and Bound" on page 9-48. • '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 9-48.	intlinprog	optimoptions only
ObjectiveCutOff	Real greater than -Inf. The default is Inf.	intlinprog	optimoptions only
ObjectiveImprovement Threshold	Nonnegative real. intlinprog changes the current feasible solution only when it locates another with an objective function value that is at least ObjectiveImprovementThreshold lower: (fold – fnew)/(1 + |fold|) > ObjectiveImprovementThreshold.	intlinprog	optimoptions only
ObjectiveLimit	If the objective function value goes below ObjectiveLimit and the iterate is feasible, then the iterations halt.	fmincon, fminunc, lsqlin, quadprog	
OptimalityTolerance	Termination tolerance on the first-order optimality.	fgoalattain, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, linprog (interior-point only), lsqcurvefit, lsqlin, lsqnonlin, quadprog	optimoptions only. For optimset, use TolFun

Option Name	Description	Used by Functions	Restrictions
OutputFcn	Specify one or more user-defined functions that the optimization function calls at each iteration. Pass a function handle or a cell array of function handles. See "Output Function and Plot Function Syntax" on page 15-27 or "intlinprog Output Function and Plot Function Syntax" on page 15-35.	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, intlinprog, lsqcurvefit, lsqnonlin	

Option Name	Description	Used by Functions	Restrictions
PlotFcn	Plots various measures of progress while the algorithm executes. Select from predefined plots or write your own. Give the function name as listed, or as a function handle such as @optimplotx. Pass a built-in plot function name, a function handle, or a cell array of built-in names or function handles. For custom plot functions, pass function handles. • 'optimplotx' plots the current point • 'optimplotfunccount' plots the function count • 'optimplotfval' plots the function value • 'optimplotfvalconstr' plots the best feasible objective function value found as a line plot. The plot shows infeasible points as red, and feasible points as blue, using a feasibility tolerance of 1e-6. • 'optimplotconstrviolation' plots the maximum constraint violation • 'optimplotresnorm' plots the norm of the residuals • 'optimplotfirstorderopt' plots the first-order of optimality • 'optimplotstepsize' plots the step size • 'optimplotmilp' plots the gap for mixed-integer linear programs See "Plot Functions" on page 3-27 or "intlinprog Output Function and Plot Function Syntax" on page 15-35.	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, intlinprog, lsqcurvefit, lsqnonlin. See the individual function reference pages for the values that apply.	optimoptions only. For optimset, use PlotFcns

Option Name	Description	Used by Functions	Restrictions
RelativeGapTolerance	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 `RelativeGapTolerance`: `(U − L) / (abs(U) + 1) <= RelativeGapTolerance`. `intlinprog` automatically modifies the tolerance for large L magnitudes: `tolerance = min(1/(1+|L|), RelativeGapTolerance)`	intlinprog	optimoptions only
RootLPAlgorithm	Algorithm for solving linear programs: • `'dual-simplex'` — Dual simplex algorithm • `'primal-simplex'` — Primal simplex algorithm	intlinprog	optimoptions only
RootLPMaxIterations	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, `true` causes the algorithm to normalize all constraints and the objective function by their initial values. Disable by setting to the default `false`.	fmincon	
SpecifyConstraintGradient	User-defined gradients for the nonlinear constraints.	fgoalattain, fmincon, fminimax	optimoptions only. For optimset, use GradConstr
SpecifyObjectiveGradient	User-defined gradients or Jacobians for the objective functions.	fgoalattain, fmincon, fminimax, fminunc, fseminf, fsolve, lsqcurvefit, lsqnonlin	optimoptions only. For optimset, use GradObj or Jacobian
StepTolerance	Termination tolerance on x.	All functions except linprog and lsqlin	optimoptions only. For optimset, use TolX

Option Name	Description	Used by Functions	Restrictions
SubproblemAlgorithm	Determines how the iteration step is calculated.	fmincon	
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, fminunc, fsolve, lsqcurvefit, lsqnonlin.	

Hidden Options

optimoptions "hides" some options, meaning it does not display their values. To learn how to view these options, and why they are hidden, see "View Options" on page 2-66.

Function reference pages list these options in *italics.*

- "Hidden Optimization Toolbox Options" on page 15-17
- "Hidden Global Optimization Toolbox Options" on page 15-21

Hidden Optimization Toolbox Options

This table lists the hidden Optimization Toolbox options.

Options that optimoptions Hides

Option Name	Description	Used by Functions	Restrictions
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`	
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`	
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`	
InitBarrierParam	Initial barrier value.	`fmincon`	
InitDamping	Initial Levenberg-Marquardt parameter.	`fsolve`, `lsqcurvefit`, `lsqnonlin`	`optimoptions` only

Option Name	Description	Used by Functions	Restrictions
InitTrustRegionRadius	Initial radius of the trust region.	`fmincon`	
JacobPattern	Sparsity pattern of the Jacobian for finite differencing. The size of the matrix is m-by-n, where m is the number of values in the first argument returned by the user-specified function `fun`, and n is the number of elements in `x0`, the starting point.	`fsolve`, `lsqcurvefit`, `lsqnonlin`	
LPPreprocess	Type of preprocessing for the solution to the relaxed linear program (see "Linear Program Preprocessing" on page 9-44): • `'none'` — No preprocessing. • `'basic'` — Use preprocessing.	`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`	
MeritFunction	Use goal attainment/minimax merit function (multiobjective) vs. `fmincon` (single objective).	`fgoalattain`, `fminimax`	

Option Name	Description	Used by Functions	Restrictions
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`	
RelLineSrchBndDuration	Number of iterations for which the bound specified in `RelLineSrchBnd` should be active.	`fgoalattain, fmincon, fminimax, fseminf`	
ScaleProblem	When using the `Algorithm` option `'levenberg-marquardt'`, setting the `ScaleProblem` option to `'jacobian'` sometimes helps the solver on badly-scaled problems.	`fsolve, lsqcurvefit, lsqnonlin`	
TolConSQP	Constraint violation tolerance for the inner SQP iteration.	`fgoalattain, fmincon, fminimax, fseminf`	
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`	

Hidden Global Optimization Toolbox Options

For the reasons these options are hidden, see "Options that optimoptions Hides" (Global Optimization Toolbox).

Options that `optimoptions` Hides

Option Name	Used by Functions
Cache	`patternsearch`
CacheSize	`patternsearch`
CacheTol	`patternsearch`
DisplayInterval	`particleswarm`, `simulannealbnd`
FunValCheck	`particleswarm`
HybridInterval	`simulannealbnd`
InitialPenalty	`ga`, `patternsearch`
MaxMeshSize	`patternsearch`
MeshRotate	`patternsearch`
MigrationDirection	`ga`
MigrationFraction	`ga`
MigrationInterval	`ga`
PenaltyFactor	`ga`, `patternsearch`
PlotInterval	`ga`, `patternsearch`, `simulannealbnd`
StallTest	`ga`
TolBind	`patternsearch`

See Also

More About

- "Current and Legacy Option Names" on page 15-22

Current and Legacy Option Names

Many option names changed in R2016a. `optimset` uses only legacy option names. `optimoptions` accepts both legacy and current names. However, when you set an option using a legacy name-value pair, `optimoptions` displays the current equivalent value. For example, the legacy `TolX` option is equivalent to the current `StepTolerance` option.

```
options = optimoptions('fsolve','TolX',1e-4)

options =

  fsolve options:

   Options used by current Algorithm ('trust-region-dogleg'):
   (Other available algorithms: 'levenberg-marquardt', 'trust-region-reflective')

   Set properties:
               StepTolerance: 1.0000e-04

   Default properties:
                   Algorithm: 'trust-region-dogleg'
             CheckGradients: 0
                    Display: 'final'
     FiniteDifferenceStepSize: 'sqrt(eps)'
        FiniteDifferenceType: 'forward'
           FunctionTolerance: 1.0000e-06
       MaxFunctionEvaluations: '100*numberOfVariables'
                MaxIterations: 400
          OptimalityTolerance: 1.0000e-06
                    OutputFcn: []
                      PlotFcn: []
       SpecifyObjectiveGradient: 0
                     TypicalX: 'ones(numberOfVariables,1)'
                   UseParallel: 0

   Show options not used by current Algorithm ('trust-region-dogleg')
```

The following tables provide the same information. The first table lists options in alphabetical order by legacy option name, and the second table lists options in alphabetical order by current option name. The tables include only those names that differ or have different values, and show values only when they differ between legacy and current. For changes in Global Optimization Toolbox solvers, see "Options Changes in R2016a" (Global Optimization Toolbox).

Option Names in Legacy Order

Legacy Name	Current Name	Legacy Values	Current Values
AlwaysHonorConstraints	HonorBounds	'bounds', 'none'	true, false
BranchingRule	BranchRule		
CutGenMaxIter	CutMaxIterations		
DerivativeCheck	CheckGradients	'on', 'off'	true, false
FinDiffRelStep	FiniteDifferenceStepSize		
FinDiffType	FiniteDifferenceType		
GoalsExactAchieve	EqualityGoalCount		
GradConstr	SpecifyConstraintGradient	'on', 'off'	true, false
GradObj	SpecifyObjectiveGradient	'on', 'off'	true, false
Hessian	HessianApproximation	'user-supplied', 'bfgs', 'lbfgs', 'fin-diff-grads', 'on', 'off'	'bfgs', 'lbfgs', 'finite-difference' Ignored when HessianFcn or HessianMultiplyFcn is nonempty
HessFcn	HessianFcn		
HessMult	HessianMultiplyFcn		
IPPreprocess	IntegerPreprocess		
Jacobian	SpecifyObjectiveGradient		
JacobMult	JacobianMultiplyFcn		
LPMaxIter	LPMaxIterations		
MaxFunEvals	MaxFunctionEvaluations		
MaxIter	MaxIterations		
MaxNumFeasPoints	MaxFeasiblePoints		
MinAbsMax	AbsoluteMaxObjectiveCount		
PlotFcns	PlotFcn		
RelObjThreshold	ObjectiveImprovementThreshold		

Legacy Name	Current Name	Legacy Values	Current Values
RootLPMaxIter	RootLPMaxIterations		
ScaleProblem	ScaleProblem	'obj-and-constr', 'none'	true, false
TolCon	ConstraintTolerance		
TolFun (usage 1)	OptimalityTolerance		
TolFun (usage 2)	FunctionTolerance		
TolFunLP	LPOptimalityTolerance		
TolGapAbs	AbsoluteGapTolerance		
TolGapRel	RelativeGapTolerance		
TolInteger	IntegerTolerance		
TolX	StepTolerance		

Option Names in Current Order

Current Name	Legacy Name	Current Values	Legacy Values
AbsoluteGapTolerance	TolGapAbs		
AbsoluteMaxObjectiveCount	MinAbsMax		
BranchRule	BranchingRule		
CheckGradients	DerivativeCheck	true, false	'on', 'off'
ConstraintTolerance	TolCon		
CutMaxIterations	CutGenMaxIter		
EqualityGoalCount	GoalsExactAchieve		
FiniteDifferenceStepSize	FinDiffRelStep		
FiniteDifferenceType	FinDiffType		
FunctionTolerance	TolFun (usage 2)		
HessianApproximation	Hessian	'bfgs', 'lbfgs', 'finite-difference' Ignored when HessianFcn or HessianMultiplyFcn is nonempty	'user-supplied', 'bfgs', 'lbfgs', 'fin-diff-grads', 'on', 'off'
HessianFcn	HessFcn		
HessianMultiplyFcn	HessMult		
HonorBounds	AlwaysHonorConstraints	true, false	'bounds', 'none'
IntegerPreprocess	IPPreprocess		
IntegerTolerance	TolInteger		
JacobianMultiplyFcn	JacobMult		
LPMaxIterations	LPMaxIter		
LPOptimalityTolerance	TolFunLP		
MaxFeasiblePoints	MaxNumFeasPoints		
MaxFunctionEvaluations	MaxFunEvals		
MaxIterations	MaxIter		
ObjectiveImprovementThreshold	RelObjThreshold		

Current Name	Legacy Name	Current Values	Legacy Values
OptimalityTolerance	TolFun (usage 1)		
PlotFcn	PlotFcns		
RelativeGapTolerance	TolGapRel		
RootLPMaxIterations	RootLPMaxIter		
ScaleProblem	ScaleProblem	true, false	'obj-and-constr', 'none'
SpecifyConstraintGradient	GradConstr	true, false	'on', 'off'
SpecifyObjectiveGradient	GradObj or Jacobian	true, false	'on', 'off'
StepTolerance	TolX		

See Also

More About

•   "Optimization Options Reference" on page 15-6

Output Function and Plot Function Syntax

In this section...
"What Are Output Functions and Plot Functions?" on page 15-27
"Structure of the Output Function or Plot Function" on page 15-28
"Fields in optimValues" on page 15-28
"States of the Algorithm" on page 15-33
"Stop Flag" on page 15-33

What Are Output Functions and Plot Functions?

For examples of output functions and plot functions, see "Output Functions for Optimization Toolbox™" on page 3-30 and "Plot Functions" on page 3-27.

The `OutputFcn` option 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. You can also halt the execution of a solver according to conditions you set; see "Structure of the Output Function or Plot Function" on page 15-28.

Similarly, the `PlotFcn` option specifies one or more functions that an optimization function calls at each iteration, and can halt the solver. The difference between a plot function and an output function is twofold:

- Predefined plot functions exist for most solvers, enabling you to obtain typical plots easily.
- A plot function sends output to a window having **Pause** and **Stop** buttons, enabling you to halt the solver early without losing information.

Caution `intlinprog` output functions and plot functions differ from those in other solvers. See "intlinprog Output Function and Plot Function Syntax" on page 15-35.

To set up an output function or plot function, do the following:

1. Write the function as a function file or local function.
2. Use `optimoptions` to set the value of `OutputFcn` or `PlotFcn` 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 or plot function, use the syntax

   ```
   options = optimoptions('solvername','OutputFcn',{@outfun, @outfun2});
   ```

   To use tab-completion to help select a built-in plot function name, use quotes rather than a function handle.

```
optimplotconstrviolation
optimplotfirstorderopt
optimplotfunccount
optimplotfval
optimplotfvalconstr
optimplotstepsize
optimplotx
```

```
options = optimoptions('fmincon','PlotFcn','optimplot
```

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

"Passing Extra Parameters" on page 2-57 explains how to pass parameters or data to your output function or plot function, if necessary.

Structure of the Output Function or Plot Function

The function definition line of the output function or plot 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 15-28 describes the structure in detail.
- `state` is the current state of the algorithm. "States of the Algorithm" on page 15-33 lists the possible values.
- `stop` is a flag that is `true` or `false` depending on whether the optimization routine should stop (`true`) or continue (`false`). For details, see "Stop Flag" on page 15-33.

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 3-14. 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 8-3.	`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 3-14.
`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 3-14.
`degenerate`	Measure of degeneracy. A point is *degenerate* if: • The partial derivative with respect to one of the variables is 0 at the point, and • A bound constraint is active for that variable at the point. See "Degeneracy" on page 15-33.	`fmincon` (TRR), `lsqcurvefit` (TRR), `lsqnonlin` (TRR)	None
`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 3-14.
`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 3-14.
`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 3-14.

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
`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 3-14.
`gradient`	Current gradient of objective function — either analytic gradient if you provide it or finite-differencing approximation. Final value equals optimization function output `grad`.	`fgoalattain, fmincon, fminimax, fminunc, fseminf, fsolve, lsqcurvefit, lsqnonlin`	None
`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 3-14.
`lambda`	The Levenberg-Marquardt parameter, `lambda`, at the current iteration. See "Levenberg-Marquardt Method" on page 12-6.	`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 3-14.
`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

15-31

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
`procedure`	Procedure messages.	`fgoalattain, fminbnd, fmincon` (AS)`, fminimax, fminsearch, fseminf, fzero` `fmincon` TRR and SQP, and `fminunc` provide blank field values.	`Procedure` See "Iterative Display" on page 3-14.
`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 3-14.
`resnorm`	2-norm of the residual squared.	`lsqcurvefit, lsqnonlin`	`Resnorm` See "Iterative Display" on page 3-14.
`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
`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 3-14.
`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 3-14.

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 `'interrupt'` state occurs only in the `fmincon 'active-set'` algorithm and the `fgoalattain`, `fminimax`, and `fseminf` solvers. There, the state can occur before a quadratic programming subproblem solution or a line search.

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

```matlab
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 stop (`true`) or continue (`false`). The following examples show typical ways to use the `stop` flag.

Stopping an Optimization Based on Data in optimValues

The output function or plot function can stop an optimization at any iteration based on the current data in `optimValues`. For example, the following code sets `stop` to `true`, stopping the optimization, when the size of the directional derivative is less than `.01`:

```
function stop = outfun(x,optimValues,state)
stop = false;
% Check whether directional derivative norm is less than .01.
if norm(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');
```

See Also

More About

- "Output Functions for Optimization Toolbox™" on page 3-30
- "Plot Functions" on page 3-27

intlinprog Output Function and Plot Function Syntax

In this section...
"What Are Output Functions and Plot Functions?" on page 15-35
"Custom Function Syntax" on page 15-35
"optimValues Structure" on page 15-36

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, completing a heuristic successfully, finding a new integer feasible solution during branch-and-bound, 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 and Plot Function Syntax" on page 15-27 and "Plot Functions" on page 3-27.

- There is one built-in output function: `savemilpsolutions`. This function collects the integer feasible points that the algorithm finds at event times. 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: Solver-Based" on page 9-57.

Call output functions or plot functions by passing the `OutputFcn` or `PlotFcn` name-value pairs, including the handle to the output function or plot function. For example,

```
options = optimoptions(@intlinprog,'OutputFcn',@savemilpsolutions,'PlotFcn',@optimplotmilp);
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 an empty matrix `[]` 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 15-36.
- state — 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.
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. The relative gap is a percentage from 0 to 100, exactly as in the output argument. 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

coneprog

Second-order cone programming solver

Syntax

```
x = coneprog(f,socConstraints)
x = coneprog(f,socConstraints,A,b,Aeq,beq)
x = coneprog(f,socConstraints,A,b,Aeq,beq,lb,ub)
x = coneprog(f,socConstraints,A,b,Aeq,beq,lb,ub,options)
x = coneprog(problem)
[x,fval] = coneprog( ___ )
[x,fval,exitflag,output] = coneprog( ___ )
[x,fval,exitflag,output,lambda] = coneprog( ___ )
```

Description

The `coneprog` function is a second-order cone programming solver that finds the minimum of a problem specified by

$$\min_{x} f^{T} x$$

subject to the constraints

$$\|A_{sc}(i) \cdot x - b_{sc}(i)\| \le d_{sc}^{T}(i) \cdot x - \gamma(i)$$
$$A \cdot x \le b$$
$$\text{Aeq} \cdot x = \text{beq}$$
$$\text{lb} \le x \le \text{ub}.$$

f, x, b, beq, lb, and *ub* are vectors, and *A* and *Aeq* are matrices. For each *i*, the matrix $A_{sc}(i)$, vectors $d_{sc}(i)$ and $b_{sc}(i)$, and scalar $\gamma(i)$ are in a second-order cone constraint that you create using `secondordercone`.

For more details about cone constraints, see "Second-Order Cone Constraint" on page 16-14.

`x = coneprog(f,socConstraints)` solves the second-order cone programming problem with the constraints in `socConstraints` encoded as

- $A_{sc}(i)$ = `socConstraints.A(i)`
- $b_{sc}(i)$ = `socConstraints.b(i)`
- $d_{sc}(i)$ = `socConstraints.d(i)`
- $\gamma(i)$ = `socConstraints.gamma(i)`

`x = coneprog(f,socConstraints,A,b,Aeq,beq)` solves the problem subject to the inequality constraints A*x ≤ b and equality constraints Aeq*x = beq. Set A = [] and b = [] if no inequalities exist.

x = coneprog(f,socConstraints,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 = coneprog(f,socConstraints,A,b,Aeq,beq,lb,ub,options) minimizes using the optimization options specified by options. Use optimoptions to set these options.

x = coneprog(problem) finds the minimum for problem, a structure described in problem.

[x,fval] = coneprog(___) also returns the objective function value at the solution fval = f'*x, using any of the input argument combinations in previous syntaxes.

[x,fval,exitflag,output] = coneprog(___) 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] = coneprog(___) additionally returns a structure lambda whose fields contain the dual variables at the solution x.

Examples

Single Cone Constraint

To set up a problem with a second-order cone constraint, create a second-order cone constraint object.

```
A = diag([1,1/2,0]);
b = zeros(3,1);
d = [0;0;1];
gamma = 0;
socConstraints = secondordercone(A,b,d,gamma);
```

Create an objective function vector.

```
f = [-1,-2,0];
```

The problem has no linear constraints. Create empty matrices for these constraints.

```
Aineq = [];
bineq = [];
Aeq = [];
beq = [];
```

Set upper and lower bounds on x(3).

```
lb = [-Inf,-Inf,0];
ub = [Inf,Inf,2];
```

Solve the problem by using the coneprog function.

```
[x,fval] = coneprog(f,socConstraints,Aineq,bineq,Aeq,beq,lb,ub)
```

Optimal solution found.

x = *3×1*

```
    0.4851
    3.8806
    2.0000
```

```
fval = -8.2462
```

The solution component `x(3)` is at its upper bound. The cone constraint is active at the solution:

```
norm(A*x-b) - d'*x % Near 0 when the constraint is active
```

```
ans = -2.5677e-08
```

Several Cone Constraints

To set up a problem with several second-order cone constraints, create an array of constraint objects. To save time and memory, create the highest-index constraint first.

```
A = diag([1,2,0]);
b = zeros(3,1);
d = [0;0;1];
gamma = -1;
socConstraints(3) = secondordercone(A,b,d,gamma);

A = diag([3,0,1]);
d = [0;1;0];
socConstraints(2) = secondordercone(A,b,d,gamma);

A = diag([0;1/2;1/2]);
d = [1;0;0];
socConstraints(1) = secondordercone(A,b,d,gamma);
```

Create the linear objective function vector.

```
f = [-1;-2;-4];
```

Solve the problem by using the `coneprog` function.

```
[x,fval] = coneprog(f,socConstraints)
```

```
Optimal solution found.
```

```
x = 3×1

    0.4238
    1.6477
    2.3225
```

```
fval = -13.0089
```

Cone Programming with Linear Constraints

Specify an objective function vector and a single second-order cone constraint.

```
f = [-4;-9;-2];
Asc = diag([1,4,0]);
b = [0;0;0];
d = [0;0;1];
gamma = 0;
socConstraints = secondordercone(Asc,b,d,gamma);
```

Specify a linear inequality constraint.

```
A = [1/4,1/9,1];
b = 5;
```

Solve the problem.

```
[x,fval] = coneprog(f,socConstraints,A,b)
```

```
Optimal solution found.

x = 3×1

    3.2304
    0.6398
    4.1213


fval = -26.9225
```

Cone Programming with Nondefault Options

To observe the iterations of the `coneprog` solver, set the `Display` option to `'iter'`.

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

Create a second-order cone programming problem and solve it using `options`.

```
Asc = diag([1,1/2,0]);
b = zeros(3,1);
d = [0;0;1];
gamma = 0;
socConstraints = secondordercone(Asc,b,d,gamma);
f = [-1,-2,0];
Aineq = [];
bineq = [];
Aeq = [];
beq = [];
lb = [-Inf,-Inf,0];
ub = [Inf,Inf,2];
[x,fval] = coneprog(f,socConstraints,Aineq,bineq,Aeq,beq,lb,ub,options)
```

```
Iter           Fval  Primal Infeas    Dual Infeas    Duality Gap    Time
   1   0.000000e+00   0.000000e+00   5.714286e-01   1.250000e-01    0.04
   2  -7.558066e+00   0.000000e+00   7.151114e-02   1.564306e-02    0.16
   3  -7.366973e+00   0.000000e+00   1.075440e-02   2.352525e-03    0.19
   4  -8.243432e+00   0.000000e+00   5.191882e-05   1.135724e-05    0.22
   5  -8.246067e+00   0.000000e+00   2.430813e-06   5.317403e-07    0.36
   6  -8.246211e+00   1.387779e-17   6.154504e-09   1.346298e-09    0.39
Optimal solution found.
```

```
x = 3×1

    0.4851
    3.8806
    2.0000


fval = -8.2462
```

Cone Programming with Problem Structure

Create the elements of a second-order cone programming problem. To save time and memory, create the highest-index constraint first.

```
A = diag([1,2,0]);
b = zeros(3,1);
d = [0;0;1];
gamma = -1;
socConstraints(3) = secondordercone(A,b,d,gamma);
A = diag([3,0,1]);
d = [0;1;0];
socConstraints(2) = secondordercone(A,b,d,gamma);
A = diag([0;1/2;1/2]);
d = [1;0;0];
socConstraints(1) = secondordercone(A,b,d,gamma);
f = [-1;-2;-4];
options = optimoptions('coneprog','Display','iter');
```

Create a problem structure with the required fields, as described in "problem" on page 16-0 .

```
problem = struct('f',f,...
    'socConstraints',socConstraints,...
    'Aineq',[],'bineq',[],...
    'Aeq',[],'beq',[],...
    'lb',[],'ub',[],...
    'solver','coneprog',...
    'options',options);
```

Solve the problem by calling `coneprog`.

```
[x,fval] = coneprog(problem)
```

```
Iter          Fval    Primal Infeas    Dual Infeas    Duality Gap    Time
   1    0.000000e+00    0.000000e+00    5.333333e-01    5.555556e-02    0.35
   2   -9.696012e+00    1.850372e-17    7.631901e-02    7.949897e-03    0.41
   3   -1.178942e+01    9.251859e-18    1.261803e-02    1.314378e-03    0.43
   4   -1.294426e+01    0.000000e+00    1.683078e-03    1.753206e-04    0.44
   5   -1.295217e+01    9.251859e-18    8.994595e-04    9.369370e-05    0.45
   6   -1.295331e+01    9.251859e-18    4.748841e-04    4.946709e-05    0.46
   7   -1.300753e+01    1.850372e-17    2.799942e-05    2.916606e-06    0.47
   8   -1.300671e+01    0.000000e+00    2.366136e-05    2.464725e-06    0.64
   9   -1.300850e+01    9.251859e-18    8.197001e-06    8.538542e-07    0.67
  10   -1.300843e+01    1.850372e-17    7.365088e-06    7.671967e-07    0.69
  11   -1.300859e+01    9.251859e-18    2.866457e-06    2.985893e-07    0.70
  12   -1.300891e+01    1.850372e-17    2.029507e-07    2.114069e-08    0.73
Optimal solution found.
```

```
x = 3×1

    0.4238
    1.6477
    2.3225


fval = -13.0089
```

Examine coneprog Solution Process

Create a second-order cone programming problem. To save time and memory, create the highest-index constraint first.

```
A = diag([1,2,0]);
b = zeros(3,1);
d = [0;0;1];
gamma = -1;
socConstraints(3) = secondordercone(A,b,d,gamma);
A = diag([3,0,1]);
d = [0;1;0];
socConstraints(2) = secondordercone(A,b,d,gamma);
A = diag([0;1/2;1/2]);
d = [1;0;0];
socConstraints(1) = secondordercone(A,b,d,gamma);
f = [-1;-2;-4];
options = optimoptions('coneprog','Display','iter');
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
```

Solve the problem, requesting information about the solution process.

```
[x,fval,exitflag,output] = coneprog(f,socConstraints,A,b,Aeq,beq,lb,ub,options)
```

```
Iter          Fval    Primal Infeas    Dual Infeas    Duality Gap    Time
   1   0.000000e+00   0.000000e+00   5.333333e-01   5.555556e-02   0.05
   2  -9.696012e+00   1.850372e-17   7.631901e-02   7.949897e-03   0.11
   3  -1.178942e+01   9.251859e-18   1.261803e-02   1.314378e-03   0.12
   4  -1.294426e+01   0.000000e+00   1.683078e-03   1.753206e-04   0.13
   5  -1.295217e+01   9.251859e-18   8.994595e-04   9.369370e-05   0.15
   6  -1.295331e+01   9.251859e-18   4.748841e-04   4.946709e-05   0.16
   7  -1.300753e+01   1.850372e-17   2.799942e-05   2.916606e-06   0.18
   8  -1.300671e+01   0.000000e+00   2.366136e-05   2.464725e-06   0.33
   9  -1.300850e+01   9.251859e-18   8.197001e-06   8.538542e-07   0.34
  10  -1.300843e+01   1.850372e-17   7.365088e-06   7.671967e-07   0.35
  11  -1.300859e+01   9.251859e-18   2.866457e-06   2.985893e-07   0.36
  12  -1.300891e+01   1.850372e-17   2.029507e-07   2.114069e-08   0.37
Optimal solution found.

x = 3×1

    0.4238
```

```
        1.6477
        2.3225


fval = -13.0089

exitflag = 1

output = struct with fields:
            iterations: 12
      primalfeasibility: 1.8504e-17
        dualfeasibility: 2.0295e-07
             dualitygap: 2.1141e-08
              algorithm: 'interior-point'
                message: 'Optimal solution found.'
```

- Both the iterative display and the output structure show that `coneprog` used 12 iterations to arrive at the solution.
- The exit flag value `1` and the `output.message` value `'Optimal solution found.'` indicate that the solution is reliable.
- The `output` structure shows that the infeasibilities tend to decrease through the solution process, as does the duality gap.
- You can reproduce the `fval` output by multiplying `f'*x`.

```
f'*x

ans = -13.0089
```

Obtain coneprog Dual Variables

Create a second-order cone programming problem. To save time and memory, create the highest-index constraint first.

```
A = diag([1,2,0]);
b = zeros(3,1);
d = [0;0;1];
gamma = -1;
socConstraints(3) = secondordercone(A,b,d,gamma);
A = diag([3,0,1]);
d = [0;1;0];
socConstraints(2) = secondordercone(A,b,d,gamma);
A = diag([0;1/2;1/2]);
d = [1;0;0];
socConstraints(1) = secondordercone(A,b,d,gamma);
f = [-1;-2;-4];
```

Solve the problem, requesting dual variables at the solution along with all other `coneprog` output..

```
[x,fval,exitflag,output,lambda] = coneprog(f,socConstraints);

Optimal solution found.
```

Examine the returned `lambda` structure. Because the only problem constraints are cone constraints, examine only the `soc` field in the `lambda` structure.

```
disp(lambda.soc)

   1.0e-05 *

   0.1262
   0.4289
   0.0302
```

The constraints have nonzero dual values, indicating the constraints are active at the solution.

Input Arguments

f — Coefficient vector
real vector | real array

Coefficient vector, specified as a real vector or real array. The coefficient vector represents the objective function `f'*x`. The notation assumes that `f` is a column vector, but you can use a row vector or array. Internally, `coneprog` converts `f` to the column vector `f(:)`.

Example: `f = [1,3,5,-6]`

Data Types: `double`

socConstraints — Second-order cone constraints
vector of `SecondOrderConeConstraint` objects

Second-order cone constraints, specified as vector of `SecondOrderConeConstraint` objects. Create these objects using the `secondordercone` function.

`socConstraints` encodes the constraints

$$\|A_{sc}(i) \cdot x - b_{sc}(i)\| \le d_{sc}^T(i) \cdot x - \gamma(i)$$

where the mapping between the array and the equation is as follows:

- $A_{sc}(i) = $ `socConstraints.A(i)`
- $b_{sc}(i) = $ `socConstraints.b(i)`
- $d_{sc}(i) = $ `socConstraints.d(i)`
- $\gamma(i) = $ `socConstraints.gamma(i)`

Example: `Asc = diag([1 1/2 0]); bsc = zeros(3,1); dsc = [0;0;1]; gamma = -1; socConstraints = secondordercone(Asc,bsc,dsc,gamma);`

Data Types: `struct`

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 (length of `f`). 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, consider these inequalities:

$x_1 + 2x_2 \le 10$
$3x_1 + 4x_2 \le 20$
$5x_1 + 6x_2 \le 30$.

Specify the inequalities by entering the following 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, consider these inequalities:

$x_1 + 2x_2 \le 10$
$3x_1 + 4x_2 \le 20$
$5x_1 + 6x_2 \le 30$.

Specify the inequalities by entering the following constraints.

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x components sum to 1 or less, use 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 (length of f). 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, consider these equalities:

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20.$

Specify the equalities by entering the following 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 `Me`-by-`N`.

For example, consider these equalities:

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20.$

Specify the equalities by entering the following constraints.

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use `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 length of `f` is equal to the length of `lb`, then `lb` specifies that

`x(i) >= lb(i)` for all `i`.

If `numel(lb) < numel(f)`, 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, use `lb = zeros(size(f))`.

Data Types: `double`

ub — Upper bounds
real vector | real array

Upper bounds, specified as a real vector or real array. If the length of f is equal to the length of ub, then ub specifies that

x(i) <= ub(i) for all i.

If numel(ub) < numel(f), 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 1, use ub = ones(size(f)).

Data Types: double

options — Optimization options
output of optimoptions

Optimization options, specified as the output of optimoptions.

Option	Description
ConstraintTolerance	Feasibility tolerance for constraints, a scalar from 0 through 1. Constrai... primal feasibility tolerance. The default is 1e-6.
Display	Level of display (see "Iterative Display" on page 3-14): • 'final' (default) displays only the final output. • 'iter' displays output at each iteration. • 'off' or 'none' displays no output.
MaxIterations	Maximum number of iterations allowed, a positive integer. The default is 2... See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and ... page 3-9.
MaxTime	Maximum amount of time in seconds that the algorithm runs, a positive n... default is Inf, which disables this stopping criterion.
OptimalityTolerance	Termination tolerance on the dual feasibility, a positive scalar. The default...

Example: optimoptions('coneprog','Display','iter','MaxIterations',100)

problem — Problem structure
structure

Problem structure, specified as a structure with the following fields.

Field Name	Entry
f	Linear objective function vector f
socConstraints	Structure array of second-order cone constraints
Aineq	Matrix of linear inequality constraints
bineq	Vector of linear inequality constraints

Field Name	Entry
Aeq	Matrix of linear equality constraints
beq	Vector of linear equality constraints
lb	Vector of lower bounds
ub	Vector of upper bounds
solver	'coneprog'
options	Options created with optimoptions

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. The x output is empty when the exitflag value is –2, –3, or –10.

fval — Objective function value at the solution
real number

Objective function value at the solution, returned as a real number. Generally, fval = f'*x. The fval output is empty when the exitflag value is –2, –3, or –10.

exitflag — Reason coneprog stopped
integer

Reason coneprog stopped, returned as an integer.

Value	Description
1	The function converged to a solution x.
0	The number of iterations exceeded options.MaxIterations, or the solution tim options.MaxTime.
-2	No feasible point was found.
-3	The problem is unbounded.
-7	The search direction became too small. No further progress could be made.
-10	The problem is numerically unstable.

output — Information about optimization process
structure

Information about the optimization process, returned as a structure with these fields.

Field	Description
algorithm	Optimization algorithm used
dualfeasibility	Maximum of dual constraint violations

Field	Description
dualitygap	Duality gap
iterations	Number of iterations
message	Exit message
primalfeasibility	Maximum of constraint violations

The output fields dualfeasibility, dualitygap, and primalfeasibility are empty when the exitflag value is –2, –3, or –10.

lambda — Dual variables at the solution
structure

Dual variables at the solution, returned as a structure with these fields.

Field	Description
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
soc	Second-order cone constraints corresponding to socConstrai

lambda fields are empty when the exitflag value is –2, –3, or –10.

The Lagrange multipliers (dual variables) are part of the following Lagrangian, which is stationary (zero gradient) at a solution:

$$f^T x + \sum_i \lambda_{\text{soc}}(i)\left(d_{\text{soc}}^T(i)x - \text{gamma}(i) - \|A_{\text{soc}}(i)x - b_{\text{soc}}(i)\|\right)$$

$$+ \lambda_{\text{ineqlin}}^T(b - Ax) + \lambda_{\text{eqlin}}^T(\text{Aeq}\,x - \text{beq}) + \lambda_{\text{upper}}^T(\text{ub}-x) + \lambda_{\text{lower}}^T(x - \text{lb}).$$

The inequality terms that multiply the lambda fields are nonnegative.

More About

Second-Order Cone Constraint

Why is the constraint

$$\|A \cdot x - b\| \le d^T \cdot x - \gamma$$

called a second-order cone constraint? Consider a cone in 3-D space with elliptical cross-sections in the x-y plane, and a diameter proportional to the z coordinate. The y coordinate has scale ½, and the x coordinate has scale 1. The inequality defining the inside of this cone with its point at [0,0,0] is

$$\sqrt{x^2 + \frac{y^2}{4}} \le z.$$

In the coneprog syntax, this cone has the following arguments.

```
A = diag([1 1/2 0]);
b = [0;0;0];
d = [0;0;1];
gamma = 0;
```

Plot the boundary of the cone.

```
[X,Y] = meshgrid(-2:0.1:2);
Z = sqrt(X.^2 + Y.^2/4);
surf(X,Y,Z)
view(8,2)
xlabel 'x'
ylabel 'y'
zlabel 'z'
```
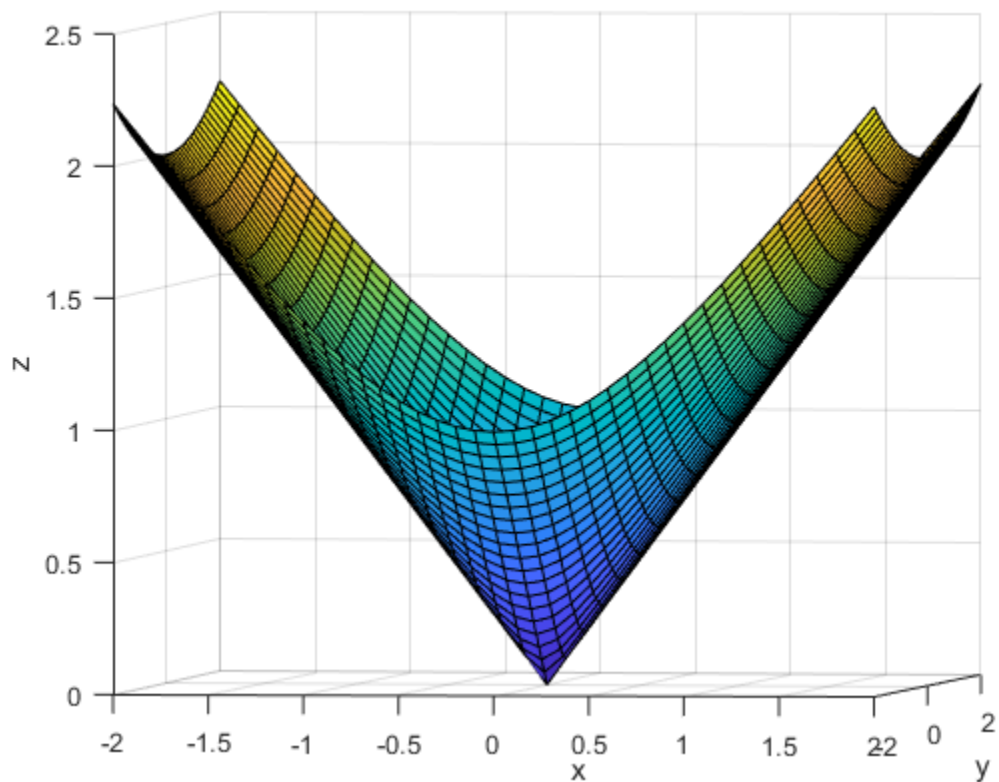

The b and gamma arguments move the cone. The A and d arguments rotate the cone and change its shape.

Algorithms

The algorithm uses an interior-point method. For details, see "Second-Order Cone Programming Algorithm" on page 11-15.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `coneprog`.

See Also
Optimize | SecondOrderConeConstraint | linprog | quadprog | secondordercone

Topics
"Minimize Energy of Piecewise Linear Mass-Spring System Using Cone Programming" on page 11-72
"Convert Quadratic Constraints to Second-Order Cone Constraints" on page 11-66
"Convert Quadratic Programming Problem to Second-Order Cone Program" on page 11-68

Introduced in R2020b

EquationProblem

System of nonlinear equations

Description

Specify a system of equations using optimization variables, and solve the system using `solve`.

Tip For the full workflow, see "Problem-Based Workflow for Solving Equations" on page 10-4.

Creation

Create an `EquationProblem` object by using the `eqnproblem` function. Add equations to the problem by creating `OptimizationEquality` objects and setting them as `Equations` properties of the `EquationProblem` object.

```
prob = eqnproblem;
x = optimvar('x');
eqn = x^5 - x^4 + 3*x == 1/2;
prob.Equations.eqn = eqn;
```

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

Properties

Equations — Problem equations
[] (default) | OptimizationEquality array | structure with OptimizationEquality arrays as fields

Problem equations, specified as an `OptimizationEquality` array or structure with `OptimizationEquality` arrays as fields.

Example: `sum(x.^2,2) == 4`

Description — Problem label
'' (default) | string | character vector

Problem label, specified as a string or character vector. The software does not use `Description` for computation. `Description` is an arbitrary label that you can use for any reason. For example, you can share, archive, or present a model or problem, and store descriptive information about the model or problem in `Description`.

Example: `"An iterative approach to the Traveling Salesman problem"`

Data Types: `char` | `string`

`Variables` — Optimization variables in object
structure of `OptimizationVariable` objects

This property is read-only.

Optimization variables in the object, specified as a structure of `OptimizationVariable` objects.

Data Types: `struct`

Object Functions

optimoptions	Create optimization options
prob2struct	Convert optimization problem or equation problem to solver form
show	Display information about optimization object
solve	Solve optimization problem or equation problem
varindex	Map problem variables to solver-based variable index
write	Save optimization object description

Examples

Solve Nonlinear System of Equations, Problem-Based

To solve the nonlinear system of equations

$$\exp(-\exp(-(x_1 + x_2))) = x_2\left(1 + x_1^2\right)$$

$$x_1\cos(x_2) + x_2\sin(x_1) = \frac{1}{2}$$

using the problem-based approach, first define `x` as a two-element optimization variable.

```
x = optimvar('x',2);
```

Create the first equation as an optimization equality expression.

```
eq1 = exp(-exp(-(x(1) + x(2)))) == x(2)*(1 + x(1)^2);
```

Similarly, create the second equation as an optimization equality expression.

```
eq2 = x(1)*cos(x(2)) + x(2)*sin(x(1)) == 1/2;
```

Create an equation problem, and place the equations in the problem.

```
prob = eqnproblem;
prob.Equations.eq1 = eq1;
prob.Equations.eq2 = eq2;
```

Review the problem.

```
show(prob)
  EquationProblem :

    Solve for:
       x
```

```
   eq1:
        exp(-exp(-(x(1) + x(2)))) == (x(2) .* (1 + x(1).^2))

   eq2:
        ((x(1) .* cos(x(2))) + (x(2) .* sin(x(1)))) == 0.5
```

Solve the problem starting from the point `[0,0]`. For the problem-based approach, specify the initial point as a structure, with the variable names as the fields of the structure. For this problem, there is only one variable, `x`.

```
x0.x = [0 0];
[sol,fval,exitflag] = solve(prob,x0)
```

```
Solving problem using fsolve.

Equation solved.

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

sol = struct with fields:
    x: [2x1 double]


fval = struct with fields:
    eq1: -2.4069e-07
    eq2: -3.8253e-08


exitflag =
    EquationSolved
```

View the solution point.

```
disp(sol.x)

    0.3532
    0.6061
```

Unsupported Functions Require `fcn2optimexpr`

If your equation functions are not composed of elementary functions, you must convert the functions to optimization expressions using `fcn2optimexpr`. For the present example:

```
ls1 = fcn2optimexpr(@(x)exp(-exp(-(x(1)+x(2)))),x);
eq1 = ls1 == x(2)*(1 + x(1)^2);
ls2 = fcn2optimexpr(@(x)x(1)*cos(x(2))+x(2)*sin(x(1)),x);
eq2 = ls2 == 1/2;
```

See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8.

See Also

`OptimizationEquality` | `eqnproblem` | `fcn2optimexpr` | `optimvar` | `show` | `write`

Topics
"Systems of Nonlinear Equations"
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2019b

eqnproblem

Create equation problem

Syntax

```
prob = eqnproblem
prob = eqnproblem(Name,Value)
```

Description

Use `eqnproblem` to create an equation problem.

Tip For the full workflow, see "Problem-Based Workflow for Solving Equations" on page 10-4.

`prob = eqnproblem` creates an equation problem with default properties.

`prob = eqnproblem(Name,Value)` specifies additional options using one or more name-value pair arguments. For example, you can specify equations when constructing the problem by using the `Equations` name.

Examples

Solve Nonlinear System of Equations, Problem-Based

To solve the nonlinear system of equations

$$\exp(-\exp(-(x_1 + x_2))) = x_2\left(1 + x_1^2\right)$$

$$x_1\cos(x_2) + x_2\sin(x_1) = \frac{1}{2}$$

using the problem-based approach, first define `x` as a two-element optimization variable.

```
x = optimvar('x',2);
```

Create the first equation as an optimization equality expression.

```
eq1 = exp(-exp(-(x(1) + x(2)))) == x(2)*(1 + x(1)^2);
```

Similarly, create the second equation as an optimization equality expression.

```
eq2 = x(1)*cos(x(2)) + x(2)*sin(x(1)) == 1/2;
```

Create an equation problem, and place the equations in the problem.

```
prob = eqnproblem;
prob.Equations.eq1 = eq1;
prob.Equations.eq2 = eq2;
```

Review the problem.

```
show(prob)

  EquationProblem :

    Solve for:
       x

 eq1:
       exp(-exp(-(x(1) + x(2)))) == (x(2) .* (1 + x(1).^2))

 eq2:
       ((x(1) .* cos(x(2))) + (x(2) .* sin(x(1)))) == 0.5
```

Solve the problem starting from the point `[0,0]`. For the problem-based approach, specify the initial point as a structure, with the variable names as the fields of the structure. For this problem, there is only one variable, `x`.

```
x0.x = [0 0];
[sol,fval,exitflag] = solve(prob,x0)

Solving problem using fsolve.

Equation solved.

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

sol = struct with fields:
    x: [2x1 double]


fval = struct with fields:
    eq1: -2.4069e-07
    eq2: -3.8253e-08


exitflag =
    EquationSolved
```

View the solution point.

```
disp(sol.x)

    0.3532
    0.6061
```

Unsupported Functions Require `fcn2optimexpr`

If your equation functions are not composed of elementary functions, you must convert the functions to optimization expressions using `fcn2optimexpr`. For the present example:

```
ls1 = fcn2optimexpr(@(x)exp(-exp(-(x(1)+x(2)))),x);
eq1 = ls1 == x(2)*(1 + x(1)^2);
ls2 = fcn2optimexpr(@(x)x(1)*cos(x(2))+x(2)*sin(x(1)),x);
eq2 = ls2 == 1/2;
```

See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8.

Solve Nonlinear System of Polynomials, Problem-Based

When x is a 2-by-2 matrix, the equation

$$x^3 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

is a system of polynomial equations. Here, x^3 means $x*x*x$ using matrix multiplication. You can easily formulate and solve this system using the problem-based approach.

First, define the variable x as a 2-by-2 matrix variable.

```
x = optimvar('x',2,2);
```

Define the equation to be solved in terms of x.

```
eqn = x^3 == [1 2;3 4];
```

Create an equation problem with this equation.

```
prob = eqnproblem('Equations',eqn);
```

Solve the problem starting from the point [1 1;1 1].

```
x0.x = ones(2);
sol = solve(prob,x0)
```

```
Solving problem using fsolve.

Equation solved.

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

sol = struct with fields:
    x: [2x2 double]
```

Examine the solution.

```
disp(sol.x)

   -0.1291    0.8602
    1.2903    1.1612
```

Display the cube of the solution.

```
sol.x^3

ans = 2×2

    1.0000    2.0000
```

```
   3.0000    4.0000
```

Input Arguments

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `prob = eqnproblem('Equations',eqn)`

Equations — Problem equations
`[]` (default) | `OptimizationEquality` array | structure with `OptimizationEquality` arrays as fields

Problem equations, specified as an `OptimizationEquality` array or structure with `OptimizationEquality` arrays as fields.

Example: `sum(x.^2,2) == 4`

Description — Problem label
`''` (default) | string | character vector

Problem label, specified as a string or character vector. The software does not use `Description` for computation. `Description` is an arbitrary label that you can use for any reason. For example, you can share, archive, or present a model or problem, and store descriptive information about the model or problem in `Description`.

Example: `"An iterative approach to the Traveling Salesman problem"`

Data Types: `char` | `string`

Output Arguments

prob — Equation problem
`EquationProblem` object

Equation problem, returned as an `EquationProblem` object. Typically, to complete the problem description, you specify `prob.Equations` and, for nonlinear equations, an initial point structure. Solve a complete problem by calling `solve`.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

See Also
`EquationProblem` | `OptimizationEquality` | `optimvar` | `solve`

Topics
"Systems of Nonlinear Equations"

"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2019b

evaluate

Package: `optim.problemdef`

Evaluate optimization expression

Syntax

```
val = evaluate(expr,pt)
```

Description

Use `evaluate` to find the numeric value of an optimization expression at a point.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`val = evaluate(expr,pt)` returns the value of the optimization expression `expr` at the value `pt`.

Examples

Evaluate Optimization Expression At Point

Create an optimization expression in two variables.

```
x = optimvar('x',3,2);
y = optimvar('y',1,2);
expr = sum(x,1) - 2*y;
```

Evaluate the expression at a point.

```
xmat = [3,-1;
    0,1;
    2,6];
sol.x = xmat;
sol.y = [4,-3];
val = evaluate(expr,sol)
```

```
val = 1×2

    -3     12
```

Evaluate Objective Function At Solution

Solve a linear programming problem.

```
x = optimvar('x');
y = optimvar('y');
prob = optimproblem;
prob.Objective = -x -y/3;
prob.Constraints.cons1 = x + y <= 2;
prob.Constraints.cons2 = x + y/4 <= 1;
prob.Constraints.cons3 = x - y <= 2;
prob.Constraints.cons4 = x/4 + y >= -1;
prob.Constraints.cons5 = x + y >= 1;
prob.Constraints.cons6 = -x + y <= 2;

sol = solve(prob)

Solving problem using linprog.

Optimal solution found.

sol = struct with fields:
    x: 0.6667
    y: 1.3333
```

Find the value of the objective function at the solution.

```
val = evaluate(prob.Objective,sol)

val = -1.1111
```

Input Arguments

expr — Optimization expression
OptimizationExpression object

Optimization expression, specified as an `OptimizationExpression` object.

Example: `expr = 5*x+3`, where x is an `OptimizationVariable`

pt — Values of variables in expression
structure

Values of variables in expression, specified as a structure. The structure `pt` has the following requirements:

- All variables in `expr` match field names in `pt`.
- The values of the matching field names are numeric.

For example, `pt` can be the solution to an optimization problem, as returned by `solve`.

Example: `pt.x = 3, pt.y = -5`

Data Types: `struct`

Output Arguments

val — Numeric value of expression
double

Numeric value of expression, returned as a double.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

See Also
OptimizationExpression | infeasibility | solve

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

fcn2optimexpr

Package: `optim.problemdef`

Convert function to optimization expression

Syntax

```
[out1,out2,...,outN] = fcn2optimexpr(fcn,in1,in2,...,inK)
[out1,out2,...,outN] = fcn2optimexpr(fcn,in1,in2,...,inK,Name,Value)
```

Description

`[out1,out2,...,outN] = fcn2optimexpr(fcn,in1,in2,...,inK)` converts the function `fcn(in1,in2,...,inK)` to an optimization expression with `N` outputs.

`[out1,out2,...,outN] = fcn2optimexpr(fcn,in1,in2,...,inK,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, you can save a function evaluation by passing `OutputSize`.

Examples

Convert Objective Function to Expression

To use a MATLAB™ function in the problem-based approach when it is not composed of supported functions, first convert it to an optimization expression. See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8.

To use the objective function `gamma` (the mathematical function $\Gamma(x)$, an extension of the factorial function), create an optimization variable `x` and use it in a converted anonymous function.

```
x = optimvar('x');
obj = fcn2optimexpr(@gamma,x);
prob = optimproblem('Objective',obj);
show(prob)

  OptimizationProblem :

    Solve for:
       x

    minimize :
       gamma(x)
```

To solve the resulting problem, give an initial point structure and call `solve`.

```
x0.x = 1/2;
sol =  solve(prob,x0)

Solving problem using fminunc.
```

```
Local minimum found.

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

sol = struct with fields:
    x: 1.4616
```

For more complex functions, convert a function file. The function file `gammabrock.m` computes an objective of two optimization variables.

```
type gammabrock

function f = gammabrock(x,y)
f = (10*(y - gamma(x)))^2 + (1 - x)^2;
```

Include this objective in a problem.

```
x = optimvar('x','LowerBound',0);
y = optimvar('y');
obj = fcn2optimexpr(@gammabrock,x,y);
prob = optimproblem('Objective',obj);
show(prob)

  OptimizationProblem :

    Solve for:
       x, y

    minimize :
       gammabrock(x, y)


    variable bounds:
       0 <= x
```

The `gammabrock` function is a sum of squares. You get a more efficient problem formulation by expressing the function as an explicit sum of squares of optimization expressions.

```
f = fcn2optimexpr(@(x,y)y - gamma(x),x,y);
obj2 = (10*f)^2 + (1-x)^2;
prob2 = optimproblem('Objective',obj2);
```

To see the difference in efficiency, solve `prob` and `prob2` and examine the differences in number of iterations.

```
x0.x = 1/2;
x0.y = 1/2;
[sol,fval,~,output] = solve(prob,x0);

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

```
[sol2,fval2,~,output2] = solve(prob2,x0);

Solving problem using lsqnonlin.

Local minimum found.

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

fprintf('prob took %d iterations, but prob2 took %d iterations\n',output.iterations,output2.itera

prob took 21 iterations, but prob2 took 2 iterations
```

If your function has several outputs, you can use them as elements of the objective function. In this case, u is a 2-by-2 variable, v is a 2-by-1 variable, and `expfn3` has three outputs.

```
type expfn3

function [f,g,mineval] = expfn3(u,v)
mineval = min(eig(u));
f = v'*u*v;
f = -exp(-f);
t = u*v;
g = t'*t + sum(t) - 3;
```

Create appropriately sized optimization variables, and create an objective function from the first two outputs.

```
u = optimvar('u',2,2);
v = optimvar('v',2);
[f,g,mineval] = fcn2optimexpr(@expfn3,u,v);
prob = optimproblem;
prob.Objective = f*g/(1 + f^2);
show(prob)

  OptimizationProblem :

    Solve for:
       u, v

    minimize :
       ((arg3 .* arg4) ./ (1 + arg2.^2))

       where:

          [arg2,~,~] = expfn3(u, v);
          [arg3,~,~] = expfn3(u, v);
          [~,arg4,~] = expfn3(u, v);
```

You can use the `mineval` output in a subsequent constraint expression.

Create Nonlinear Constraints from Function

In problem-based optimization, constraints are two optimization expressions with a comparison operator (==, <=, or >=) between them. You can use `fcn2optimexpr` to create one or both optimization expressions. See "Convert Nonlinear Function to Optimization Expression" on page 7-8.

Create the nonlinear constraint that `gammafn2` is less than or equal to –1/2. This function of two variables is in the `gammafn2.m` file.

```
type gammafn2
```

```
function f = gammafn2(x,y)
f = -gamma(x)*(y/(1+y^2));
```

Create optimization variables, convert the function file to an optimization expression, and then express the constraint as `confn`.

```
x = optimvar('x','LowerBound',0);
y = optimvar('y','LowerBound',0);
expr1 = fcn2optimexpr(@gammafn2,x,y);
confn = expr1 <= -1/2;
show(confn)
```

```
  gammafn2(x, y) <= -0.5
```

Create another constraint that `gammafn2` is greater than or equal to `x + y`.

```
confn2 = expr1 >= x + y;
```

Create an optimization problem and place the constraints in the problem.

```
prob = optimproblem;
prob.Constraints.confn = confn;
prob.Constraints.confn2 = confn2;
show(prob)
```

```
  OptimizationProblem :

    Solve for:
       x, y

    minimize :

    subject to confn:
       gammafn2(x, y) <= -0.5

    subject to confn2:
       gammafn2(x, y) >= (x + y)

    variable bounds:
       0 <= x

       0 <= y
```

Compute Common Objective and Constraint Efficiently

If your problem involves a common, time-consuming function to compute the objective and nonlinear constraint, you can save time by using the `'ReuseEvaluation'` name-value pair argument. The `rosenbrocknorm` function computes both the Rosenbrock objective function and the norm of the argument for use in the constraint $\|x\|^2 \le 4$.

```
type rosenbrocknorm
```

```
function [f,c] = rosenbrocknorm(x)
pause(1) % Simulates time-consuming function
c = dot(x,x);
f = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
```

Create a 2-D optimization variable x. Then convert rosenbrocknorm to an optimization expression by using fcn2optimexpr and specifying 'ReuseEvaluation'.

```
x = optimvar('x',2);
[f,c] = fcn2optimexpr(@rosenbrocknorm,x,'ReuseEvaluation',true);
```

Create objective and constraint expressions from the returned expressions. Include the objective and constraint expressions in an optimization problem. Review the problem using show.

```
prob = optimproblem('Objective',f);
prob.Constraints.cineq = c <= 4;
show(prob)

  OptimizationProblem :

    Solve for:
       x

    minimize :
       [argout,~] = rosenbrocknorm(x)


    subject to cineq:
       arg_LHS <= 4

       where:

          [~,arg_LHS] = rosenbrocknorm(x);
```

Solve the problem starting from the initial point x0.x = [-1;1], timing the result.

```
x0.x = [-1;1];
tic
[sol,fval,exitflag,output] = solve(prob,x0)

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

<stopping criteria details>

sol = struct with fields:
    x: [2×1 double]


fval = 4.5793e-11

exitflag =
    OptimalSolution
```

```
output = struct with fields:
         iterations: 44
          funcCount: 164
     constrviolation: 0
           stepsize: 4.3124e-08
          algorithm: 'interior-point'
       firstorderopt: 5.1691e-07
        cgiterations: 10
            message: '↵Local minimum found that satisfies the constraints.↵↵Optimization complete
        bestfeasible: [1×1 struct]
             solver: 'fmincon'
```

```
toc
```

```
Elapsed time is 164.410724 seconds.
```

The solution time in seconds is nearly the same as the number of function evaluations. This result indicates that the solver reused function values, and did not waste time by reevaluating the same point twice.

For a more extensive example, see "Objective and Constraints Having a Common Function in Serial or Parallel, Problem-Based" on page 2-52. For more information on using `fcn2optimexpr`, see "Convert Nonlinear Function to Optimization Expression" on page 7-8.

Input Arguments

`fcn` — Function to convert
function handle

Function to convert, specified as a function handle.

Example: `@sin` specifies the sine function.

Data Types: `function_handle`

`in` — Input argument
MATLAB variable

Input argument, specified as a MATLAB variable. The input can have any data type and any size. You can include any problem variables or data in the input argument `in`; see "Pass Extra Parameters in Problem-Based Approach" on page 10-11.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[out1,out2] = fcn2optimexpr(@fun,x,y,'OutputSize', [1,1],'ReuseEvaluation',true)` specifies that `out1` and `out2` are scalars that will be reused between objective and constraint functions without recalculation.

`OutputSize` — Size of output expressions
integer vector | cell array of integer vectors

Size of the output expressions, specified as:

- An integer vector — If the function has one output `out1`, `OutputSize` specifies the size of `out1`. If the function has multiple outputs `out1,...,outN`, `OutputSize` specifies that all outputs have the same size.

- A cell array of integer vectors — The size of output `out`j is the jth element of `OutputSize`.

Note A scalar has size `[1,1]`.

If you do not specify the `'OutputSize'` name-value pair argument, then `fcn2optimexpr` passes data to `fcn` in order to determine the size of the outputs (see "Algorithms" on page 16-36). By specifying `'OutputSize'`, you enable `fcn2optimexpr` to skip this step, which saves time. Also, if you do not specify `'OutputSize'` and the evaluation of `fcn` fails for any reason, then `fcn2optimexpr` fails as well.

Example: `[out1,out2,out3] = fcn2optimexpr(@fun,x,'OutputSize',[1,1])` specifies that the three outputs `[out1,out2,out3]` are scalars.

Example: `[out1,out2] = fcn2optimexpr(@fun,x,'OutputSize',{[4,4],[3,5]})` specifies that `out1` has size 4-by-4 and `out2` has size 3-by-5.

Data Types: `double` | `cell`

`ReuseEvaluation` — Indicator to reuse values
`false` (default) | `true`

Indicator to reuse values, specified as `false` (do not reuse) or `true` (reuse).

`'ReuseEvaluation'` can make your problem run faster when, for example, the objective and some nonlinear constraints rely on a common calculation. In this case, the solver stores the value for reuse wherever needed and avoids recalculating the value.

Reusable values involve some overhead, so it is best to enable reusable values only for expressions that share a value.

Example: `[out1,out2,out3] = fcn2optimexpr(@fun,x,'ReuseEvaluation',true)` allows `out1`, `out2`, and `out3` to be used in multiple computations, with the outputs being calculated only once per evaluation point.

Data Types: `logical`

Output Arguments

`out` — Output argument
`OptimizationExpression`

Output argument, returned as an `OptimizationExpression`. The size of the expression depends on the input function.

Tips

- When possible, create your objective or nonlinear constraint functions by using supported operations on optimization variables and expressions instead of `fcn2optimexpr`. Doing so has these advantages:

  - `solve` includes gradients calculated by automatic differentiation. See "Effect of Automatic Differentiation in Problem-Based Optimization" on page 7-23.

  - `solve` has a wider choice of available solvers. When using `fcn2optimexpr`, `solve` uses only `fmincon` or `fminunc`.

  For details, see "Convert Nonlinear Function to Optimization Expression" on page 7-8 and "Supported Operations on Optimization Variables and Expressions" on page 10-43.

Algorithms

To find the output size of each returned expression when you do not specify `OutputSize`, `fcn2optimexpr` evaluates the function at the following point for each element of the problem variables.

Variable Characteristics	Evaluation Point
Finite upper bound `ub` and finite lower bound `lb`	`(lb + ub)/2 + ((ub - lb)/2)*eps`
Finite lower bound and no upper bound	`lb + max(1,abs(lb))*eps`
Finite upper bound and no lower bound	`ub - max(1,abs(ub))*eps`
No bounds	`1 + eps`
Variable is specified as an integer	`floor` of the point given previously

An evaluation point might lead to an error in function evaluation. To avoid this error, specify `'OutputSize'`.

See Also

Topics
"Problem-Based Optimization Workflow" on page 10-2
"Convert Nonlinear Function to Optimization Expression" on page 7-8
"Optimization Expressions" on page 10-6
"Pass Extra Parameters in Problem-Based Approach" on page 10-11

Introduced in R2019a

fgoalattain

Solve multiobjective goal attainment problems

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,exitflag,output] = fgoalattain( ___ )
[x,fval,attainfactor,exitflag,output,lambda] = fgoalattain( ___ )
```

Description

`fgoalattain` solves the goal attainment problem, a formulation for minimizing a multiobjective optimization problem.

`fgoalattain` finds the minimum of a problem specified by

$$\underset{x,\,\gamma}{\text{minimize } \gamma} \text{ such that } \begin{cases} F(x) - \text{weight} \cdot \gamma \leq \text{goal} \\ \quad c(x) \leq 0 \\ \quad ceq(x) = 0 \\ \quad A \cdot x \leq b \\ \quad Aeq \cdot x = beq \\ \quad lb \leq x \leq ub\,. \end{cases}$$

`weight`, `goal`, b, and beq are vectors, A and Aeq are matrices, and $F(x)$, $c(x)$, and $ceq(x)$, 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 2-31.

`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`.

Note "Passing Extra Parameters" on page 2-57 explains how to pass extra parameters to the objective functions and nonlinear constraint functions, if necessary.

`x = fgoalattain(fun,x0,goal,weight,A,b)` solves the goal attainment problem subject to the inequalities A*x ≤ b.

`x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq)` solves the goal attainment problem subject to the equalities Aeq*x = beq. If no inequalities exist, set A = [] and b = [].

`x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub)` solves the goal attainment problem subject to the bounds `lb ≤ x ≤ ub`. If no equalities exist, set `Aeq = []` and `beq = []`. If `x(i)` is unbounded below, set `lb(i) = -Inf`; if `x(i)` is unbounded above, set `ub(i) = Inf`.

Note See "Iterations Can Violate Constraints" on page 2-33.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the output `fval` is `[]`.

`x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub,nonlcon)` solves the goal attainment problem subject to the nonlinear inequalities `c(x)` or equalities `ceq(x)` defined in `nonlcon`. `fgoalattain` optimizes such that `c(x) ≤ 0` and `ceq(x) = 0`. If no bounds exist, set `lb = []` or `ub = []`, or both.

`x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub,nonlcon,options)` solves the goal attainment problem with the optimization options specified in `options`. Use `optimoptions` to set these options.

`x = fgoalattain(problem)` solves the goal attainment problem for `problem`, a structure described in `problem`.

`[x,fval] = fgoalattain( ___ )`, for any syntax, returns the values of the objective functions computed in `fun` at the solution `x`.

`[x,fval,attainfactor,exitflag,output] = fgoalattain( ___ )` additionally returns the attainment factor at the solution `x`, a value `exitflag` that describes the exit condition of `fgoalattain`, and a structure `output` with information about the optimization process.

`[x,fval,attainfactor,exitflag,output,lambda] = fgoalattain( ___ )` additionally returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

Examples

Basic Goal Attainment Problem

Consider the two-objective function

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

This function clearly minimizes $F_1(x)$ at $x = 3$, attaining the value 2, and minimizes $F_2(x)$ at $x = 0$, attaining the value 5.

Set the goal [3,6] and weight [1,1], and solve the goal attainment problem starting at `x0 = 1`.

```
fun = @(x)[2+(x-3)^2;5+x^2/4];
goal = [3,6];
weight = [1,1];
x0 = 1;
x = fgoalattain(fun,x0,goal,weight)
```

```
Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = 2.0000
```

Find the value of $F(x)$ at the solution.

```
fun(x)
```

```
ans = 2×1

    3.0000
    6.0000
```

`fgoalattain` achieves the goals exactly.

Goal Attainment with Linear Constraint

The objective function is

$$F(x) = \begin{bmatrix} 2 + \|x - p_1\|^2 \\ 5 + \|x - p_2\|^2/4 \end{bmatrix}.$$

Here, p_1 = [2,3] and p_2 = [4,1]. The goal is [3,6], the weight is [1,1], and the linear constraint is $x_1 + x_2 \leq 4$.

Create the objective function, goal, and weight.

```
p_1 = [2,3];
p_2 = [4,1];
fun = @(x)[2 + norm(x-p_1)^2;5 + norm(x-p_2)^2/4];
goal = [3,6];
weight = [1,1];
```

Create the linear constraint matrices A and b representing A*x <= b.

```
A = [1,1];
b = 4;
```

Set an initial point [1,1] and solve the goal attainment problem.

```
x0 = [1,1];
x = fgoalattain(fun,x0,goal,weight,A,b)
```

```
Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = 1×2
```

```
    2.0694    1.9306
```

Find the value of $F(x)$ at the solution.

```
fun(x)
```

```
ans = 2×1
```

```
    3.1484
    6.1484
```

`fgoalattain` does not meet the goals. Because the weights are equal, the solver underachieves each goal by the same amount.

Goal Attainment with Bounds

The objective function is

$$F(x) = \begin{bmatrix} 2 + \|x - p_1\|^2 \\ 5 + \|x - p_2\|^2/4 \end{bmatrix}.$$

Here, p_1 = [2,3] and p_2 = [4,1]. The goal is [3,6], the weight is [1,1], and the bounds are $0 \le x_1 \le 3$, $2 \le x_2 \le 5$.

Create the objective function, goal, and weight.

```
p_1 = [2,3];
p_2 = [4,1];
fun = @(x)[2 + norm(x-p_1)^2;5 + norm(x-p_2)^2/4];
goal = [3,6];
weight = [1,1];
```

Create the bounds.

```
lb = [0,2];
ub = [3,5];
```

Set the initial point to [1,4] and solve the goal attainment problem.

```
x0 = [1,4];
A = []; % no linear constraints
b = [];
Aeq = [];
beq = [];
x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub)
```

```
Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = 1×2
```

```
     2.6667     2.3333
```

Find the value of $F(x)$ at the solution.

```
fun(x)
```

```
ans = 2×1

     2.8889
     5.8889
```

`fgoalattain` more than meets the goals. Because the weights are equal, the solver overachieves each goal by the same amount.

Goal Attainment with Nonlinear Constraint

The objective function is

$$F(x) = \begin{bmatrix} 2 + \|x - p_1\|^2 \\ 5 + \|x - p_2\|^2/4 \end{bmatrix}.$$

Here, p_1 = [2,3] and p_2 = [4,1]. The goal is [3,6], the weight is [1,1], and the nonlinear constraint is $\|x\|^2 \le 4$.

Create the objective function, goal, and weight.

```
p_1 = [2,3];
p_2 = [4,1];
fun = @(x)[2 + norm(x-p_1)^2;5 + norm(x-p_2)^2/4];
goal = [3,6];
weight = [1,1];
```

The nonlinear constraint function is in the `norm4.m` file.

```
type norm4
```

```
function [c,ceq] = norm4(x)
ceq = [];
c = norm(x)^2 - 4;
```

Create empty input arguments for the linear constraints and bounds.

```
A = [];
Aeq = [];
b = [];
beq = [];
lb = [];
ub = [];
```

Set the initial point to [1,1] and solve the goal attainment problem.

```
x0 = [1,1];
x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub,@norm4)
```

```
Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = 1×2

    1.1094    1.6641
```

Find the value of $F(x)$ at the solution.

```
fun(x)

ans = 2×1

    4.5778
    7.1991
```

fgoalattain does not meet the goals. Despite the equal weights, $F_1(x)$ is about 1.58 from its goal of 3, and $F_2(x)$ is about 1.2 from its goal of 6. The nonlinear constraint prevents the solution x from achieving the goals equally.

Goal Attainment Using Nondefault Options

Monitor a goal attainment solution process by setting options to return iterative display.

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

The objective function is

$$F(x) = \begin{bmatrix} 2 + \|x - p_1\|^2 \\ 5 + \|x - p_2\|^2/4 \end{bmatrix}.$$

Here, p_1 = [2,3] and p_2 = [4,1]. The goal is [3,6], the weight is [1,1], and the linear constraint is $x_1 + x_2 \le 4$.

Create the objective function, goal, and weight.

```
p_1 = [2,3];
p_2 = [4,1];
fun = @(x)[2 + norm(x-p_1)^2;5 + norm(x-p_2)^2/4];
goal = [3,6];
weight = [1,1];
```

Create the linear constraint matrices A and b representing A*x <= b.

```
A = [1,1];
b = 4;
```

Create empty input arguments for the linear equality constraints, bounds, and nonlinear constraints.

```
Aeq = [];
beq = [];
lb = [];
ub = [];
nonlcon = [];
```

Set an initial point [1,1] and solve the goal attainment problem.

```
x0 = [1,1];
x = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

```
                    Attainment          Max     Line search     Directional
    Iter F-count        factor     constraint    steplength      derivative  Procedure
       0      4              0              4
       1      9             -1            2.5              1          -0.535
       2     14     -5.194e-09         0.2813              1           0.883
       3     19         0.1452       0.005926              1           0.883
       4     24         0.1484      2.868e-06              1           0.883
       5     29         0.1484      6.839e-13              1           0.883   Hessian modified
```

```
Local minimum possible. Constraints satisfied.
```

```
fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.
```

```
x = 1×2
```

```
    2.0694    1.9306
```

The positive value of the reported attainment factor indicates that `fgoalattain` does not find a solution satisfying the goals.

Obtain Objective Function Values in Goal Attainment

The objective function is

$$F(x) = \begin{bmatrix} 2 + \|x - p_1\|^2 \\ 5 + \|x - p_2\|^2/4 \end{bmatrix}.$$

Here, p_1 = [2,3] and p_2 = [4,1]. The goal is [3,6], the weight is [1,1], and the linear constraint is $x_1 + x_2 \le 4$.

Create the objective function, goal, and weight.

```
p_1 = [2,3];
p_2 = [4,1];
fun = @(x)[2 + norm(x-p_1)^2;5 + norm(x-p_2)^2/4];
goal = [3,6];
weight = [1,1];
```

Create the linear constraint matrices A and b representing A*x <= b.

```
A = [1,1];
b = 4;
```

Set an initial point [1,1] and solve the goal attainment problem. Request the value of the objective function.

```
x0 = [1,1];
[x,fval] = fgoalattain(fun,x0,goal,weight,A,b)
```

Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than twice the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.

x = *1×2*

    2.0694   1.9306

fval = *2×1*

    3.1484
    6.1484

The objective function values are higher than the goal, meaning `fgoalattain` does not satisfy the goal.

Obtain All Outputs in Goal Attainment

The objective function is

$$F(x) = \begin{bmatrix} 2 + \|x - p_1\|^2 \\ 5 + \|x - p_2\|^2/4 \end{bmatrix}.$$

Here, p_1 = [2,3] and p_2 = [4,1]. The goal is [3,6], the weight is [1,1], and the linear constraint is $x_1 + x_2 \le 4$.

Create the objective function, goal, and weight.

```
p_1 = [2,3];
p_2 = [4,1];
fun = @(x)[2 + norm(x-p_1)^2;5 + norm(x-p_2)^2/4];
goal = [3,6];
weight = [1,1];
```

Create the linear constraint matrices A and b representing A*x <= b.

```
A = [1,1];
b = 4;
```

Set an initial point [1,1] and solve the goal attainment problem. Request the value of the objective function, attainment factor, exit flag, output structure, and Lagrange multipliers.

```
x0 = [1,1];
[x,fval,attainfactor,exitflag,output,lambda] = fgoalattain(fun,x0,goal,weight,A,b)
```

```
Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.
```

x = *1×2*

```
    2.0694    1.9306
```

fval = *2×1*

```
    3.1484
    6.1484
```

attainfactor = 0.1484

exitflag = 4

output = *struct with fields:*
```
        iterations: 6
         funcCount: 29
       lssteplength: 1
           stepsize: 4.1263e-13
          algorithm: 'active-set'
       firstorderopt: []
      constrviolation: 6.8393e-13
            message: '...'
```

lambda = *struct with fields:*
```
         lower: [2x1 double]
         upper: [2x1 double]
         eqlin: [0x1 double]
       eqnonlin: [0x1 double]
        ineqlin: 0.5394
      ineqnonlin: [0x1 double]
```

The positive value of `attainfactor` indicates that the goals are not attained; you can also see this by comparing `fval` with `goal`.

The `lambda.ineqlin` value is nonzero, indicating that the linear inequality constrains the solution.

Effects of Weights, Goals, and Constraints in Goal Attainment

The objective function is

$$F(x) = \begin{bmatrix} 2 + \|x - p_1\|^2 \\ 5 + \|x - p_2\|^2/4 \end{bmatrix}.$$

Here, p_1 = [2,3] and p_2 = [4,1]. The goal is [3,6], and the initial weight is [1,1].

Create the objective function, goal, and initial weight.

```
p_1 = [2,3];
p_2 = [4,1];
fun = @(x)[2 + norm(x-p_1)^2;5 + norm(x-p_2)^2/4];
goal = [3,6];
weight = [1,1];
```

Set the linear constraint $x_1 + x_2 \le 4$.

```
A = [1 1];
b = 4;
```

Solve the goal attainment problem starting from the point `x0 = [1 1]`.

```
x0 = [1 1];
[x,fval] = fgoalattain(fun,x0,goal,weight,A,b)
```

Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = *1×2*

   2.0694   1.9306

fval = *2×1*

    3.1484
    6.1484

Each component of `fval` is above the corresponding component of `goal`, indicating that the goals
are not attained.

Increase the importance of satisfying the first goal by setting `weight(1)` to a smaller value.

```
weight(1) = 1/10;
[x,fval] = fgoalattain(fun,x0,goal,weight,A,b)
```

Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = *1×2*

   2.0115   1.9885

fval = *2×1*

    3.0233

```
    6.2328
```

Now the value of `fval(1)` is much closer to `goal(1)`, whereas `fval(2)` is farther from `goal(2)`.

Change `goal(2)` to 7, which is above the current solution. The solution changes.

```
goal(2) = 7;
[x,fval] = fgoalattain(fun,x0,goal,weight,A,b)
```

```
Local minimum possible. Constraints satisfied.
```

```
fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.
```

```
x = 1×2

    1.9639    2.0361
```

```
fval = 2×1

    2.9305
    6.3047
```

Both components of `fval` are less than the corresponding components of `goal`. But `fval(1)` is much closer to `goal(1)` than `fval(2)` is to `goal(2)`. A smaller weight is more likely to make its component nearly satisfied when the goals cannot be achieved, but makes the degree of overachievement less when the goal can be achieved.

Change the weights to be equal. The `fval` results have equal distance from their goals.

```
weight(2) = 1/10;
[x,fval] = fgoalattain(fun,x0,goal,weight,A,b)
```

```
Local minimum possible. Constraints satisfied.
```

```
fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.
```

```
x = 1×2

    1.7613    2.2387
```

```
fval = 2×1

    2.6365
    6.6365
```

Constraints can keep the resulting `fval` from being equally close to the goals. For example, set an upper bound of 2 on `x(2)`.

```
ub = [Inf,2];
lb = [];
```

```
Aeq = [];
beq = [];
[x,fval] = fgoalattain(fun,x0,goal,weight,A,b,Aeq,beq,lb,ub)

Local minimum possible. Constraints satisfied.

fgoalattain stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = 1×2

    2.0000    2.0000


fval = 2×1

    3.0000
    6.2500
```

In this case, `fval(1)` meets its goal exactly, but `fval(2)` is less than its goal.

Input Arguments

fun — Objective functions
function handle | function name

Objective functions, specified as a function handle or function name. `fun` is a function that accepts a vector `x` and returns a vector `F`, the objective functions evaluated at `x`. You can specify the function `fun` 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 arrays, `fgoalattain` converts them to vectors using linear indexing (see "Array Indexing").

To make an objective function as near as possible to a goal value (that is, neither greater than nor less than), use `optimoptions` to set the `EqualityGoalCount` 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`.

Suppose that the gradient of the objective function can also be computed *and* the `SpecifyObjectiveGradient` option is `true`, as set by:

```
options = optimoptions('fgoalattain','SpecifyObjectiveGradient',true)
```

In this case, 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) (that is, the jth column of G is the gradient of the jth objective function F(j)).

Note Setting SpecifyObjectiveGradient to true is effective only when the problem has no nonlinear constraints, or the problem has a nonlinear constraint with SpecifyConstraintGradient set to true. 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.

Data Types: char | string | 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 x0 and the size of x0 to determine the number and size of variables that fun accepts.

Example: x0 = [1,2,3,4]

Data Types: double

goal — Goal to attain
real vector

Goal to attain, specified as a real vector. fgoalattain attempts to find the smallest multiplier γ that makes these inequalities hold for all values of *i* at the solution *x*:

$$F_i(x) - \text{goal}_i \leq \text{weight}_i\, \gamma.$$

Assuming that weight is a positive vector:

- If the solver finds a point x that simultaneously achieves all the goals, then the attainment factor γ is negative, and the goals are overachieved.
- If the solver cannot find a point x that simultaneously achieves all the goals, then the attainment factor γ is positive, and the goals are underachieved.

Example: [1 3 6]

Data Types: double

weight — Relative attainment factor
real vector

Relative attainment factor, specified as a real vector. fgoalattain attempts to find the smallest multiplier γ that makes these inequalities hold for all values of *i* at the solution *x*:

$$F_i(x) - \text{goal}_i \leq \text{weight}_i\, \gamma.$$

When the values of goal are *all nonzero*, to ensure the same percentage of underachievement or overattainment of the active objectives, set weight 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 causes the corresponding goal constraint to be treated as a hard constraint rather than a goal constraint. An alternative method to setting a hard constraint is to use the input argument `nonlcon`.

When `weight` is positive, `fgoalattain` attempts to make the objective functions less than the goal values. To make the objective functions greater than the goal values, set `weight` to be negative rather than positive. To see some effects of weights on a solution, see "Effects of Weights, Goals, and Constraints in Goal Attainment" on page 16-45.

To make an objective function as near as possible to a goal value, use the `EqualityGoalCount` option and specify the objective as the first element of the vector returned by `fun` (see `fun` and `options`). For an example, see "Multi-Objective Goal Attainment Optimization" on page 8-18.

Example: `abs(goal)`

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

$x_1 + 2x_2 \leq 10$
$3x_1 + 4x_2 \leq 20$
$5x_1 + 6x_2 \leq 30$,

enter 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, use `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, consider these inequalities:

$x_1 + 2x_2 \leq 10$
$3x_1 + 4x_2 \leq 20$
$5x_1 + 6x_2 \leq 30.$

Specify the inequalities by entering the following constraints.

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x components sum to 1 or less, use `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

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20,$

enter these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use `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 Me-by-N.

For example, consider these equalities:

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20.$

Specify the equalities by entering the following constraints.

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use `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 the number of elements in `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)`.

If there are fewer elements in `lb` than in `x0`, solvers issue a warning.

Example: To specify that all x components are positive, use `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 the number of elements in `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)`.

If there are fewer elements in `ub` than in `x0`, solvers issue a warning.

Example: To specify that all x components are less than 1, use `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`. `fgoalattain` attempts to satisfy `c(x) <= 0` for all entries of `c`.
- `ceq(x)` is the array of nonlinear equality constraints at `x`. `fgoalattain` attempts to satisfy `ceq(x) = 0` for all entries of `ceq`.

For example,

```
x = fgoalattain(@myfun,x0,...,@mycon)
```

where `mycon` is a MATLAB function such as the following:

```
function [c,ceq] = mycon(x)
c = ...     % Compute nonlinear inequalities at x.
ceq = ...   % Compute nonlinear equalities at x.
```

Suppose that the gradients of the constraints can also be computed *and* the `SpecifyConstraintGradient` option is `true`, as set by:

```
options = optimoptions('fgoalattain','SpecifyConstraintGradient',true)
```

In this case, 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)`. See "Nonlinear Constraints" on page 2-37 for an explanation of 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)` (that is, the jth column of `GC` is the gradient of the jth inequality constraint `c(j)`). Likewise, if `ceq` has `p` components, the gradient `GCeq` of `ceq(x)` is an n-by-p matrix, where `GCeq(i,j)` is the partial derivative of `ceq(j)` with respect to `x(i)` (that is, the jth column of `GCeq` is the gradient of the jth equality constraint `ceq(j)`).

Note Setting `SpecifyConstraintGradient` to `true` is effective only when `SpecifyObjectiveGradient` is set to `true`. 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 accept only inputs of type `double`, user-supplied objective and nonlinear constraint functions must return outputs of type `double`.

See "Passing Extra Parameters" on page 2-57 for an explanation of how to parameterize the nonlinear constraint function `nonlcon`, if necessary.

Data Types: char | function_handle | string

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 are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

For details about options that have different names for `optimset`, see "Current and Legacy Option Names" on page 15-22.

Option	Description
ConstraintTolerance	Termination tolerance on the constraint violation, a positive scalar. The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68. For optimset, the name is TolCon.
Diagnostics	Display of diagnostic information about the function to be minimized or solved. The choices are 'on' or 'off' (the default).
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 3-14): • '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.
EqualityGoalCount	Number of objectives required for the objective fun to equal the goal goal (a nonnegative integer). The objectives must be partitioned into the first few elements of F. The default is 0. For an example, see "Multi-Objective Goal Attainment Optimization" on page 8-18. For optimset, the name is GoalsExactAchieve.

Option	Description

FiniteDifferenceStepSize	Scalar or vector step size factor for finite differences. When you set FiniteDifferenceStepSize to a vector v, the forward finite differences 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 FiniteDifferenceStepSize expands to a vector. The default is sqrt(eps) for forward finite differences, and eps^(1/3) for central finite differences.
	
	For optimset, the name is FinDiffRelStep.

FiniteDifferenceType	Type of finite differences used to estimate gradients, either 'forward' (default), or 'central' (centered). 'central' takes twice as many function evaluations, but is generally more accurate. The algorithm is careful to obey bounds when estimating both types of finite differences. For example, it might take a backward step, rather than a forward step, to avoid evaluating at a point outside the bounds. For optimset, the name is FinDiffType.
FunctionTolerance	Termination tolerance on the function value (a positive scalar). The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68. For optimset, the name is TolFun.
FunValCheck	Check that signifies whether the objective function and constraint 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.
MaxFunctionEvaluations	Maximum number of function evaluations allowed (a positive integer). The default is 100*numberOfVariables. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9. For optimset, the name is MaxFunEvals.

Option	Description
`MaxIterations`	Maximum number of iterations allowed (a positive integer). The default is `400`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9. For `optimset`, the name is `MaxIter`.
MaxSQPIter	Maximum number of SQP iterations allowed (a positive integer). The default is `10*max(numberOfVariables, numberOfInequalities + numberOfBounds)`.
MeritFunction	If this option is set to `'multiobj'` (the default), use goal attainment merit function. If this option is set to `'singleobj'`, use the `fmincon` merit function.
`OptimalityTolerance`	Termination tolerance on the first-order optimality (a positive scalar). The default is `1e-6`. See "First-Order Optimality Measure" on page 3-11. For `optimset`, the name is `TolFun`.
`OutputFcn`	One or more user-defined functions that an optimization function calls at each iteration. Pass a function handle or a cell array of function handles. The default is none (`[]`). See "Output Function and Plot Function Syntax" on page 15-27.

Option	Description
PlotFcn	Plots showing various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a name, function handle, or cell array of names or function handles. For custom plot functions, pass function handles. The default is none ([]). • 'optimplotx' plots the current point. • 'optimplotfunccount' plots the function count. • 'optimplotfval' plots the objective function values. • 'optimplotconstrviolation' plots the maximum constraint violation. • 'optimplotstepsize' plots the step size. Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27. For optimset, the name is PlotFcns.
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· max($\|x(i)\|$,$\|typicalx(i)\|$). This option provides control over the magnitude of the displacements in x when the solver takes steps that are too large. The default is none ([]).
RelLineSrchBndDuration	Number of iterations for which the bound specified in RelLineSrchBnd should be active. The default is 1.
SpecifyConstraintGradient	Gradient for nonlinear constraint functions defined by the user. When this option is set to true, fgoalattain expects the constraint function to have four outputs, as described in nonlcon. When this option is set to false (the default), fgoalattain estimates gradients of the nonlinear constraints using finite differences. For optimset, the name is GradConstr and the values are 'on' or 'off'.

Option	Description
SpecifyObjectiveGradient	Gradient for the objective function defined by the user. Refer to the description of fun to see how to define the gradient. Set this option to true to have fgoalattain use a user-defined gradient of the objective function. The default, false, causes fgoalattain to estimate gradients using finite differences. For optimset, the name is GradObj and the values are 'on' or 'off'.
StepTolerance	Termination tolerance on x (a positive scalar). The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68. For optimset, the name is TolX.
TolConSQP	Termination tolerance on the inner iteration SQP constraint violation (a positive scalar). The default is 1e-6.
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 fgoalattain function uses TypicalX for scaling finite differences for gradient estimation.
UseParallel	Indication of parallel computing. When true, fgoalattain estimates gradients in parallel. The default is false. See "Parallel Computing".

Example: optimoptions('fgoalattain','PlotFcn','optimplotfval')

problem — Problem structure
structure

Problem structure, specified as a structure with the fields in this table.

Field Name	Entry
objective	Objective function fun
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

Field Name	Entry
ub	Vector of upper bounds
nonlcon	Nonlinear constraint function
solver	'fgoalattain'
options	Options created with optimoptions

You must supply at least the objective, x0, goal, weight, solver, and options fields in the problem structure.

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 4-18.

fval — Objective function values at solution
real array

Objective function values at the solution, returned as a real array. Generally, fval = fun(x).

attainfactor — Attainment factor
real number

Attainment factor, returned as a real number. 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. See goal.

exitflag — Reason fgoalattain stopped
integer

Reason fgoalattain stopped, returned as an integer.

1	Function converged to a solution x
4	Magnitude of the search direction was less than the specified tolerance, and the constraint violation was less than options.ConstraintTolerance
5	Magnitude of the directional derivative was less than the specified tolerance, and the constraint violation was less than options.ConstraintTolerance
0	Number of iterations exceeded options.MaxIterations or the number of function evaluations exceeded options.MaxFunctionEvaluations
-1	Stopped by an output function or plot function
-2	No feasible point was found.

output — Information about optimization process
structure

Information about the optimization process, returned as a structure with the fields in this table.

iterations	Number of iterations taken
funcCount	Number of function evaluations
lssteplength	Size of the line search step relative to the search direction
constrviolation	Maximum of the constraint functions
stepsize	Length of the last displacement in x
algorithm	Optimization algorithm used
firstorderopt	Measure of first-order optimality
message	Exit message

lambda — Lagrange multipliers at solution
structure

Lagrange multipliers at the solution, returned as a structure with the fields in this table.

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

Algorithms

For a description of the fgoalattain algorithm and a discussion of goal attainment concepts, see "Algorithms" on page 8-3.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for fgoalattain.

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the 'UseParallel' option to true.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see "Using Parallel Computing in Optimization Toolbox" on page 14-5.

See Also

Optimize | `fmincon` | `fminimax` | `optimoptions`

Topics

"Multi-Objective Goal Attainment Optimization" on page 8-18
"Generate and Plot Pareto Front" on page 8-15
"Create Function Handle"
"Multiobjective Optimization"

Introduced before R2006a

findindex

Package: `optim.problemdef`

Find numeric index equivalents of named index variables

Syntax

```
[numindex1,numindex2,...,numindexk] = findindex(var,strindex1,strindex2,...,
strindexk)
numindex = findindex(var,strindex1,strindex2,...,strindexk)
```

Description

`[numindex1,numindex2,...,numindexk] = findindex(var,strindex1,strindex2,...,strindexk)` finds the numeric index equivalents of the named index variables in the optimization variable `var`.

`numindex = findindex(var,strindex1,strindex2,...,strindexk)` finds the linear index equivalents of the named index variables.

Examples

Find Numeric Equivalents of Named Index Variables

Create an optimization variable named `colors` that is indexed by the primary additive color names and the primary subtractive color names. Include `'black'` and `'white'` as additive color names and `'black'` as a subtractive color name.

```
colors = optimvar('colors',["black","white","red","green","blue"],["cyan","magenta","yellow","bla
```

Find the index numbers for the additive colors `'red'` and `'black'` and for the subtractive color `'black'`.

```
[idxadd,idxsub] = findindex(colors,{'red','black'},{'black'})
```

```
idxadd = 1×2

     3     1


idxsub = 4
```

Find Linear Index Equivalents of Named Index Variables

Create an optimization variable named `colors` that is indexed by the primary additive color names and the primary subtractive color names. Include `'black'` and `'white'` as additive color names and `'black'` as a subtractive color name.

```
colors = optimvar('colors',["black","white","red","green","blue"],["cyan","magenta","yellow","bla
```

Find the linear index equivalents to the combinations ["white","black"], ["red","cyan"], ["green","magenta"], and ["blue","yellow"].

```
idx = findindex(colors,["white","red","green","blue"],["black","cyan","magenta","yellow"])
```

```
idx = 1×4

   17     3     9    15
```

View Solution with Index Variables

Create and solve an optimization problem using named index variables. The problem is to maximize the profit-weighted flow of fruit to various airports, subject to constraints on the weighted flows.

```
rng(0) % For reproducibility
p = optimproblem('ObjectiveSense', 'maximize');
flow = optimvar('flow', ...
    {'apples', 'oranges', 'bananas', 'berries'}, {'NYC', 'BOS', 'LAX'}, ...
    'LowerBound',0,'Type','integer');
p.Objective = sum(sum(rand(4,3).*flow));
p.Constraints.NYC = rand(1,4)*flow(:,'NYC') <= 10;
p.Constraints.BOS = rand(1,4)*flow(:,'BOS') <= 12;
p.Constraints.LAX = rand(1,4)*flow(:,'LAX') <= 35;
sol = solve(p);
```

```
Solving problem using intlinprog.
LP:                Optimal objective value is -1027.472366.

Heuristics:        Found 1 solution using ZI round.
                   Upper bound is -1027.233133.
                   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.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).
```

Find the optimal flow of oranges and berries to New York and Los Angeles.

```
[idxFruit,idxAirports] = findindex(flow, {'oranges','berries'}, {'NYC', 'LAX'})
```

```
idxFruit = 1×2

    2     4
```

```
idxAirports = 1×2

    1     3
```

```
orangeBerries = sol.flow(idxFruit, idxAirports)
```

orangeBerries = *2×2*

```
         0   980.0000
   70.0000          0
```

This display means that no oranges are going to NYC, 70 berries are going to NYC, 980 oranges are going to LAX, and no berries are going to LAX.

List the optimal flow of the following:

```
Fruit Airports

----- --------

Berries NYC

Apples BOS

Oranges LAX

idx = findindex(flow, {'berries', 'apples', 'oranges'}, {'NYC', 'BOS', 'LAX'})
```

idx = *1×3*

```
    4     5    10
```

```
optimalFlow = sol.flow(idx)
```

optimalFlow = *1×3*

```
   70.0000   28.0000  980.0000
```

This display means that 70 berries are going to NYC, 28 apples are going to BOS, and 980 oranges are going to LAX.

Create Initial Point with Named Index Variables

Create named index variables for a problem with various land types, potential crops, and plowing methods.

```
land = ["irr-good","irr-poor","dry-good","dry-poor"];
crops = ["wheat-lentil","wheat-corn","barley-chickpea","barley-lentil","wheat-onion","barley-oni
plow = ["tradition","mechanized"];
xcrop = optimvar('xcrop',land,crops,plow,'LowerBound',0);
```

Set the initial point to a zero array of the correct size.

```
x0.xcrop = zeros(size(xcrop));
```

Set the initial value to 3000 for the "wheat-onion" and "wheat-lentil" crops that are planted in any dry condition and are plowed traditionally.

```
[idxLand, idxCrop, idxPlough] = findindex(xcrop, ["dry-good","dry-poor"], ...
            ["wheat-onion","wheat-lentil"],"tradition");
x0.xcrop(idxLand,idxCrop,idxPlough) = 3000;
```

Set the initial values for the following three points.

```
Land      Crops           Method     Value
dry-good  wheat-corn      mechanized 2000
irr-poor  barley-onion    tradition  5000
irr-good  barley-chickpea mechanized 3500

idx = findindex(xcrop,...
    ["dry-good","irr-poor","irr-good"],...
    ["wheat-corn","barley-onion","barley-chickpea"],...
    ["mechanized","tradition","mechanized"]);
x0.xcrop(idx) = [2000,5000,3500];
```

Input Arguments

var — Optimization variable
OptimizationVariable object

Optimization variable, specified as an OptimizationVariable object. Create var using optimvar.

Example: var = optimvar('var',4,6)

strindex — Named index
cell array of character vectors | character vector | string vector | integer vector

Named index, specified as a cell array of character vectors, character vector, string vector, or integer vector. The number of strindex arguments must be the number of dimensions in var.

Example: ["small","medium","large"]

Data Types: double | char | string | cell

Output Arguments

numindex — Numeric index equivalent
integer vector

Numeric index equivalent, returned as an integer vector. The number of output arguments must be one of the following:

- The number of dimensions in var. Each output vector numindexj is the numeric equivalent of the corresponding input argument strindexj.
- One. In this case, the size of each input strindexj must be the same for all j, and the output satisfies the linear indexing criterion

  var(numindex(j)) = var(strindex1(j),...,strindexk(j)) for all j.

See Also
OptimizationVariable | solve

Topics
"Create Initial Point for Optimization with Named Index Variables" on page 10-50
"Named Index for Optimization Variables" on page 10-20

Introduced in R2018a

fminbnd

Find minimum of single-variable function on fixed interval

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` is a one-dimensional minimizer that 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 finite scalars, and $f(x)$ is a function that returns a scalar.

`x = fminbnd(fun,x1,x2)` returns a value `x` that is a local minimizer of the scalar valued function that is described in `fun` in the interval `x1 < x < x2`.

`x = fminbnd(fun,x1,x2,options)` minimizes with the optimization options specified in `options`. Use `optimset` to set these options.

`x = fminbnd(problem)` finds the minimum for `problem`, a structure described in `problem`.

`[x,fval] = fminbnd( ___ )`, for any input arguments, returns the value of the objective function computed in `fun` at the solution `x`.

`[x,fval,exitflag] = fminbnd( ___ )` additionally returns a value `exitflag` that describes the exit condition.

`[x,fval,exitflag,output] = fminbnd( ___ )` additionally returns a structure `output` that contains information about the optimization.

Examples

Minimum of `sin`

Find the point where the sin(x) function takes its minimum in the range $0 < x < 2\pi$.

```
fun = @sin;
x1 = 0;
x2 = 2*pi;
x = fminbnd(fun,x1,x2)

x = 4.7124
```

To display precision, this is the same as the correct value $x = 3\pi/2$.

```
3*pi/2
```

```
ans = 4.7124
```

Minimize a Function Specified by a File

Minimize a function that is specified by a separate function file. A function accepts a point x and returns a real scalar representing the value of the objective function at x.

Write the following function as a file, and save the file as `scalarobjective.m` on your MATLAB® path.

```
function f = scalarobjective(x)
f = 0;
for k = -10:10
    f = f + (k+1)^2*cos(k*x)*exp(-k^2/2);
end
```

Find the x that minimizes `scalarobjective` on the interval 1 <= x <= 3.

```
x = fminbnd(@scalarobjective,1,3)
```

```
x =

    2.0061
```

Minimize with Extra Parameter

Minimize a function when there is an extra parameter. The function $\sin(x - a)$ has a minimum that depends on the value of the parameter a. Create an anonymous function of x that includes the value of the parameter a. Minimize this function over the interval $0 < x < 2\pi$.

```
a = 9/7;
fun = @(x)sin(x-a);
x = fminbnd(fun,1,2*pi)
```

```
x = 5.9981
```

This answer is correct; the theoretical value is

```
3*pi/2 + 9/7
```

```
ans = 5.9981
```

For more information about including extra parameters, see "Parameterizing Functions".

Monitor Iterations

Monitor the steps `fminbnd` takes to minimize the $\sin(x)$ function for $0 < x < 2\pi$.

```
fun = @sin;
x1 = 0;
x2 = 2*pi;
options = optimset('Display','iter');
x = fminbnd(fun,x1,x2,options)
```

```
 Func-count     x          f(x)         Procedure
    1        2.39996      0.67549        initial
    2        3.88322     -0.67549        golden
    3        4.79993     -0.996171       golden
    4        5.08984     -0.929607       parabolic
    5        4.70582     -0.999978       parabolic
    6         4.7118           -1        parabolic
    7        4.71239           -1        parabolic
    8        4.71236           -1        parabolic
    9        4.71242           -1        parabolic

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-04

x = 4.7124
```

Find Minimum Location and Function Value

Find the location of the minimum of $\sin(x)$ and the value of the minimum for $0 < x < 2\pi$.

```
fun = @sin;
[x,fval] = fminbnd(fun,1,2*pi)

x = 4.7124

fval = -1.0000
```

Obtain All Information

Return all information about the `fminbnd` solution process by requesting all outputs. Also, monitor the solution process using a plot function.

```
fun = @sin;
x1 = 0;
x2 = 2*pi;
options = optimset('PlotFcns',@optimplotfval);
[x,fval,exitflag,output] = fminbnd(fun,x1,x2,options)
```
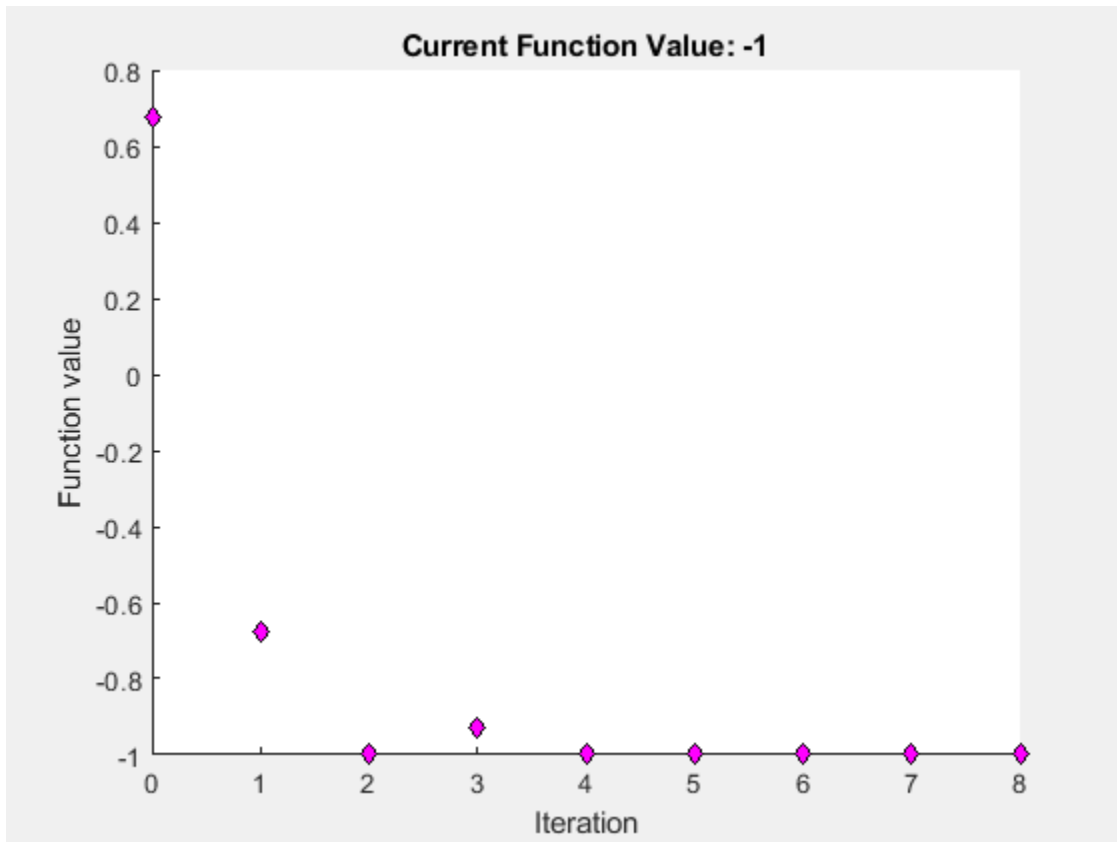
```
x = 4.7124

fval = -1.0000

exitflag = 1

output = struct with fields:
     iterations: 8
      funcCount: 9
      algorithm: 'golden section search, parabolic interpolation'
        message: 'Optimization terminated:...'
```

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 real scalar `x` and returns a real scalar `f` (the objective function evaluated at `x`).

Specify `fun` 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
```

You can also specify `fun` as a function handle for an anonymous function:

```
x = fminbnd(@(x)norm(x)^2,x1,x2);
```

Example: `fun = @(x)-x*exp(-3*x)`

Data Types: `char` | `function_handle` | `string`

x1 — Lower bound
finite real scalar

Lower bound, specified as a finite real scalar.

Example: `x1 = -3`

Data Types: `double`

x2 — Upper bound
finite real scalar

Upper bound, specified as a finite real scalar.

Example: `x2 = 5`

Data Types: `double`

options — Optimization options
structure such as `optimset` returns

Optimization options, specified as a structure such as `optimset` returns. You can use `optimset` to set or change the values of these fields in the options structure. See "Optimization Options Reference" on page 15-6 for detailed information.

`Display`	Level of display (see "Iterative Display" on page 3-14): • `'notify'` (default) displays output only if the function does not converge. • `'off'` or `'none'` displays no output. • `'iter'` displays output at each iteration. • `'final'` displays just the final output.
`FunValCheck`	Check whether objective function values are valid. The default `'off'` allows `fminbnd` to proceed when the objective function returns a value that is `complex` or `NaN`. The `'on'` setting throws an error when the objective function returns a value that is `complex` or `NaN`.
`MaxFunEvals`	Maximum number of function evaluations allowed, a positive integer. The default is `500`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9.
`MaxIter`	Maximum number of iterations allowed, a positive integer. The default is `500`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9.

| 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 and Plot Function Syntax" on page 15-27. |
| 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

Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27.

| TolX | Termination tolerance on `x`, a positive scalar. The default is `1e-4`. See "Tolerances and Stopping Criteria" on page 2-68. |

Example: `options = optimset('Display','iter')`

Data Types: `struct`

problem — Problem structure
structure

Problem structure, specified as a structure with the following fields.

Field Name	Entry
`objective`	Objective function
`x1`	Left endpoint
`x2`	Right endpoint
`solver`	`'fminbnd'`
`options`	Options structure such as returned by `optimset`

Data Types: `struct`

Output Arguments

x — Solution
real scalar

Solution, returned as a real scalar. 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 4-18.

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 fminbnd stopped
integer

Reason `fminbnd` stopped, returned as an integer.

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` — 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`	`'golden section search, parabolic interpolation'`
`message`	Exit message

Limitations

- The function to be minimized must be continuous.
- `fminbnd` might only give local solutions.
- `fminbnd` can exhibit slow convergence when the solution is on a boundary of the interval. In such a case, `fmincon` often gives faster and more accurate solutions.

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.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `fminbnd`.

References

[1] Forsythe, G. E., M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations*. Englewood Cliffs, NJ: Prentice Hall, 1976.

[2] Brent, Richard. P. *Algorithms for Minimization without Derivatives*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation:

- `fminbnd` does not support the problem structure argument.
- `fminbnd` ignores the `Display` option and does not give iterative display or an exit message. To check solution quality, examine the exit flag.
- The output structure does not include the `algorithm` or `message` fields.
- `fminbnd` ignores the `OutputFcn` and `PlotFcns` options.

See Also

Optimize | `fmincon` | `fminsearch` | `optimset`

Topics
"Create Function Handle"
"Anonymous Functions"

Introduced before R2006a

fmincon

Find minimum of constrained nonlinear multivariable function

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

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 2-31.

`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 2-57 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, `fmincon` throws an error. In this case, output `x` is `x0` and `fval` is `[]`.

For the default `'interior-point'` algorithm, `fmincon` sets components of `x0` that violate the bounds `lb ≤ x ≤ ub`, or are equal to a bound, to the interior of the bound region. For the `'trust-region-reflective'` algorithm, `fmincon` sets violating components to the interior of the bound region. For other algorithms, `fmincon` sets violating components to the closest bound. Components that respect the bounds are not changed. See "Iterations Can Violate Constraints" on page 2-33.

`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`, a structure described in `problem`.

`[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 3-24.

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, Solver-Based" on page 1-11.

```
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 $x(1) + 2x(2) ≤ 1$. 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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x = 1×2

    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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x = 1×2

    0.4149    0.1701
```

Minimize with 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, $x(1) \leq 1$, and $x(2) \leq 2$.

```
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.

```
x0 = (lb + ub)/2;
```

Solve the problem.

```
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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x = 1×2

    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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x = 1×2
10-6 ×

    0.4000    0.4000
```

To determine which solution is better, see "Obtain the Objective Function Value" on page 16-83.

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 value of the optimality tolerance,
and constraints are satisfied to within the 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.

```
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)
```

Iter	Func-count	Fval	Feasibility	Step Length	Norm of step	First-order optimality
0	3	1.000000e+00	0.000e+00	1.000e+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	6.172e+00
4	31	2.733703e-01	0.000e+00	1.000e+00	5.254e-02	5.705e-01
5	34	2.397111e-01	0.000e+00	1.000e+00	7.498e-02	3.164e+00
6	37	2.036002e-01	0.000e+00	1.000e+00	5.960e-02	3.106e+00
7	40	1.164353e-01	0.000e+00	1.000e+00	1.459e-01	1.059e+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	2.159e-09	1.511e-05

```
Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
the value of the step size tolerance and constraints are
satisfied to within the 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 Gradients and Hessians" on page 2-19. 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 named `rosenbrockwithgrad.m` on your MATLAB® path.

Create options to use the objective function gradient.

```matlab
options = optimoptions('fmincon','SpecifyObjectiveGradient',true);
```

Create the other inputs for the problem. Then call `fmincon`.

```matlab
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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.


x =

    1.0000    1.0000
```

Use a Problem Structure

Solve the same problem as in "Nondefault Options" on page 16-79 using a problem structure instead of separate arguments.

Create the options and a problem structure. See "problem" on page 16-0   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];
```

The nonlinear constraint function `unitdisk` appears at the end of this example on page 16-0   . Include the nonlinear constraint function in `problem`.

```
problem.nonlcon = @unitdisk;
```

Solve the problem.

```
x = fmincon(problem)
```

Iter	Func-count	Fval	Feasibility	Step Length	Norm of step	First-order optimality
0	3	1.000000e+00	0.000e+00	1.000e+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	6.172e+00
4	31	2.733703e-01	0.000e+00	1.000e+00	5.254e-02	5.705e-01
5	34	2.397111e-01	0.000e+00	1.000e+00	7.498e-02	3.164e+00
6	37	2.036002e-01	0.000e+00	1.000e+00	5.960e-02	3.106e+00
7	40	1.164353e-01	0.000e+00	1.000e+00	1.459e-01	1.059e+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	2.159e-09	1.511e-05

```
Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.
```

x = *1×2*

```
   0.7864    0.6177
```

The iterative display and solution are the same as in "Nondefault Options" on page 16-79.

The following code creates the `unitdisk` function.

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

Obtain the Objective Function Value

Call `fmincon` with the `fval` output to obtain the value of the objective function at the solution.

The "Minimize with Bound Constraints" on page 16-77 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 = (lb + ub)/2;
[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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x = 1×2

    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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x2 = 1×2
10⁻⁶ ×

    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.

```
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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.


x =

    0.7864    0.6177


fval =

    0.0457


exitflag =

     1


output =

  struct with fields:

         iterations: 24
```

```
        funcCount: 84
   constrviolation: 0
          stepsize: 6.9162e-06
         algorithm: 'interior-point'
     firstorderopt: 2.0234e-08
      cgiterations: 4
           message: '...'
       bestfeasible: [1x1 struct]
```

- 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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x =

    0.7864    0.6177
```

```
fval =

    0.0457


exitflag =

    1


output =

  struct with fields:

          iterations: 24
           funcCount: 84
       constrviolation: 0
            stepsize: 6.9162e-06
           algorithm: 'interior-point'
        firstorderopt: 2.0234e-08
         cgiterations: 4
             message: '...'
          bestfeasible: [1x1 struct]


lambda =

  struct with fields:

          eqlin: [0x1 double]
       eqnonlin: [0x1 double]
        ineqlin: [0x1 double]
          lower: [2x1 double]
          upper: [2x1 double]
      ineqnonlin: 0.1215


grad =

    -0.1911
    -0.1501


hessian =

   497.2838 -314.5553
  -314.5553  200.2369
```

- 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 3-24.

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 `SpecifyObjectiveGradient` option is set to `true`, as set by

```
options = optimoptions('fmincon','SpecifyObjectiveGradient',true)
```

then `fun` must return the gradient vector `g(x)` in the second output argument.

If you can also compute the Hessian matrix *and* the `HessianFcn` option is set to `'objective'` via `optimoptions` *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 "Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms" on page 2-21 for details.

If you can also compute the Hessian matrix and the `Algorithm` option is set to `'interior-point'`, there is a different way to pass the Hessian to `fmincon`. For more information, see "Hessian for fmincon interior-point algorithm" on page 2-21. For an example using Symbolic Math Toolbox to compute the gradient and Hessian, see "Calculate Gradients and Hessians Using Symbolic Math Toolbox™" on page 6-97.

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 2-23.

Example: `fun = @(x)sin(x(1))*cos(x(2))`

Data Types: `char` | `function_handle` | `string`

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.

- `'interior-point'` algorithm — If the `HonorBounds` option is `true` (default), `fmincon` resets `x0` components that are on or outside bounds `lb` or `ub` to values strictly between the bounds.
- `'trust-region-reflective'` algorithm — `fmincon` resets infeasible `x0` components to be feasible with respect to bounds or linear equalities.
- `'sqp'`, `'sqp-legacy'`, or `'active-set'` algorithm — `fmincon` resets `x0` components that are outside bounds to the values of the corresponding bounds.

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

$x_1 + 2x_2 \le 10$
$3x_1 + 4x_2 \le 20$
$5x_1 + 6x_2 \le 30$,

enter 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, use `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, consider these inequalities:

$x_1 + 2x_2 \le 10$
$3x_1 + 4x_2 \le 20$

$5x_1 + 6x_2 \le 30$.

Specify the inequalities by entering the following constraints.

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x components sum to 1 or less, use 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

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20$,

enter these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use 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 Me-by-N.

For example, consider these equalities:

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20$.

Specify the equalities by entering the following constraints.

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use `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 the number of elements in `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)`.

If there are fewer elements in `lb` than in `x0`, solvers issue a warning.

Example: To specify that all x components are positive, use `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 the number of elements in `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)`.

If there are fewer elements in `ub` than in `x0`, solvers issue a warning.

Example: To specify that all x components are less than 1, use `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 `SpecifyConstraintGradient` option is `true`, as set by

```
options = optimoptions('fmincon','SpecifyConstraintGradient',true)
```

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 2-37.

Data Types: `char` | `function_handle` | `string`

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 15-6 for detailed information.

Some options are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

All Algorithms

Algorithm	Choose the optimization algorithm:
	• `'interior-point'` (default)
	• `'trust-region-reflective'`
	• `'sqp'`
	• `'sqp-legacy'` (`optimoptions` only)
	• `'active-set'`
	For information on choosing the algorithm, see "Choosing the Algorithm" on page 2-6.
	The `trust-region-reflective` algorithm requires:
	• A gradient to be supplied in the objective function
	• `SpecifyObjectiveGradient` to be set to `true`
	• 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'`, `'sqp-legacy'`, and `'sqp'` algorithms are not large-scale. See "Large-Scale vs. Medium-Scale Algorithms" on page 2-10.
CheckGradients	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. Choices are `false` (default) or `true`.
	For `optimset`, the name is `DerivativeCheck` and the values are `'on'` or `'off'`. See "Current and Legacy Option Names" on page 15-22.
ConstraintTolerance	Tolerance on the constraint violation, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-68.
	For `optimset`, the name is `TolCon`. See "Current and Legacy Option Names" on page 15-22.
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 3-14):

- `'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.

FiniteDifferenceStepSize

Scalar or vector step size factor for finite differences. When you set FiniteDifferenceStepSize to a vector v, the forward finite differences `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 FiniteDifferenceStepSize expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.

For `optimset`, the name is `FinDiffRelStep`. See "Current and Legacy Option Names" on page 15-22.

FiniteDifferenceType

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 FiniteDifferenceType only when CheckGradients is set to `true`.

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 HonorBounds option is set to `false`.

For `optimset`, the name is `FinDiffType`. See "Current and Legacy Option Names" on page 15-22.

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`.
`MaxFunctionEvaluations`	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 2-68 and "Iterations and Function Counts" on page 3-9.
	For `optimset`, the name is `MaxFunEvals`. See "Current and Legacy Option Names" on page 15-22.
`MaxIterations`	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 2-68 and "Iterations and Function Counts" on page 3-9.
	For `optimset`, the name is `MaxIter`. See "Current and Legacy Option Names" on page 15-22.
`OptimalityTolerance`	Termination tolerance on the first-order optimality (a positive scalar). The default is `1e-6`. See "First-Order Optimality Measure" on page 3-11.
	For `optimset`, the name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration. Pass a function handle or a cell array of function handles. The default is none (`[]`). See "Output Function and Plot Function Syntax" on page 15-27.

PlotFcn	Plots various measures of progress while the algorithm executes; select from predefined plots or write your own. Pass a built-in plot function name, a function handle, or a cell array of built-in plot function names or function handles. For custom plot functions, pass function handles. The default is none (`[]`):

- `'optimplotx'` plots the current point
- `'optimplotfunccount'` plots the function count
- `'optimplotfval'` plots the function value
- `'optimplotfvalconstr'` plots the best feasible objective function value found as a line plot. The plot shows infeasible points as red, and feasible points as blue, using a feasibility tolerance of `1e-6`.
- `'optimplotconstrviolation'` plots the maximum constraint violation
- `'optimplotstepsize'` plots the step size
- `'optimplotfirstorderopt'` plots the first-order optimality measure

Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27.

For `optimset`, the name is `PlotFcns`. See "Current and Legacy Option Names" on page 15-22.

SpecifyConstraintGradient	Gradient for nonlinear constraint functions defined by the user. When set to the default, `false`, fmincon estimates gradients of the nonlinear constraints by finite differences. When set to `true`, fmincon expects the constraint function to have four outputs, as described in `nonlcon`. The `trust-region-reflective` algorithm does not accept nonlinear constraints.

For `optimset`, the name is `GradConstr` and the values are `'on'` or `'off'`. See "Current and Legacy Option Names" on page 15-22.

SpecifyObjectiveGradient	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, `false`, causes fmincon to estimate gradients using finite differences. Set to `true` to have fmincon use a user-defined gradient of the objective function. To use the `'trust-region-reflective'` algorithm, you must provide the gradient, and set `SpecifyObjectiveGradient` to `true`.

For `optimset`, the name is `GradObj` and the values are `'on'` or `'off'`. See "Current and Legacy Option Names" on page 15-22.

StepTolerance	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 2-68.
	For optimset, the name is TolX. See "Current and Legacy Option Names" on page 15-22.
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 CheckGradients 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

FunctionTolerance	Termination tolerance on the function value, a positive scalar. The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68.
	For optimset, the name is TolFun. See "Current and Legacy Option Names" on page 15-22.
HessianFcn	If [] (default), fmincon approximates the Hessian using finite differences, or uses a Hessian multiply function (with option HessianMultiplyFcn). If 'objective', fmincon uses a user-defined Hessian (defined in fun). See "Hessian as an Input" on page 16-103.
	For optimset, the name is HessFcn. See "Current and Legacy Option Names" on page 15-22.

HessianMultiplyFcn	Hessian multiply function, specified as a function handle. 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 2-57.

Note To use the `HessianMultiplyFcn` option, `HessianFcn` must be set to [], and `SubproblemAlgorithm` must be `'cg'` (default).

See "Hessian Multiply Function" on page 16-104. See "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-93 for an example.

For `optimset`, the name is `HessMult`. See "Current and Legacy Option Names" on page 15-22.

HessPattern	Sparsity pattern of the Hessian for finite differencing. Set `HessPattern(i,j) = 1` when you can have $\partial^2\text{fun}/\partial\text{x}(\text{i})\partial\text{x}(\text{j}) \neq 0$. 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 bound-constrained problems, and is `numberOfVariables` for equality-constrained problems. For more information, see "Preconditioned Conjugate Gradient Method" on page 6-21.

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.
`SubproblemAlgorithm`	Determines how the iteration step is calculated. The default, `'cg'`, takes a faster but less accurate step than `'factorization'`. See "fmincon Trust Region Reflective Algorithm" on page 6-19.
TolPCG	Termination tolerance on the PCG iteration, a positive scalar. The default is `0.1`.

Active-Set Algorithm

`FunctionTolerance`	Termination tolerance on the function value, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-68.
	For `optimset`, the name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.
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 $\lvert \Delta x(i) \rvert \le$ relLineSrchBnd·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 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

`HessianApproximation`	Chooses how `fmincon` calculates the Hessian (see "Hessian as an Input" on page 16-103). The choices are:

- `'bfgs'` (default)
- `'finite-difference'`
- `'lbfgs'`
- `{'lbfgs',Positive Integer}`

Note To use `HessianApproximation`, both `HessianFcn` and `HessianMultiplyFcn` must be empty entries (`[]`).

For `optimset`, the name is `Hessian` and the values are `'user-supplied'`, `'bfgs'`, `'lbfgs'`, `'fin-diff-grads'`, `'on'`, or `'off'`. See "Current and Legacy Option Names" on page 15-22.

HessianFcn	If [] (default), fmincon approximates the Hessian using finite differences, or uses a supplied HessianMultiplyFcn. If a function handle, fmincon uses HessianFcn to calculate the Hessian. See "Hessian as an Input" on page 16-103.
	For optimset, the name is HessFcn. See "Current and Legacy Option Names" on page 15-22.
HessianMultiplyFcn	User-supplied function that gives a Hessian-times-vector product (see "Hessian Multiply Function" on page 16-104). Pass a function handle.
	Note To use the HessianMultiplyFcn option, HessianFcn must be set to [], and SubproblemAlgorithm must be 'cg'.
	For optimset, the name is HessMult. See "Current and Legacy Option Names" on page 15-22.
HonorBounds	The default true ensures that bound constraints are satisfied at every iteration. Disable by setting to false.
	For optimset, the name is AlwaysHonorConstraints and the values are 'bounds' or 'none'. See "Current and Legacy Option Names" on page 15-22.
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	true causes the algorithm to normalize all constraints and the objective function. Disable by setting to the default false.
	For optimset, the values are 'obj-and-constr' or 'none'. See "Current and Legacy Option Names" on page 15-22.
SubproblemAlgorithm	Determines how the iteration step is calculated. The default, 'factorization', is usually faster than 'cg' (conjugate gradient), though 'cg' might be faster for large problems with dense Hessians. See "fmincon Interior Point Algorithm" on page 6-30.
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 and SQP Legacy Algorithms

| `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` | `true` causes the algorithm to normalize all constraints and the objective function. Disable by setting to the default `false`.

For `optimset`, the values are `'obj-and-constr'` or `'none'`. See "Current and Legacy Option Names" on page 15-22. |

Example: `options = optimoptions('fmincon','SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true)`

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.

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 4-18.

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.OptimalityTolerance`, and maximum constraint violation was less than `options.ConstraintTolerance`.
0	Number of iterations exceeded `options.MaxIterations` or number of function evaluations exceeded `options.MaxFunctionEvaluations`.
-1	Stopped by an output function or plot function.
-2	No feasible point was found.

All algorithms except `active-set`:

2	Change in `x` was less than `options.StepTolerance` and maximum constraint violation was less than `options.ConstraintTolerance`.

`trust-region-reflective` algorithm only:

3	Change in the objective function value was less than `options.FunctionTolerance` and maximum constraint violation was less than `options.ConstraintTolerance`.

`active-set` algorithm only:

4	Magnitude of the search direction was less than 2*`options.StepTolerance` and maximum constraint violation was less than `options.ConstraintTolerance`.
5	Magnitude of directional derivative in search direction was less than 2*`options.OptimalityTolerance` and maximum constraint violation was less than `options.ConstraintTolerance`.

`interior-point`, `sqp-legacy`, and `sqp` algorithms:

-3	Objective function at current iteration went below `options.ObjectiveLimit` and maximum constraint violation was less than `options.ConstraintTolerance`.

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
bestfeasible	Best (lowest objective function) feasible point encountered. A structure with these fields: • x • fval • firstorderopt • constrviolation If no feasible point is found, the `bestfeasible` field is empty. For this purpose, a point is feasible when the maximum of the constraint functions does not exceed `options.ConstraintTolerance`. The `bestfeasible` point can differ from the returned solution point x for a variety of reasons. For an example, see "Obtain Best Feasible Point" on page 6-117.
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 Output" on page 3-24.

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` and set the `'SpecifyObjectiveGradient'` option to `true`.

- 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 `HessianMultiplyFcn` function; one for `trust-region-reflective`, and another for `interior-point`. See "Including Hessians" on page 2-21.

  - 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 current point `x` and the Lagrange multiplier structure `lambda`.

- When the problem is infeasible, `fmincon` attempts to minimize the maximum constraint value.

More About

Hessian as an Input

`fmincon` uses a Hessian as an optional input. This Hessian is the matrix of second derivatives of the Lagrangian (see "Equation 3-1"), namely,

$$\nabla_{xx}^2 L(x, \lambda) = \nabla^2 f(x) + \sum \lambda_i \nabla^2 c_i(x) + \sum \lambda_i \nabla^2 ceq_i(x). \tag{16-1}$$

For details of how to supply a Hessian to the `trust-region-reflective` or `interior-point` algorithms, see "Including Hessians" on page 2-21.

The `active-set` and `sqp` algorithms do not accept an input Hessian. They compute a quasi-Newton approximation to the Hessian of the Lagrangian.

The `interior-point` algorithm has several choices for the `'HessianApproximation'` option; see "Choose Input Hessian Approximation for interior-point fmincon" on page 2-24:

- `'bfgs'` — `fmincon` calculates the Hessian by a dense quasi-Newton approximation. This is the default Hessian approximation.

- `'lbfgs'` — `fmincon` calculates the Hessian by a limited-memory, large-scale quasi-Newton approximation. The default memory, 10 iterations, is used.

- `{'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.

- `'finite-difference'` — `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). Set the `'SpecifyObjectiveGradient'` option

to `true` and, if applicable, the `'SpecifyConstraintGradient'` option to `true`. You must set the `'SubproblemAlgorithm'` to `'cg'`.

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. For details, see "Hessian Multiply Function" on page 2-23.

Algorithms

Choosing the Algorithm

For help choosing the algorithm, see "fmincon Algorithms" on page 2-6. To set the algorithm, use `optimoptions` to create `options`, and use the `'Algorithm'` name-value pair.

The rest of this section gives brief summaries or pointers to information about each algorithm.

Interior-Point Optimization

This algorithm is described in "fmincon Interior Point Algorithm" on page 6-30. There is more extensive description in [1], [41], and [9].

SQP and SQP-Legacy Optimization

The fmincon `'sqp'` and `'sqp-legacy'` algorithms are similar to the `'active-set'` algorithm described in "Active-Set Optimization" on page 16-104. "fmincon SQP Algorithm" on page 6-29 describes the main differences. In summary, these differences are:

- "Strict Feasibility With Respect to Bounds" on page 6-29
- "Robustness to Non-Double Results" on page 6-29
- "Refactored Linear Algebra Routines" on page 6-29
- "Reformulated Feasibility Routines" on page 6-29

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 6-22 describes this algorithm in detail.

See also "SQP Implementation" on page 6-25 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 6-19.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `fmincon`.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `fmincon` supports code generation using either the `codegen` function or the MATLAB Coder app. You must have a MATLAB Coder license to generate code.
- The target hardware must support standard double-precision floating-point computations. You cannot generate code for single-precision or fixed-point computations.

- Code generation targets do not use the same math kernel libraries as MATLAB solvers. Therefore, code generation solutions can vary from solver solutions, especially for poorly conditioned problems.

- All code for generation must be MATLAB code. In particular, you cannot use a custom black-box function as an objective function for `fmincon`. You can use `coder.ceval` to evaluate a custom function coded in C or C++. However, the custom function must be called in a MATLAB function.

- `fmincon` does not support the `problem` argument for code generation.

  ```
  [x,fval] = fmincon(problem) % Not supported
  ```

- You must specify the objective function and any nonlinear constraint function by using function handles, not strings or character names.

  ```
  x = fmincon(@fun,x0,A,b,Aeq,beq,lb,ub,@nonlcon) % Supported
  % Not supported: fmincon('fun',...) or fmincon("fun",...)
  ```

- All `fmincon` input matrices such as `A`, `Aeq`, `lb`, and `ub` must be full, not sparse. You can convert sparse matrices to full by using the `full` function.

- The `lb` and `ub` arguments must have the same number of entries as the `x0` argument or must be empty `[]`.

- For advanced code optimization involving embedded processors, you also need an Embedded Coder license.

- You must include options for `fmincon` and specify them using `optimoptions`. The options must include the `Algorithm` option, set to `'sqp'` or `'sqp-legacy'`.

  ```
  options = optimoptions('fmincon','Algorithm','sqp');
  [x,fval,exitflag] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options);
  ```

- Code generation supports these options:

  - `Algorithm` — Must be `'sqp'` or `'sqp-legacy'`
  - `ConstraintTolerance`
  - `FiniteDifferenceStepSize`
  - `FiniteDifferenceType`
  - `MaxFunctionEvaluations`
  - `MaxIterations`
  - `ObjectiveLimit`
  - `OptimalityTolerance`
  - `ScaleProblem`
  - `SpecifyConstraintGradient`
  - `SpecifyObjectiveGradient`
  - `StepTolerance`
  - `TypicalX`

- Generated code has limited error checking for options. The recommended way to update an option is to use `optimoptions`, not dot notation.

  ```
  opts = optimoptions('fmincon','Algorithm','sqp');
  opts = optimoptions(opts,'MaxIterations',1e4); % Recommended
  opts.MaxIterations = 1e4; % Not recommended
  ```

- Do not load options from a file. Doing so can cause code generation to fail. Instead, create options in your code.
- Usually, if you specify an option that is not supported, the option is silently ignored during code generation. However, if you specify a plot function or output function by using dot notation, code generation can issue an error. For reliability, specify only supported options.
- Because output functions and plot functions are not supported, `fmincon` does not return the exit flag –1.
- Code generated from `fmincon` does not contain the `bestfeasible` field in a returned `output` structure.

For an example, see "Code Generation for Optimization Basics" on page 6-127.

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the `'UseParallel'` option to `true`.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see "Using Parallel Computing in Optimization Toolbox" on page 14-5.

See Also
Optimize | `fminbnd` | `fminsearch` | `fminunc` | `optimoptions`

Topics
"Solver-Based Nonlinear Optimization"
"Solver-Based Optimization Problem Setup"
"Constrained Nonlinear Optimization Algorithms" on page 6-19

Introduced before R2006a

fminimax

Solve minimax constraint problem

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,exitflag,output] = fminimax(___)
[x,fval,maxfval,exitflag,output,lambda] = fminimax(___)
```

Description

`fminimax` seeks a point that minimizes the maximum of a set of objective functions.

The problem includes any type of constraint. In detail, `fminimax` seeks the minimum of a problem specified by

$$\underset{x}{\operatorname{minmax}}\, F_i(x) \ \text{ such that } \begin{cases} c(x) \le 0 \\ ceq(x) = 0 \\ A \cdot x \le b \\ Aeq \cdot x = beq \\ lb \le x \le ub \end{cases}$$

where 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 2-31.

You can also solve max-min problems with `fminimax`, using the identity

$$\underset{x}{\operatorname{maxmin}}\, F_i(x) = -\underset{x}{\operatorname{minmax}}(-F_i(x)).$$

You can solve problems of the form

$$\underset{x}{\operatorname{minmax}} |F_i(x)|$$

by using the `AbsoluteMaxObjectiveCount` option; see "Solve Minimax Problem Using Absolute Value of One Objective" on page 16-114.

`x = fminimax(fun,x0)` starts at `x0` and finds a minimax solution `x` to the functions described in `fun`.

Note "Passing Extra Parameters" on page 2-57 explains how to pass extra parameters to the objective functions and nonlinear constraint functions, if necessary.

`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. If no inequalities exist, set `A = []` and `b = []`.

`x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub)` solves the minimax problem subject to the bounds `lb ≤ x ≤ ub`. If no equalities exist, set `Aeq = []` and `beq = []`. If `x(i)` is unbounded below, set `lb(i) = −Inf`; if `x(i)` is unbounded above, set `ub(i) = Inf`.

Note See "Iterations Can Violate Constraints" on page 2-33.

Note If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the output `fval` is `[]`.

`x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)` solves the minimax problem subject to the nonlinear inequalities `c(x)` or equalities `ceq(x)` defined in `nonlcon`. The function optimizes such that `c(x) ≤ 0` and `ceq(x) = 0`. If no bounds exist, set `lb = []` or `ub = []`, or both.

`x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)` solves the minimax problem with the optimization options specified in `options`. Use `optimoptions` to set these options.

`x = fminimax(problem)` solves the minimax problem for `problem`, a structure described in `problem`.

`[x,fval] = fminimax( ___ )`, for any syntax, returns the values of the objective functions computed in `fun` at the solution `x`.

`[x,fval,maxfval,exitflag,output] = fminimax( ___ )` additionally returns the maximum value of the objective functions at the solution `x`, a value `exitflag` that describes the exit condition of `fminimax`, and a structure `output` with information about the optimization process.

`[x,fval,maxfval,exitflag,output,lambda] = fminimax( ___ )` additionally returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

Examples

Minimize Maximum of `sin` and `cos`

Create a plot of the `sin` and `cos` functions and their maximum over the interval `[−pi,pi]`.

```
t = linspace(-pi,pi);
plot(t,sin(t),'r-')
hold on
plot(t,cos(t),'b-');
plot(t,max(sin(t),cos(t)),'ko')
legend('sin(t)','cos(t)','max(sin(t),cos(t))','Location','NorthWest')
```
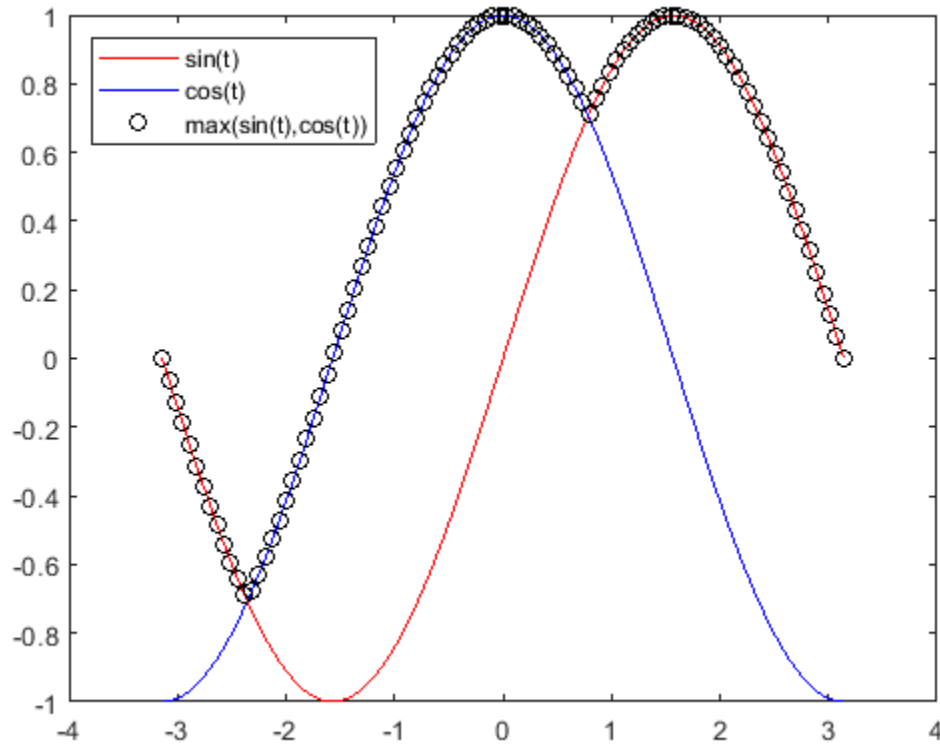
The plot shows two local minima of the maximum, one near 1, and the other near –2. Find the minimum near 1.

```
fun = @(x)[sin(x);cos(x)];
x0 = 1;
x1 = fminimax(fun,x0)
```

Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than twice the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.

x1 = 0.7854

Find the minimum near –2.

```
x0 = -2;
x2 = fminimax(fun,x0)
```

Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than twice the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.

x2 = -2.3562

Solve Linearly Constrained Minimax Problem

The objective functions for this example are linear plus constants. For a description and plot of the objective functions, see "Compare fminimax and fminunc" on page 8-6.

Set the objective functions as three linear functions of the form $dot(x, v) + v_0$ for three vectors v and three constants v_0.

```
a = [1;1];
b = [-1;1];
c = [0;-1];
a0 = 2;
b0 = -3;
c0 = 4;
fun = @(x)[x*a+a0,x*b+b0,x*c+c0];
```

Find the minimax point subject to the inequality `x(1) + 3*x(2) <= −4`.

```
A = [1,3];
b = -4;
x0 = [-1,-2];
x = fminimax(fun,x0,A,b)
```

```
Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = 1×2

   -5.8000    0.6000
```

Solve Bound-Constrained Minimax Problem

The objective functions for this example are linear plus constants. For a description and plot of the objective functions, see "Compare fminimax and fminunc" on page 8-6.

Set the objective functions as three linear functions of the form $dot(x, v) + v_0$ for three vectors v and three constants v_0.

```
a = [1;1];
b = [-1;1];
c = [0;-1];
a0 = 2;
b0 = -3;
c0 = 4;
fun = @(x)[x*a+a0,x*b+b0,x*c+c0];
```

Set bounds that $−2 <= x(1) <= 2$ and $−1 <= x(2) <= 1$ and solve the minimax problem starting from `[0,0]`.

```
lb = [-2,-1];
ub = [2,1];
x0 = [0,0];
A = []; % No linear constraints
b = [];
Aeq = [];
beq = [];
[x,fval] = fminimax(fun,x0,A,b,Aeq,beq,lb,ub)

Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = 1×2

  -0.0000    1.0000


fval = 1×3

   3.0000   -2.0000    3.0000
```

In this case, the solution is not unique. Many points satisfy the constraints and have the same minimax value. Plot the surface representing the maximum of the three objective functions, and plot a red line showing the points that have the same minimax value.

```
[X,Y] = meshgrid(linspace(-2,2),linspace(-1,1));
Z = max(fun([X(:),Y(:)]),[],2);
Z = reshape(Z,size(X));
surf(X,Y,Z,'LineStyle','none')
view(-118,28)
hold on
line([-2,0],[1,1],[3,3],'Color','r','LineWidth',8)
hold off
```
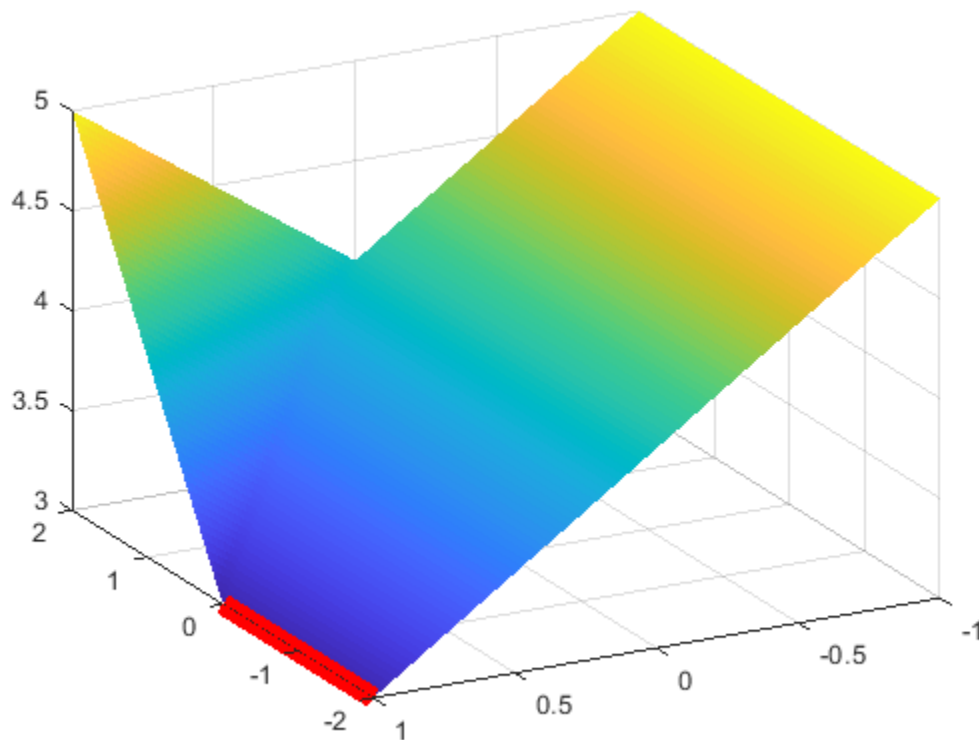
Find Minimax Subject to Nonlinear Constraints

The objective functions for this example are linear plus constants. For a description and plot of the objective functions, see "Compare fminimax and fminunc" on page 8-6.

Set the objective functions as three linear functions of the form $\text{dot}(x, v) + v_0$ for three vectors v and three constants v_0.

```
a = [1;1];
b = [-1;1];
c = [0;-1];
a0 = 2;
b0 = -3;
c0 = 4;
fun = @(x)[x*a+a0,x*b+b0,x*c+c0];
```

The `unitdisk` function represents the nonlinear inequality constraint $\|x\|^2 \le 1$.

```
type unitdisk
```

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

Solve the minimax problem subject to the `unitdisk` constraint, starting from x0 = [0,0].

```
x0 = [0,0];
A = []; % No other constraints
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
nonlcon = @unitdisk;
x = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
```

```
Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = 1×2

   -0.0000    1.0000
```

Solve Minimax Problem Using Absolute Value of One Objective

fminimax can minimize the maximum of either $F_i(x)$ or $|F_i(x)|$ for the first several values of i by using the AbsoluteMaxObjectiveCount option. To minimize the absolute values of k of the objectives, arrange the objective function values so that $F_1(x)$ through $F_k(x)$ are the objectives for absolute minimization, and set the AbsoluteMaxObjectiveCount option to k.

In this example, minimize the maximum of sin and cos, specify sin as the first objective, and set AbsoluteMaxObjectiveCount to 1.

```
fun = @(x)[sin(x),cos(x)];
options = optimoptions('fminimax','AbsoluteMaxObjectiveCount',1);
x0 = 1;
A = []; % No constraints
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];
nonlcon = [];
x1 = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

```
Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x1 = 0.7854
```

Try starting from x0 = −2.

```
x0 = -2;
x2 = fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

```
Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x2 = -3.1416
```

Plot the function.

```
t = linspace(-pi,pi);
plot(t,max(abs(sin(t)),cos(t)))
```
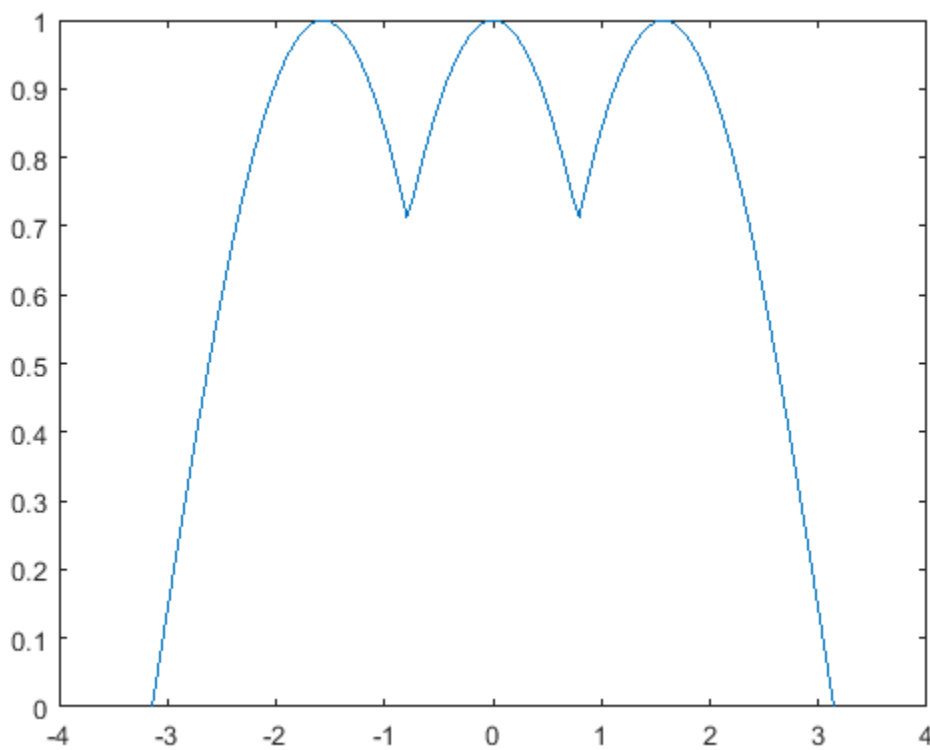

To see the effect of the `AbsoluteMaxObjectiveCount` option, compare this plot to the plot in the example "Minimize Maximum of sin and cos" on page 16-109.

Obtain Minimax Value

Obtain both the location of the minimax point and the value of the objective functions. For a description and plot of the objective functions, see "Compare fminimax and fminunc" on page 8-6.

Set the objective functions as three linear functions of the form $dot(x, v) + v_0$ for three vectors v and three constants v_0.

```
a = [1;1];
b = [-1;1];
c = [0;-1];
a0 = 2;
b0 = -3;
c0 = 4;
fun = @(x)[x*a+a0,x*b+b0,x*c+c0];
```

Set the initial point to `[0,0]` and find the minimax point and value.

```
x0 = [0,0];
[x,fval] = fminimax(fun,x0)
```

```
Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = 1×2

   -2.5000    2.2500


fval = 1×3

    1.7500    1.7500    1.7500
```

All three objective functions have the same value at the minimax point. Unconstrained problems typically have at least two objectives that are equal at the solution, because if a point is not a local minimum for any objective and only one objective has the maximum value, then the maximum objective can be lowered.

Obtain All Minimax Outputs

The objective functions for this example are linear plus constants. For a description and plot of the objective functions, see "Compare fminimax and fminunc" on page 8-6.

Set the objective functions as three linear functions of the form $dot(x, v) + v_0$ for three vectors v and three constants v_0.

```
a = [1;1];
b = [-1;1];
c = [0;-1];
a0 = 2;
b0 = -3;
c0 = 4;
fun = @(x)[x*a+a0,x*b+b0,x*c+c0];
```

Find the minimax point subject to the inequality `x(1) + 3*x(2) <= −4`.

```
A = [1,3];
b = -4;
x0 = [-1,-2];
```

Set options for iterative display, and obtain all solver outputs.

```
options = optimoptions('fminimax','Display','iter');
Aeq = []; % No other constraints
beq = [];
lb = [];
ub = [];
nonlcon = [];
[x,fval,maxfval,exitflag,output,lambda] =...
    fminimax(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

		Objective	Max	Line search	Directional	
Iter	F-count	value	constraint	steplength	derivative	Procedure
0	4	0	6			
1	9	5	0	1	0.981	
2	14	4.889	8.882e-16	1	-0.302	Hessian modified twice
3	19	3.4	8.132e-09	1	-0.302	Hessian modified twice

```
Local minimum possible. Constraints satisfied.

fminimax stopped because the size of the current search direction is less than
twice the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

x = 1×2

   -5.8000    0.6000


fval = 1×3

   -3.2000    3.4000    3.4000


maxfval = 3.4000

exitflag = 4

output = struct with fields:
        iterations: 4
         funcCount: 19
       lssteplength: 1
          stepsize: 6.0684e-10
         algorithm: 'active-set'
      firstorderopt: []
     constrviolation: 8.1323e-09
           message: '...'


lambda = struct with fields:
         lower: [2x1 double]
         upper: [2x1 double]
         eqlin: [0x1 double]
      eqnonlin: [0x1 double]
        ineqlin: 0.2000
     ineqnonlin: [0x1 double]
```

Examine the returned information:

- Two objective function values are equal at the solution.
- The solver converges in 4 iterations and 19 function evaluations.
- The `lambda.ineqlin` value is nonzero, indicating that the linear constraint is active at the solution.

Input Arguments

fun — Objective functions
function handle | function name

Objective functions, specified as a function handle or function name. `fun` is a function that accepts a vector `x` and returns a vector `F`, the objective functions evaluated at `x`. You can specify the function `fun` as a function handle for a function file:

```
x = fminimax(@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 = fminimax(@(x)sin(x.*x),x0,goal,weight);
```

If the user-defined values for `x` and `F` are arrays, `fminimax` converts them to vectors using linear indexing (see "Array Indexing").

To minimize the worst-case absolute values of some elements of the vector $F(x)$ (that is, min{max abs{$F(x)$} }), partition those objectives into the first elements of F and use `optimoptions` to set the `AbsoluteMaxObjectiveCount` option to the number of these objectives. These objectives *must* be partitioned into the first elements of the vector F returned by `fun`. For an example, see "Solve Minimax Problem Using Absolute Value of One Objective" on page 16-114.

Assume that the gradients of the objective functions can also be computed *and* the `SpecifyObjectiveGradient` option is `true`, as set by:

```
options = optimoptions('fminimax','SpecifyObjectiveGradient',true)
```

In this case, the function `fun` must return, in the second output argument, the gradient values 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)` (that is, the jth column of G is the gradient of the jth objective function `F(j)`). If you define F as an array, then the preceding discussion applies to `F(:)`, the linear ordering of the F array. In any case, G is a 2-D matrix.

Note Setting `SpecifyObjectiveGradient` to `true` is effective only when the problem has no nonlinear constraint, or when the problem has a nonlinear constraint with `SpecifyConstraintGradient` set to `true`. 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.

Data Types: `char` | `string` | `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 `x0` and the 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

$x_1 + 2x_2 \leq 10$
$3x_1 + 4x_2 \leq 20$
$5x_1 + 6x_2 \leq 30,$

enter 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, use `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, consider these inequalities:

$x_1 + 2x_2 \leq 10$
$3x_1 + 4x_2 \leq 20$

$5x_1 + 6x_2 \le 30.$

Specify the inequalities by entering the following constraints.

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x components sum to 1 or less, use `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

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20,$

enter these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use `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 `Me`-by-N.

For example, consider these equalities:

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20.$

Specify the equalities by entering the following constraints.

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use `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 the number of elements in `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)`.

If there are fewer elements in `lb` than in `x0`, solvers issue a warning.

Example: To specify that all x components are positive, use `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 the number of elements in `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)`.

If there are fewer elements in `ub` than in `x0`, solvers issue a warning.

Example: To specify that all x components are less than 1, use `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. `fminimax` attempts to satisfy `c(x) <= 0` for all entries of `c`.
- `ceq(x)` is the array of nonlinear equality constraints at x. `fminimax` attempts to satisfy `ceq(x) = 0` for all entries of `ceq`.

For example,

`x = fminimax(@myfun,x0,...,@mycon)`

where `mycon` is a MATLAB function such as the following:

```
function [c,ceq] = mycon(x)
c = ...     % Compute nonlinear inequalities at x.
ceq = ...   % Compute nonlinear equalities at x.
```

Suppose that the gradients of the constraints can also be computed *and* the `SpecifyConstraintGradient` option is `true`, as set by:

```
options = optimoptions('fminimax','SpecifyConstraintGradient',true)
```

In this case, 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)`. See "Nonlinear Constraints" on page 2-37 for an explanation of 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)` (that is, 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)` (that is, the `j`th column of `GCeq` is the gradient of the `j`th equality constraint `ceq(j)`).

Note Setting `SpecifyConstraintGradient` to `true` is effective only when `SpecifyObjectiveGradient` is set to `true`. 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 accept only inputs of type `double`, user-supplied objective and nonlinear constraint functions must return outputs of type `double`.

See "Passing Extra Parameters" on page 2-57 for an explanation of how to parameterize the nonlinear constraint function `nonlcon`, if necessary.

Data Types: `char` | `function_handle` | `string`

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 are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

For details about options that have different names for `optimset`, see "Current and Legacy Option Names" on page 15-22.

Option	Description
AbsoluteMaxObjectiveCount	Number of elements of $F_i(x)$ for which to minimize the absolute value of F_i. See "Solve Minimax Problem Using Absolute Value of One Objective" on page 16-114.
	For optimset, the name is MinAbsMax.
ConstraintTolerance	Termination tolerance on the constraint violation (a positive scalar). The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68.
	For optimset, the name is TolCon.
Diagnostics	Display of diagnostic information about the function to be minimized or solved. The choices are 'on' or 'off' (the default).
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 3-14):

* '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.

Option	Description

`FiniteDifferenceStepSize`	Scalar or vector step size factor for finite differences. When you set `FiniteDifferenceStepSize` to a vector `v`, the forward finite differences `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 `FiniteDifferenceStepSize` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.
	
	For `optimset`, the name is `FinDiffRelStep`.

`FiniteDifferenceType`	Type of finite differences used to estimate gradients, either `'forward'` (default) or `'central'` (centered). `'central'` takes twice as many function evaluations, but is generally more accurate. The algorithm is careful to obey bounds when estimating both types of finite differences. For example, it might take a backward difference, rather than a forward difference, to avoid evaluating at a point outside the bounds. For `optimset`, the name is `FinDiffType`.
`FunctionTolerance`	Termination tolerance on the function value (a positive scalar). The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-68. For `optimset`, the name is `TolFun`.
FunValCheck	Check that signifies whether the objective function and constraint 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.
`MaxFunctionEvaluations`	Maximum number of function evaluations allowed (a positive integer). The default is `100*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9. For `optimset`, the name is `MaxFunEvals`.
`MaxIterations`	Maximum number of iterations allowed (a positive integer). The default is `400`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9. For `optimset`, the name is `MaxIter`.

Option	Description
MaxSQPIter	Maximum number of SQP iterations allowed (a positive integer). The default is `10*max(numberOfVariables, numberOfInequalities + numberOfBounds)`.
MeritFunction	If this option is set to `'multiobj'` (the default), use the goal attainment or minimax merit function. If this option is set to `'singleobj'`, use the fmincon merit function.
OptimalityTolerance	Termination tolerance on the first-order optimality (a positive scalar). The default is `1e-6`. See "First-Order Optimality Measure" on page 3-11. For `optimset`, the name is `TolFun`.
OutputFcn	One or more user-defined functions that an optimization function calls at each iteration. Pass a function handle or a cell array of function handles. The default is none (`[]`). See "Output Function and Plot Function Syntax" on page 15-27.
PlotFcn	Plots showing various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a name, function handle, or cell array of names or function handles. For custom plot functions, pass function handles. The default is none (`[]`). • `'optimplotx'` plots the current point. • `'optimplotfunccount'` plots the function count. • `'optimplotfval'` plots the objective function values. • `'optimplotconstrviolation'` plots the maximum constraint violation. • `'optimplotstepsize'` plots the step size. Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27. For `optimset`, the name is `PlotFcns`.
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· max($|x(i)|, |typicalx(i)|$). This option provides control over the magnitude of the displacements in x when the solver takes steps that are too large. The default is none (`[]`).
RelLineSrchBndDuration	Number of iterations for which the bound specified in `RelLineSrchBnd` should be active. The default is `1`.

Option	Description
SpecifyConstraintGradient	Gradient for nonlinear constraint functions defined by the user. When this option is set to `true`, fminimax expects the constraint function to have four outputs, as described in nonlcon. When this option is set to `false` (the default), fminimax estimates gradients of the nonlinear constraints using finite differences.
	For optimset, the name is GradConstr and the values are 'on' or 'off'.
SpecifyObjectiveGradient	Gradient for the objective function defined by the user. Refer to the description of fun to see how to define the gradient. Set this option to `true` to have fminimax use a user-defined gradient of the objective function. The default, `false`, causes fminimax to estimate gradients using finite differences.
	For optimset, the name is GradObj and the values are 'on' or 'off'.
StepTolerance	Termination tolerance on x (a positive scalar). The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68.
	For optimset, the name is TolX.
TolConSQP	Termination tolerance on the inner iteration SQP constraint violation (a positive scalar). The default is `1e-6`.
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 fminimax function uses TypicalX for scaling finite differences for gradient estimation.
UseParallel	Option for using parallel computing. When this option is set to `true`, fminimax estimates gradients in parallel. The default is `false`. See "Parallel Computing".

Example: `optimoptions('fminimax','PlotFcn','optimplotfval')`

problem — Problem structure
structure

Problem structure, specified as a structure with the fields in this table.

Field Name	Entry
objective	Objective function fun
x0	Initial point for x
Aineq	Matrix for linear inequality constraints
bineq	Vector for linear inequality constraints
Aeq	Matrix for linear equality constraints

Field Name	Entry
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

You must supply at least the objective, x0, solver, and options fields in the problem structure.

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 4-18.

fval — Objective function values at solution
real array

Objective function values at the solution, returned as a real array. Generally, fval = fun(x).

maxfval — Maximum of objective function values at solution
real scalar

Maximum of the objective function values at the solution, returned as a real scalar. maxfval = max(fval(:)).

exitflag — Reason fminimax stopped
integer

Reason fminimax stopped, returned as an integer.

1	Function converged to a solution x
4	Magnitude of the search direction was less than the specified tolerance, and the constraint violation was less than options.ConstraintTolerance
5	Magnitude of the directional derivative was less than the specified tolerance, and the constraint violation was less than options.ConstraintTolerance
0	Number of iterations exceeded options.MaxIterations or the number of function evaluations exceeded options.MaxFunctionEvaluations
-1	Stopped by an output function or plot function
-2	No feasible point was found.

output — **Information about optimization process**
structure

Information about the optimization process, returned as a structure with the fields in this table.

iterations	Number of iterations taken
funcCount	Number of function evaluations
lssteplength	Size of the line search step relative to the search direction
constrviolation	Maximum of the constraint functions
stepsize	Length of the last displacement in x
algorithm	Optimization algorithm used
firstorderopt	Measure of first-order optimality
message	Exit message

lambda — **Lagrange multipliers at solution**
structure

Lagrange multipliers at the solution, returned as a structure with the fields in this table.

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

Algorithms

fminimax solves a minimax problem by converting it into a goal attainment problem, and then solving the converted goal attainment problem using fgoalattain. The conversion sets all goals to 0 and all weights to 1. See "Equation 8-1" in "Multiobjective Optimization Algorithms" on page 8-2.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for fminimax.

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the 'UseParallel' option to true.

options = optimoptions('*solvername*','UseParallel',true)

For more information, see "Using Parallel Computing in Optimization Toolbox" on page 14-5.

See Also
Optimize | fgoalattain | optimoptions

Topics
"Create Function Handle"
"Multiobjective Optimization"

Introduced before R2006a

fminsearch

Find minimum of unconstrained multivariable function using derivative-free method

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

Nonlinear programming solver. Searches for the minimum of a problem specified by

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

$f(x)$ is a function that returns a scalar, and x is a vector or a matrix; see "Matrix Arguments" on page 2-31.

`x = fminsearch(fun,x0)` starts at the point `x0` and attempts to find a local minimum `x` of the function described in `fun`.

`x = fminsearch(fun,x0,options)` minimizes with the optimization options specified in the structure `options`. Use `optimset` to set these options.

`x = fminsearch(problem)` finds the minimum for `problem`, a structure described in `problem`.

`[x,fval] = fminsearch( ___ )`, for any previous input syntax, returns in `fval` the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fminsearch( ___ )` additionally returns a value `exitflag` that describes the exit condition.

`[x,fval,exitflag,output] = fminsearch( ___ )` additionally returns a structure `output` with information about the optimization process.

Examples

Minimize Rosenbrock's Function

Minimize Rosenbrock's function, a notoriously difficult optimization problem for many algorithms:

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

The function is minimized at the point `x = [1,1]` with minimum value `0`.

Set the start point to `x0 = [-1.2,1]` and minimize Rosenbrock's function using `fminsearch`.

```
fun = @(x)100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
x0 = [-1.2,1];
x = fminsearch(fun,x0)
```

x = *1×2*

```
    1.0000    1.0000
```

Monitor Optimization Process

Set options to monitor the process as `fminsearch` attempts to locate a minimum.

Set options to plot the objective function at each iteration.

```
options = optimset('PlotFcns',@optimplotfval);
```

Set the objective function to Rosenbrock's function,

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

The function is minimized at the point `x = [1,1]` with minimum value `0`.

Set the start point to `x0 = [-1.2,1]` and minimize Rosenbrock's function using `fminsearch`.

```
fun = @(x)100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
x0 = [-1.2,1];
x = fminsearch(fun,x0,options)
```
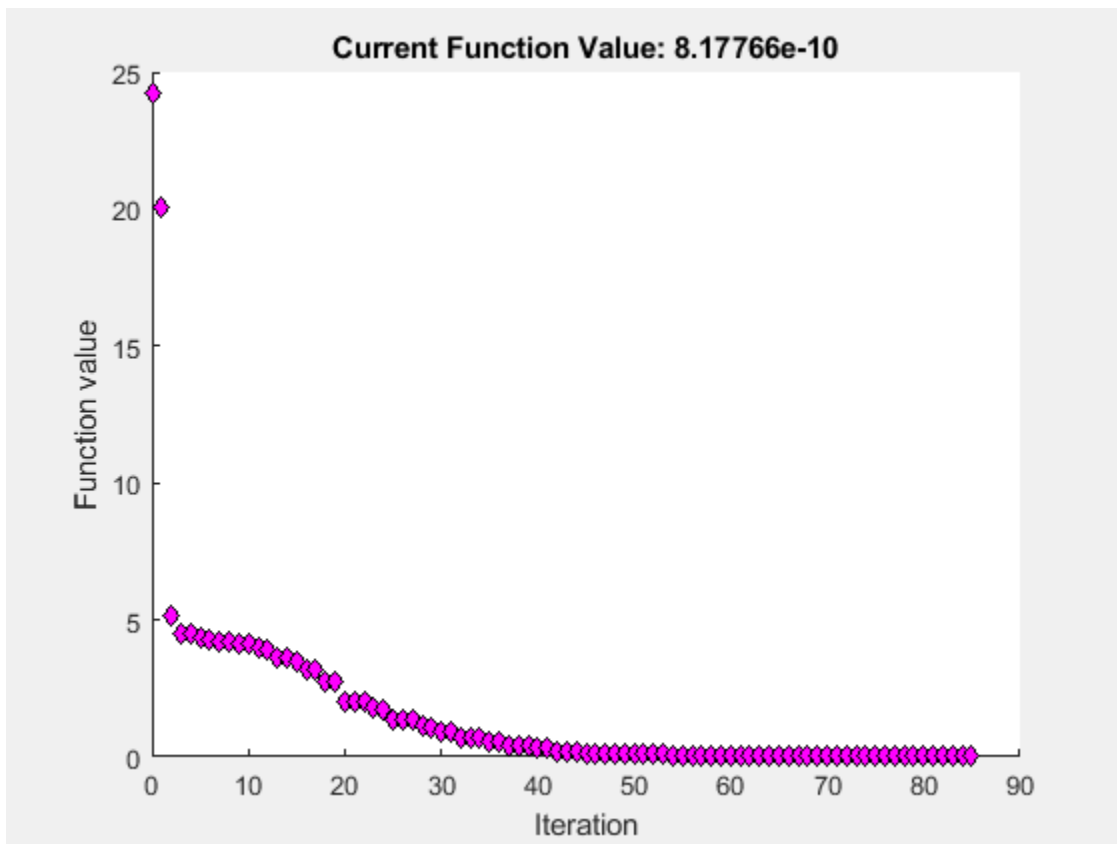
```
x = 1×2

    1.0000    1.0000
```

Minimize a Function Specified by a File

Minimize an objective function whose values are given by executing a file. A function file must accept a real vector x and return a real scalar that is the value of the objective function.

Copy the following code and include it as a file named objectivefcn1.m on your MATLAB® path.

```
function f = objectivefcn1(x)
f = 0;
for k = -10:10
    f = f + exp(-(x(1)-x(2))^2 - 2*x(1)^2)*cos(x(2))*sin(2*x(2));
end
```

Start at x0 = [0.25,-0.25] and search for a minimum of objectivefcn.

```
x0 = [0.25,-0.25];
x = fminsearch(@objectivefcn1,x0)
```

```
x =

   -0.1696    -0.5086
```

Minimize with Extra Parameters

Sometimes your objective function has extra parameters. These parameters are not variables to optimize, they are fixed values during the optimization. For example, suppose that you have a parameter `a` in the Rosenbrock-type function

$$f(x, a) = 100(x_2 - x_1^2)^2 + (a - x_1)^2 .$$

This function has a minimum value of 0 at $x_1 = a$, $x_2 = a^2$. If, for example, $a = 3$, you can include the parameter in your objective function by creating an anonymous function.

Create the objective function with its extra parameters as extra arguments.

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

Put the parameter in your MATLAB® workspace.

```
a = 3;
```

Create an anonymous function of `x` alone that includes the workspace value of the parameter.

```
fun = @(x)f(x,a);
```

Solve the problem starting at `x0 = [-1,1.9]`.

```
x0 = [-1,1.9];
x = fminsearch(fun,x0)
```

x = 1×2

```
   3.0000    9.0000
```

For more information about using extra parameters in your objective function, see "Parameterizing Functions".

Find Minimum Location and Value

Find both the location and value of a minimum of an objective function using `fminsearch`.

Write an anonymous objective function for a three-variable problem.

```
x0 = [1,2,3];
fun = @(x)-norm(x+x0)^2*exp(-norm(x-x0)^2 + sum(x));
```

Find the minimum of `fun` starting at `x0`. Find the value of the minimum as well.

```
[x,fval] = fminsearch(fun,x0)

x = 1×3

    1.5359    2.5645    3.5932


fval = -5.9565e+04
```

Inspect Optimization Process

Inspect the results of an optimization, both while it is running and after it finishes.

Set options to provide iterative display, which gives information on the optimization as the solver runs. Also, set a plot function to show the objective function value as the solver runs.

```
options = optimset('Display','iter','PlotFcns',@optimplotfval);
```

Set an objective function and start point.

```
function f = objectivefcn1(x)
f = 0;
for k = -10:10
    f = f + exp(-(x(1)-x(2))^2 - 2*x(1)^2)*cos(x(2))*sin(2*x(2));
end
```

Include the code for `objectivefcn1` as a file on your MATLAB® path.

```
x0 = [0.25,-0.25];
fun = @objectivefcn1;
```

Obtain all solver outputs. Use these outputs to inspect the results after the solver finishes.

```
[x,fval,exitflag,output] = fminsearch(fun,x0,options)
```

```
 Iteration   Func-count     min f(x)         Procedure
     0            1         -6.70447
     1            3         -6.89837         initial simplex
     2            5         -7.34101         expand
     3            7         -7.91894         expand
     4            9         -9.07939         expand
     5           11         -10.5047         expand
     6           13         -12.4957         expand
     7           15         -12.6957         reflect
     8           17         -12.8052         contract outside
     9           19         -12.8052         contract inside
    10           21         -13.0189         expand
    11           23         -13.0189         contract inside
    12           25         -13.0374         reflect
    13           27          -13.122         reflect
    14           28          -13.122         reflect
    15           29          -13.122         reflect
    16           31          -13.122         contract outside
```

```
     17            33          -13.1279          contract inside
     18            35          -13.1279          contract inside
     19            37          -13.1296          contract inside
     20            39          -13.1301          contract inside
     21            41          -13.1305          reflect
     22            43          -13.1306          contract inside
     23            45          -13.1309          contract inside
     24            47          -13.1309          contract inside
     25            49          -13.131           reflect
     26            51          -13.131           contract inside
     27            53          -13.131           contract inside
     28            55          -13.131           contract inside
     29            57          -13.131           contract outside
     30            59          -13.131           contract inside
     31            61          -13.131           contract inside
     32            63          -13.131           contract inside
     33            65          -13.131           contract outside
     34            67          -13.131           contract inside
     35            69          -13.131           contract inside

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-04
 and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 1.000000e-04


x =

   -0.1696    -0.5086


fval =

   -13.1310


exitflag =

     1


output =

  struct with fields:

    iterations: 35
     funcCount: 69
     algorithm: 'Nelder-Mead simplex direct search'
       message: 'Optimization terminated:...'
```

Current Function Value: -13.131
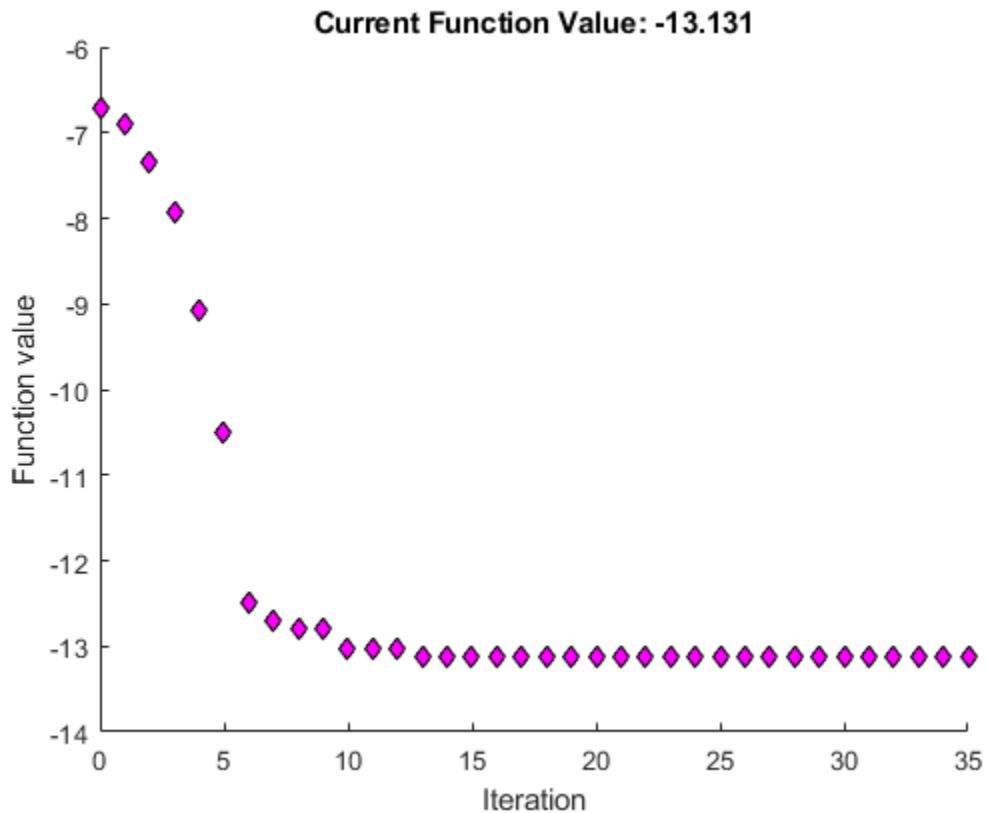
The value of `exitflag` is `1`, meaning `fminsearch` likely converged to a local minimum.

The `output` structure shows the number of iterations. The iterative display and the plot show this information as well. The `output` structure also shows the number of function evaluations, which the iterative display shows, but the chosen plot function does not.

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 = fminsearch(@myfun,x0)
```

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 = fminsearch(@(x)norm(x)^2,x0);
```

Example: `fun = @(x)-x*exp(-3*x)`

Data Types: `char` | `function_handle` | `string`

x0 — Initial point
real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in `x0` and the 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
structure such as `optimset` returns

Optimization options, specified as a structure such as `optimset` returns. You can use `optimset` to set or change the values of these fields in the options structure. See "Optimization Options Reference" on page 15-6 for detailed information.

`Display`	Level of display (see "Iterative Display" on page 3-14):
	• `'notify'` (default) displays output only if the function does not converge.
	• `'final'` displays just the final output.
	• `'off'` or `'none'` displays no output.
	• `'iter'` displays output at each iteration.
`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 2-68 and "Iterations and Function Counts" on page 3-9.
`MaxIter`	Maximum number of iterations allowed, a positive integer. The default value is `200*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9.
`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 and Plot Function Syntax" on page 15-27.

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.

Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27.

TolFun	Termination tolerance on the function value, a positive scalar. The default is `1e-4`. See "Tolerances and Stopping Criteria" on page 2-68. 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 2-68. Unlike other solvers, `fminsearch` stops when it satisfies *both* `TolFun` and `TolX`.

Example: `options = optimset('Display','iter')`

Data Types: `struct`

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	'fminsearch'
options	Options structure such as returned by optimset

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 4-18.

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 `fminsearch` stopped
integer

Reason `fminsearch` stopped, returned as an integer.

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 — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

iterations	Number of iterations
funcCount	Number of function evaluations
algorithm	`'Nelder-Mead simplex direct search'`
message	Exit message

Tips

- `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 values, split x into real and imaginary parts.
- Use `fminsearch` to solve nondifferentiable problems or problems with discontinuities, particularly if no discontinuity occurs near the solution.
- `fminsearch` is generally less efficient than `fminunc`, especially for problems of dimension greater than two. However, when the problem is discontinuous, `fminsearch` can be more robust than `fminunc`.
- `fminsearch` is not the preferred solver for problems that are sums of squares, that is, of the form

$$\min_x \|f(x)\|_2^2 = \min_x \left(f_1(x)^2 + f_2(x)^2 + \dots + f_n(x)^2 \right)$$

  Instead, use the `lsqnonlin` function, which has been optimized for problems of this form.

Algorithms

`fminsearch` uses the simplex search method of Lagarias et al. [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 6-9. The algorithm is not guaranteed to converge to a local minimum.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `fminsearch`.

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, 1998, pp. 112–147.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation:

- `fminsearch` ignores the `Display` option and does not give iterative display or an exit message. To check solution quality, examine the exit flag.
- The output structure does not include the `algorithm` or `message` fields.
- `fminsearch` ignores the `OutputFcn` and `PlotFcns` options.

See Also

Optimize | `fminbnd` | `fminunc` | `optimset`

Topics
"Create Function Handle"
"Anonymous Functions"

Introduced before R2006a

fminunc

Find minimum of unconstrained multivariable function

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

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 2-31.

`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 2-57 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 2-4.

`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`, a structure described in `problem`.

`[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 3-24.

Examples

Minimize a Polynomial

Minimize the function $f(x) = 3x_1^2 + 2x_1x_2 + x_2^2 - 4x_1 + 5x_2$.

To do so, write an anonymous function `fun` 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)
```

```
Local minimum found.

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

x = 1×2

    2.2500   -4.7500


fval = -16.3750
```

Supply 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 Gradients and Hessians" on page 2-19. 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

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

The code for the objective function with gradient appears at the end of this example on page 16-0   .

Create options to use the objective function's gradient. Also, set the algorithm to `'trust-region'`.

```
options = optimoptions('fminunc','Algorithm','trust-region','SpecifyObjectiveGradient',true);
```

Set the initial point to `[-1,2]`. Then call `fminunc`.

```
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 value of the optimality tolerance.

x = 1×2

    1.0000    1.0000
```

The following code creates the `rosenbrockwithgrad` function, which includes the gradient as the second output.

```
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
end
```

Use Problem Structure

Solve the same problem as in "Supply Gradient" on page 16-142 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 Gradients and Hessians" on page 2-19. 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

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

The code for the objective function with gradient appears at the end of this example on page 16-0 .

Create options to use the objective function's gradient. Also, set the algorithm to `'trust-region'`.

```
options = optimoptions('fminunc','Algorithm','trust-region','SpecifyObjectiveGradient',true);
```

Create a problem structure including the initial point `x0 = [-1,2]`. For the required fields in this structure, see "problem" on page 16-0 .

```
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 value of the optimality tolerance.

x = *1×2*

    1.0000   1.0000

The following code creates the `rosenbrockwithgrad` function, which includes the gradient as the second output.

```
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
end
```

Obtain 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 value of the optimality tolerance.

x = *1×2*

   -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 value of the optimality tolerance.

x = 1×2

  -0.6691    0.0000


fval = -0.4052

exitflag = 1

output = struct with fields:
      iterations: 9
       funcCount: 42
        stepsize: 2.9343e-04
     lssteplength: 1
    firstorderopt: 7.9721e-07
       algorithm: 'quasi-newton'
         message: '...'
```

- 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.

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 = fminunc(@myfun,x0)
```

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 = fminunc(@(x)norm(x)^2,x0);
```

If you can compute the gradient of `fun` *and* the `SpecifyObjectiveGradient` option is set to `true`, as set by

```
options = optimoptions('fminunc','SpecifyObjectiveGradient',true)
```

then `fun` must return the gradient vector `g(x)` in the second output argument.

If you can also compute the Hessian matrix *and* the `HessianFcn` option is set to `'objective'` via `options = optimoptions('fminunc','HessianFcn','objective')` *and* the `Algorithm` option is set to `'trust-region'`, `fun` must return the Hessian value `H(x)`, a symmetric matrix, in a third output argument. `fun` can give a sparse Hessian. See "Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms" on page 2-21 for details.

The `trust-region` 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 2-23.

Example: `fun = @(x)sin(x(1))*cos(x(2))`

Data Types: `char` | `function_handle` | `string`

x0 — Initial point
real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in `x0` and the 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 15-6 for detailed information.

Some options are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

All Algorithms

Algorithm	Choose the `fminunc` algorithm. Choices are `'quasi-newton'` (default) or `'trust-region'`.
	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 2-6.
CheckGradients	Compare user-supplied derivatives (gradient of objective) to finite-differencing derivatives. Choices are `false` (default) or `true`.
	For `optimset`, the name is `DerivativeCheck` and the values are `'on'` or `'off'`. See "Current and Legacy Option Names" on page 15-22.
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 3-14):

- `'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.

FiniteDifferenceSize	Scalar or vector step size factor for finite differences. When you set `FiniteDifferenceStepSize` to a vector `v`, the forward finite differences `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 `FiniteDifferenceStepSize` 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 `FiniteDifferenceStepSize` only when `CheckGradients` is set to `true`.
	For `optimset`, the name is `FinDiffRelStep`. See "Current and Legacy Option Names" on page 15-22.
FiniteDifferenceType	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 `FiniteDifferenceType` only when `CheckGradients` is set to `true`.
	For `optimset`, the name is `FinDiffType`. See "Current and Legacy Option Names" on page 15-22.
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`.
MaxFunctionEvaluations	Maximum number of function evaluations allowed, a positive integer. The default value is `100*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9.
	For `optimset`, the name is `MaxFunEvals`. See "Current and Legacy Option Names" on page 15-22.
MaxIterations	Maximum number of iterations allowed, a positive integer. The default value is `400`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9.
	For `optimset`, the name is `MaxIter`. See "Current and Legacy Option Names" on page 15-22.
OptimalityTolerance	Termination tolerance on the first-order optimality (a positive scalar). The default is `1e-6`. See "First-Order Optimality Measure" on page 3-11.
	For `optimset`, the name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.

OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration. Pass a function handle or a cell array of function handles. The default is none ([]). See "Output Function and Plot Function Syntax" on page 15-27.
PlotFcn	Plots various measures of progress while the algorithm executes; select from predefined plots or write your own. Pass a built-in plot function name, a function handle, or a cell array of built-in plot function names or function handles. For custom plot functions, pass 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.

Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27.

For optimset, the name is PlotFcns. See "Current and Legacy Option Names" on page 15-22.

SpecifyObjectiveGradient	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 true to have fminunc use a user-defined gradient of the objective function. The default false causes fminunc to estimate gradients using finite differences. You must provide the gradient, and set SpecifyObjectiveGradient to true, to use the trust-region algorithm. This option is not required for the quasi-Newton algorithm.

For optimset, the name is GradObj and the values are 'on' or 'off'. See "Current and Legacy Option Names" on page 15-22.

StepTolerance	Termination tolerance on x, a positive scalar. The default value is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68.

For optimset, the name is TolX. See "Current and Legacy Option Names" on page 15-22.

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 CheckGradients option.

trust-region Algorithm

FunctionTolerance	Termination tolerance on the function value, a positive scalar. The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68. For optimset, the name is TolFun. See "Current and Legacy Option Names" on page 15-22.
HessianFcn	If set to [] (default), fminunc approximates the Hessian using finite differences. If set to 'objective', fminunc uses a user-defined Hessian for the objective function. The Hessian is the third output of the objective function (see fun). For optimset, the name is HessFcn. See "Current and Legacy Option Names" on page 15-22.

HessianMultiplyFcn	Hessian multiply function, specified as a function handle. 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 2-57.
	
	Note To use the HessianMultiplyFcn option, HessianFcn must be set to [].
	
	For an example, see "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-93.
	
	For optimset, the name is HessMult. See "Current and Legacy Option Names" on page 15-22.

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). 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 16-153.
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.
SubproblemAlgorithm	Determines how the iteration step is calculated. The default, 'cg', takes a faster but less accurate step than 'factorization'. See "fminunc trust-region Algorithm" on page 6-2.
TolPCG	Termination tolerance on the PCG iteration, a positive scalar. The default is 0.1.

quasi-newton Algorithm

HessUpdate	Method for choosing the search direction in the Quasi-Newton algorithm. The choices are:

• 'bfgs', the default
• 'dfp'
• 'steepdesc'

See "Quasi-Newton Algorithm" on page 16-153 and "Hessian Update" on page 6-7 for a description of these methods. |
| 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. |

UseParallel	When `true`, fminunc estimates gradients in parallel. Disable by setting to the default, `false`. `trust-region` requires a gradient in the objective, so `UseParallel` does not apply. See "Parallel Computing".

Example: `options = optimoptions('fminunc','SpecifyObjectiveGradient',true)`

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	`'fminunc'`
options	Options created with `optimoptions`

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 4-18.

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 `OptimalityTolerance` tolerance.
2	Change in x was smaller than the `StepTolerance` tolerance.
3	Change in the objective function value was less than the `FunctionTolerance` tolerance.
5	Predicted decrease in the objective function was less than the `FunctionTolerance` tolerance.
0	Number of iterations exceeded `MaxIterations` or number of function evaluations exceeded `MaxFunctionEvaluations`.
-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 Output" on page 3-24.

Algorithms

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 6-4.

Trust Region Algorithm

The `trust-region` algorithm requires that you supply the gradient in `fun` and set `SpecifyObjectiveGradient` to `true` 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 6-3.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `fminunc`.

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.

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the `'UseParallel'` option to `true`.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see "Using Parallel Computing in Optimization Toolbox" on page 14-5.

See Also

Optimize | `fmincon` | `fminsearch` | `optimoptions`

Topics
"Solver-Based Nonlinear Optimization"
"Solver-Based Optimization Problem Setup"

"Unconstrained Nonlinear Optimization Algorithms" on page 6-2

Introduced before R2006a

fseminf

Find minimum of semi-infinitely constrained multivariable nonlinear function

Equation

Finds the minimum of a problem specified by

$$\min_{x} f(x) \text{ such that } \begin{cases} A \cdot x \le b, \\ Aeq \cdot x = beq, \\ lb \le x \le ub, \\ c(x) \le 0, \\ ceq(x) = 0, \\ K_i(x, w_i) \le 0, \ 1 \le i \le n. \end{cases}$$

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) $K_i(x,w_i) \le 0$ are continuous functions of both x and an additional set of variables $w_1, w_2, ..., w_n$. The variables $w_1, w_2, ..., w_n$ are vectors of, at most, length two.

x, lb, and ub can be passed as vectors or matrices; see "Matrix Arguments" on page 2-31.

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 2-57 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 2-33.

---

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 16-157.

[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 Input Arguments" on page 15-2 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 `SpecifyObjectiveGradient` option is `true`, as set by
	`options = optimoptions('fseminf','SpecifyObjectiveGradient',true)`
	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 16-161 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 6-139). 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 2-57 explains how to parametrize seminfcon, if necessary. "Example of Creating Sampling Points" on page 6-33 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 Input Arguments" on page 15-2 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.ConstraintTolerance.
5	Magnitude of directional derivative was less than the specified tolerance and constraint violation was less than options.ConstraintTolerance.
0	Number of iterations exceeded options.MaxIterations or number of function evaluations exceeded options.MaxFunctionEvaluations.
-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 15-6 for detailed information.

Some options are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

`CheckGradients`	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. The choices are `true` or the default `false`.
	For `optimset`, the name is `DerivativeCheck` and the values are `'on'` or `'off'`. See "Current and Legacy Option Names" on page 15-22.
`ConstraintTolerance`	Termination tolerance on the constraint violation, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-68.
	For `optimset`, the name is `TolCon`. See "Current and Legacy Option Names" on page 15-22.
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 3-14):

For `Display`:

- `'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.

`FiniteDifferenceStepSize`	Scalar or vector step size factor for finite differences. When you set `FiniteDifferenceStepSize` to a vector `v`, the forward finite differences `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 `FiniteDifferenceStepSize` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.
	
	For `optimset`, the name is `FinDiffRelStep`. See "Current and Legacy Option Names" on page 15-22.

`FiniteDifferenceType`	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. For `optimset`, the name is `FinDiffType`. See "Current and Legacy Option Names" on page 15-22.
`FunctionTolerance`	Termination tolerance on the function value, a positive scalar. The default is `1e-4`. See "Tolerances and Stopping Criteria" on page 2-68. For `optimset`, the name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.
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.
`MaxFunctionEvaluations`	Maximum number of function evaluations allowed, a positive integer. The default is `100*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9. For `optimset`, the name is `MaxFunEvals`. See "Current and Legacy Option Names" on page 15-22.

MaxIterations	Maximum number of iterations allowed, a positive integer. The default is 400. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9. For optimset, the name is MaxIter. See "Current and Legacy Option Names" on page 15-22.
MaxSQPIter	Maximum number of SQP iterations allowed, a positive integer. The default is 10*max(numberOfVariables, numberOfInequalities + numberOfBounds).
OptimalityTolerance	Termination tolerance on the first-order optimality (a positive scalar). The default is 1e-6. See "First-Order Optimality Measure" on page 3-11. For optimset, the name is TolFun. See "Current and Legacy Option Names" on page 15-22.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration. Pass a function handle or a cell array of function handles. The default is none ([]). See "Output Function and Plot Function Syntax" on page 15-27.
PlotFcn	Plots various measures of progress while the algorithm executes; select from predefined plots or write your own. Pass a name, a function handle, or a cell array of names or function handles. For custom plot functions, pass function handles. The default is none ([]): • 'optimplotx' plots the current point. • 'optimplotfunccount' plots the function count. • 'optimplotfval' plots the function value. • 'optimplotfvalconstr' plots the best feasible objective function value found as a line plot. The plot shows infeasible points as red, and feasible points as blue, using a feasibility tolerance of 1e-6. • 'optimplotconstrviolation' plots the maximum constraint violation. • 'optimplotstepsize' plots the step size. • 'optimplotfirstorderopt' plots the first-order optimality measure. Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27. For optimset, the name is PlotFcns. See "Current and Legacy Option Names" on page 15-22.

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 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)
SpecifyObjectiveGradient	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 true to have fseminf use a user-defined gradient of the objective function. The default false causes fseminf to estimate gradients using finite differences.
	For optimset, the name is GradObj and the values are 'on' or 'off'. See "Current and Legacy Option Names" on page 15-22.
StepTolerance	Termination tolerance on x, a positive scalar. The default value is 1e-4. See "Tolerances and Stopping Criteria" on page 2-68.
	For optimset, the name is TolX. See "Current and Legacy Option Names" on page 15-22.
TolConSQP	Termination tolerance on inner iteration SQP constraint violation, a positive scalar. The default is 1e-6.
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

Minimize Function with Semi-Infinite Constraints

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

$$\max_{t} g(x, t) \le 0 \text{ when } x \le 1/2.$$

To solve this problem using `fseminf`, write the objective function as an anonymous function.

```
objfun = @(x)(x-1)^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 $g(x, t)$. The function appears as `seminfcon` at the end of this section on page 16-0 .

Set the initial point `x0 = 0.2`.

```
x0 = 0.2;
```

There is one semi-infinite constraint.

```
ntheta = 1;
```

Solve the problem by calling `fseminf` and view the result.

```
x = fseminf(objfun,x0,ntheta,@seminfcon)
```

Local minimum found that satisfies the constraints.

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

x = 0.5000

The following code creates the `seminfcon` function.

```
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;
end
```

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.

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 6-25. For more details on the `fseminf` algorithm, see "fseminf Problem Formulation and Algorithm" on page 6-32.

See Also

`fmincon` | `optimoptions` | `optimtool`

Topics
"Create Function Handle"
"fseminf Problem Formulation and Algorithm" on page 6-32
"Multiobjective Optimization"

Introduced before R2006a

fsolve

Solve system of nonlinear equations

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

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 2-31.

`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`, a structure described in `problem`.

`[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.

```
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 value of the function tolerance, and
the problem appears regular as measured by the gradient.


x =

    0.3532    0.6061
```

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','PlotFcn',@optimplotfirstorderopt);
```

The equations in the nonlinear system are

$$e^{-e^{-(x_1+x_2)}} = x_2\left(1 + x_1^2\right)$$
$$x_1 \cos\left(x_2\right) + x_2 \sin\left(x_1\right) = \tfrac{1}{2}.$$

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.

```matlab
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.

```matlab
fun = @root2d;
x0 = [0,0];
x = fsolve(fun,x0,options)
```

```
x =

    0.3532    0.6061
```
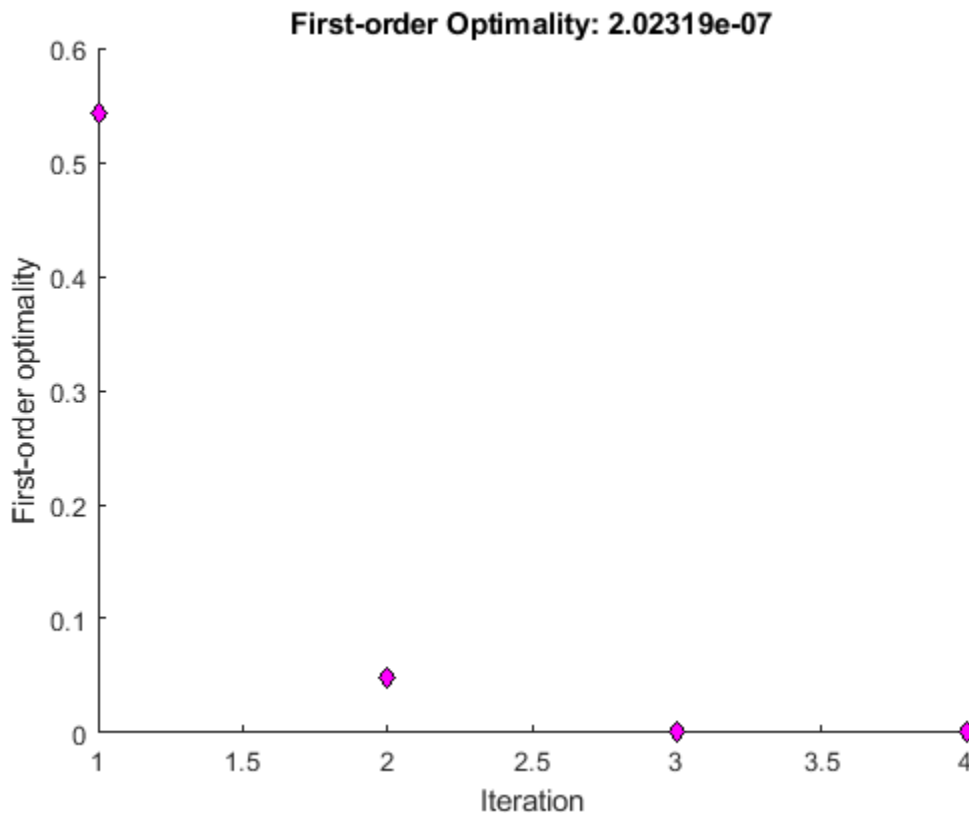

Solve a Problem Structure

Create a problem structure for `fsolve` and solve the problem.

Solve the same problem as in "Solution with Nondefault Options" on page 16-168, 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','PlotFcn',@optimplotfirstorderopt);
```

The equations in the nonlinear system are

$$e^{-e^{-(x_1+x_2)}} = x_2\left(1+x_1^2\right)$$
$$x_1\cos\left(x_2\right) + x_2\sin\left(x_1\right) = \tfrac{1}{2}.$$

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.

```
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 =

    0.3532    0.6061
```

First-order Optimality: 2.02319e-07
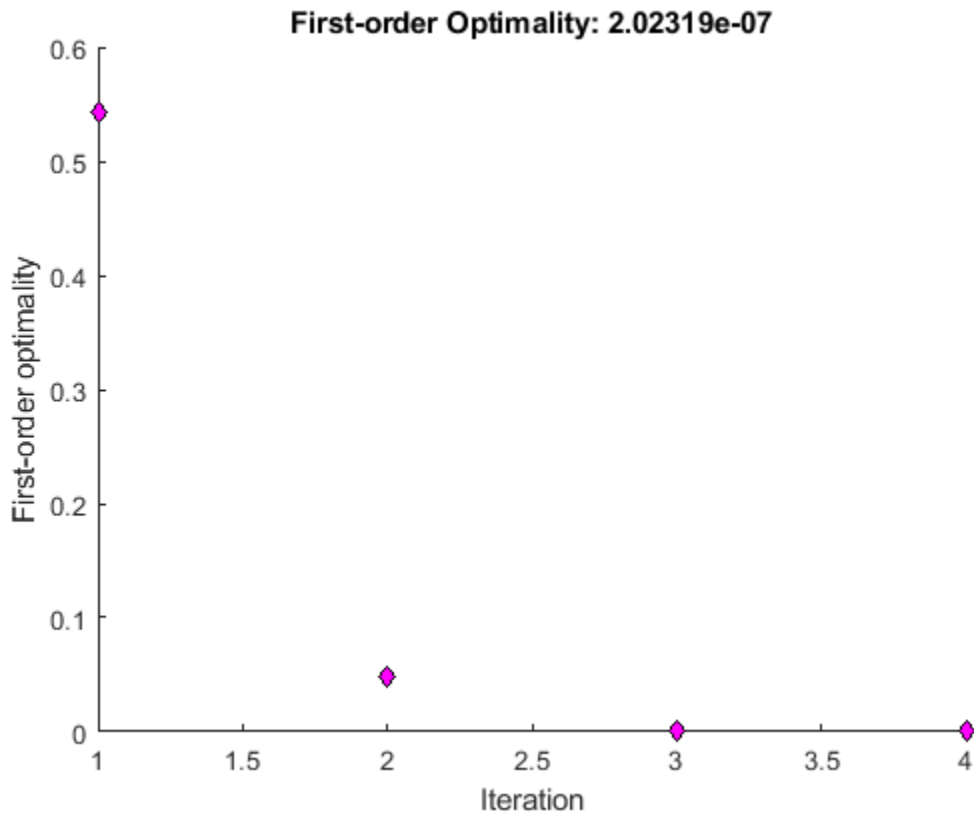
Solution Process of Nonlinear System

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 function that computes F, the values of the equations at x.

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

Create the initial point x0.

```
x0 = [-5;-5];
```

Set options to return iterative display.

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

Solve the equations.

```
[x,fval] = fsolve(F,x0,options)
```

Iteration	Func-count	f(x)	Norm of step	First-order optimality	Trust-region radius
0	3	47071.2		2.29e+04	1
1	6	12003.4	1	5.75e+03	1
2	9	3147.02	1	1.47e+03	1
3	12	854.452	1	388	1
4	15	239.527	1	107	1
5	18	67.0412	1	30.8	1
6	21	16.7042	1	9.05	1
7	24	2.42788	1	2.26	1
8	27	0.032658	0.759511	0.206	2.5
9	30	7.03149e-06	0.111927	0.00294	2.5
10	33	3.29525e-13	0.00169132	6.36e-07	2.5

```
Equation solved.

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

x = 2×1

    0.5671
    0.5671


fval = 2×1
10⁻⁶ ×

   -0.4059
   -0.4059
```

The iterative display shows `f(x)`, which is the square of the norm of the function `F(x)`. This value decreases to near zero as the iterations proceed. The first-order optimality measure likewise decreases to near zero as the iterations proceed. These entries show the convergence of the iterations to a solution. For the meanings of the other entries, see "Iterative Display" on page 3-14.

The `fval` output gives the function value `F(x)`, which should be zero at a solution (to within the `FunctionTolerance` tolerance).

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 x0 = [1,1;1,1]. Create an anonymous function that calculates the matrix equation and create the point x0.

```
fun = @(x)x*x*x - [1,2;3,4];
x0 = ones(2);
```

Set options to have no display.

```
options = optimoptions('fsolve','Display','off');
```

Examine the `fsolve` outputs to see the solution quality and process.

```
[x,fval,exitflag,output] = fsolve(fun,x0,options)
```

x = *2×2*

```
   -0.1291    0.8602
    1.2903    1.1612
```

fval = *2×2*
10^{-9} ×

```
   -0.1618    0.0778
    0.1160   -0.0474
```

exitflag = 1

output = *struct with fields:*
```
      iterations: 6
       funcCount: 35
       algorithm: 'trust-region-dogleg'
    firstorderopt: 2.4095e-10
         message: '...'
```

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 = 4.7957e-20

This small residual confirms that x is a solution.

You can see in the `output` structure how many iterations and function evaluations `fsolve` performed to find the solution.

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 arrays, they are converted to vectors using linear indexing (see "Array Indexing").

If the Jacobian can also be computed *and* the `'SpecifyObjectiveGradient'` option is `true`, set by

```
options = optimoptions('fsolve','SpecifyObjectiveGradient',true)
```

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` | `string`

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 such as `optimset` returns.

Some options apply to all algorithms, and others are relevant for particular algorithms. See "Optimization Options Reference" on page 15-6 for detailed information.

Some options are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

All Algorithms

Algorithm	Choose between `'trust-region-dogleg'` (default), `'trust-region'`, and `'levenberg-marquardt'`.
	The `Algorithm` option specifies a preference for which algorithm to use. It is only a preference because for the trust-region 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 2-6.
	To set some algorithm options using `optimset` instead of `optimoptions`:
	• `Algorithm` — Set the algorithm to `'trust-region-reflective'` instead of `'trust-region'`.
	• *InitDamping* — Set the initial Levenberg-Marquardt parameter λ by setting `Algorithm` to a cell array such as `{'levenberg-marquardt',.005}`.
CheckGradients	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. The choices are `true` or the default `false`.
	For `optimset`, the name is `DerivativeCheck` and the values are `'on'` or `'off'`. See "Current and Legacy Option Names" on page 15-22.
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 3-14):
	• `'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.

FiniteDifferenceStepSize	Scalar or vector step size factor for finite differences. When you set FiniteDifferenceStepSize to a vector v, the forward finite differences 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 FiniteDifferenceStepSize expands to a vector. The default is sqrt(eps) for forward finite differences, and eps^(1/3) for central finite differences. For optimset, the name is FinDiffRelStep. See "Current and Legacy Option Names" on page 15-22.
FiniteDifferenceType	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. For optimset, the name is FinDiffType. See "Current and Legacy Option Names" on page 15-22.
FunctionTolerance	Termination tolerance on the function value, a positive scalar. The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68. For optimset, the name is TolFun. See "Current and Legacy Option Names" on page 15-22.
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.
MaxFunctionEvaluations	Maximum number of function evaluations allowed, a positive integer. The default is 100*numberOfVariables. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9. For optimset, the name is MaxFunEvals. See "Current and Legacy Option Names" on page 15-22.
MaxIterations	Maximum number of iterations allowed, a positive integer. The default is 400. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9. For optimset, the name is MaxIter. See "Current and Legacy Option Names" on page 15-22.

`OptimalityTolerance`	Termination tolerance on the first-order optimality (a positive scalar). The default is `1e-6`. See "First-Order Optimality Measure" on page 3-11.
	Internally, the `'levenberg-marquardt'` algorithm uses an optimality tolerance (stopping criterion) of `1e-4` times `FunctionTolerance` and does not use `OptimalityTolerance`.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration. Pass a function handle or a cell array of function handles. The default is none (`[]`). See "Output Function and Plot Function Syntax" on page 15-27.
`PlotFcn`	Plots various measures of progress while the algorithm executes; select from predefined plots or write your own. Pass a built-in plot function name, a function handle, or a cell array of built-in plot function names or function handles. For custom plot functions, pass 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.
	Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27.
	For `optimset`, the name is `PlotFcns`. See "Current and Legacy Option Names" on page 15-22.
`SpecifyObjectiveGradient`	If `true`, `fsolve` uses a user-defined Jacobian (defined in `fun`), or Jacobian information (when using `JacobianMultiplyFcn`), for the objective function. If `false` (default), `fsolve` approximates the Jacobian using finite differences.
	For `optimset`, the name is `Jacobian` and the values are `'on'` or `'off'`. See "Current and Legacy Option Names" on page 15-22.
`StepTolerance`	Termination tolerance on x, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-68.
	For `optimset`, the name is `TolX`. See "Current and Legacy Option Names" on page 15-22.
`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.

UseParallel	When `true`, `fsolve` estimates gradients in parallel. Disable by setting to the default, `false`. See "Parallel Computing".

trust-region Algorithm

JacobianMultiplyFcn	Jacobian multiply function, specified as a function handle. 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 2-57 for information on how to supply values for any additional parameters `jmfun` needs.

Note `'SpecifyObjectiveGradient'` must be set to `true` for `fsolve` to pass `Jinfo` from `fun` to `jmfun`.

See "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-93 for a similar example.

For `optimset`, the name is `JacobMult`. See "Current and Legacy Option Names" on page 15-22.

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 $\partial\text{fun}(i)/\partial x(j) \neq 0$.
	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 13-2.
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.
`SubproblemAlgorithm`	Determines how the iteration step is calculated. The default, `'factorization'`, takes a slower but more accurate step than `'cg'`. See "Trust-Region Algorithm" on page 13-2.
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 12-6.
ScaleProblem	`'jacobian'` can sometimes improve the convergence of a poorly scaled problem. The default is `'none'`.

Example: `options = optimoptions('fsolve','FiniteDifferenceType','central')`

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`

Field Name	Entry
solver	'fsolve'
options	Options created with optimoptions

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 4-18.

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	Equation solved. First-order optimality is small.
2	Equation solved. Change in x smaller than the specified tolerance, or Jacobian at x is undefined.
3	Equation solved. Change in residual smaller than the specified tolerance.
4	Equation solved. Magnitude of search direction smaller than specified tolerance.
0	Number of iterations exceeded options.MaxIterations or number of function evaluations exceeded options.MaxFunctionEvaluations.
-1	Output function or plot function stopped the algorithm.
-2	Equation not solved. The exit message can have more information.
-3	Equation not solved. Trust region radius became too small (trust-region-dogleg algorithm).

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' 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.

Tips

- For large problems, meaning those with thousands of variables or more, save memory (and possibly save time) by setting the `Algorithm` option to `'trust-region'` and the `SubproblemAlgorithm` option to `'cg'`.

Algorithms

The Levenberg-Marquardt and trust-region 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 16-181).

- 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 Algorithm" on page 13-4.
- The trust-region 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 Algorithm" on page 13-2.
- The Levenberg-Marquardt method is described in references [4], [5], and [6]. See "Levenberg-Marquardt Method" on page 13-5.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `fsolve`.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

- `fsolve` supports code generation using either the `codegen` function or the MATLAB Coder app. You must have a MATLAB Coder license to generate code.
- The target hardware must support standard double-precision floating-point computations. You cannot generate code for single-precision or fixed-point computations.
- Code generation targets do not use the same math kernel libraries as MATLAB solvers. Therefore, code generation solutions can vary from solver solutions, especially for poorly conditioned problems.
- All code for generation must be MATLAB code. In particular, you cannot use a custom black-box function as an objective function for `fsolve`. You can use `coder.ceval` to evaluate a custom function coded in C or C++. However, the custom function must be called in a MATLAB function.
- `fsolve` does not support the `problem` argument for code generation.

  ```
  [x,fval] = fsolve(problem) % Not supported
  ```

- You must specify the objective function by using function handles, not strings or character names.

```
x = fsolve(@fun,x0,options) % Supported
% Not supported: fsolve('fun',...) or fsolve("fun",...)
```

- For advanced code optimization involving embedded processors, you also need an Embedded Coder license.

- You must include options for `fsolve` and specify them using `optimoptions`. The options must include the `Algorithm` option, set to `'levenberg-marquardt'`.

```
options = optimoptions('fsolve','Algorithm','levenberg-marquardt');
[x,fval,exitflag] = fsolve(fun,x0,options);
```

- Code generation supports these options:

  - `Algorithm` — Must be `'levenberg-marquardt'`
  - `FiniteDifferenceStepSize`
  - `FiniteDifferenceType`
  - `FunctionTolerance`
  - `MaxFunctionEvaluations`
  - `MaxIterations`
  - `SpecifyObjectiveGradient`
  - `StepTolerance`
  - `TypicalX`

- Generated code has limited error checking for options. The recommended way to update an option is to use `optimoptions`, not dot notation.

```
opts = optimoptions('fsolve','Algorithm','levenberg-marquardt');
opts = optimoptions(opts,'MaxIterations',1e4); % Recommended
opts.MaxIterations = 1e4; % Not recommended
```

- Do not load options from a file. Doing so can cause code generation to fail. Instead, create options in your code.

- Usually, if you specify an option that is not supported, the option is silently ignored during code generation. However, if you specify a plot function or output function by using dot notation, code generation can issue an error. For reliability, specify only supported options.

- Because output functions and plot functions are not supported, solvers do not return the exit flag – 1.

For an example, see "Generate Code for fsolve" on page 13-38.

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the `'UseParallel'` option to `true`.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see "Using Parallel Computing in Optimization Toolbox" on page 14-5.

See Also
Optimize | fzero | lsqcurvefit | lsqnonlin | optimoptions

Topics

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)
```
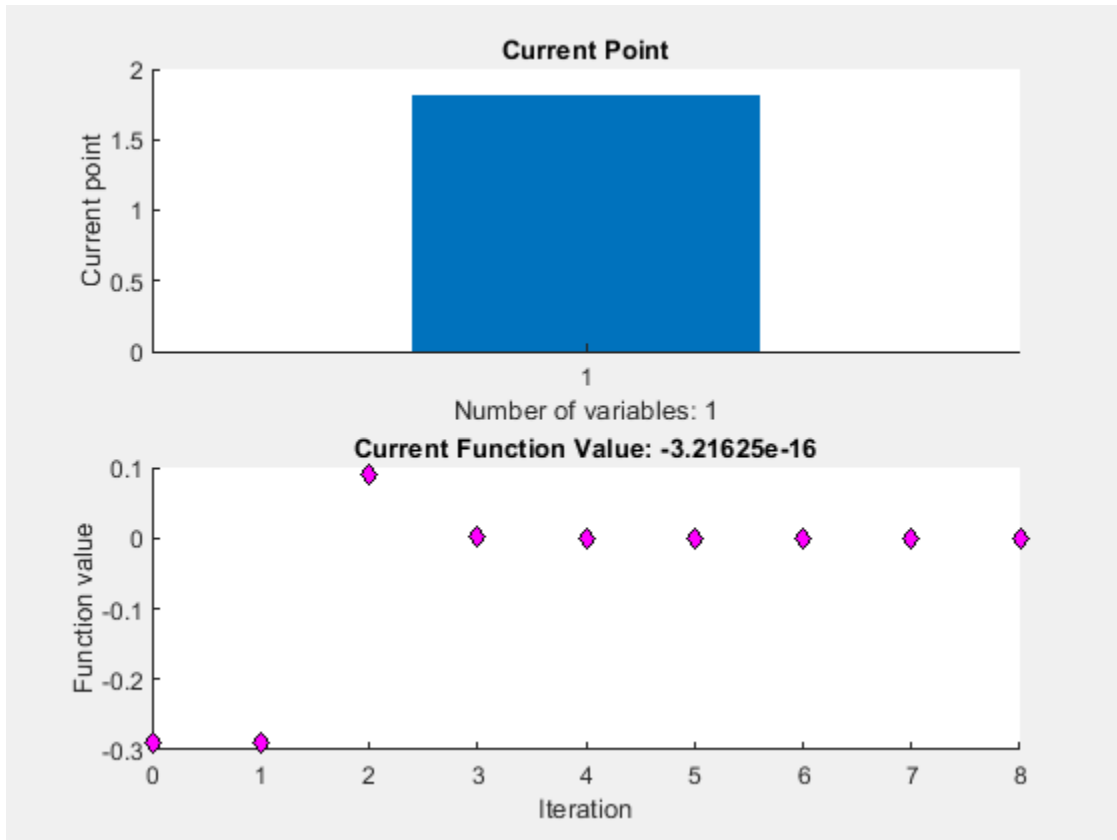


```
x = 1.8115
```

Solve Problem Structure

Solve a problem that is defined by a problem structure.

Define a structure that encodes a root-finding problem.

```
problem.objective = @(x)sin(cosh(x));
problem.x0 = 1;
problem.solver = 'fzero'; % a required part of the structure
problem.options = optimset(@fzero); % default options
```

Solve the problem.

```
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 = struct with fields:
    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.

Input Arguments

fun — Function to solve
function handle | function name

Function to solve, specified as a handle to a scalar-valued function or the name of such a 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 16-186 and the section "Passing Extra Parameters" on page 2-57.

Example: `'sin'`

Example: `@myFunction`

Example: `@(x)(x-a)^5 - 3*x + a - 1`

Data Types: char | function_handle | string

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 3-14):
	• '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 and Plot Function Syntax" on page 15-27.

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.
	Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27.
TolX	Termination tolerance on x, a positive scalar. The default is `eps`, 2.2204e–16. See "Tolerances and Stopping Criteria" on page 2-68.

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`

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

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].

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `fzero`.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation:

- The `fun` input argument must be a function handle, and not a structure or character vector.
- `fzero` ignores all options except for `TolX` and `FunValCheck`.
- `fzero` does not support the fourth output argument, the output structure.

See Also
Optimize | `fminbnd` | `fsolve` | `optimset` | `roots`

Topics
"Roots of Scalar Functions"
"Passing Extra Parameters" on page 2-57

Introduced before R2006a

infeasibility

Package: `optim.problemdef`

Constraint violation at a point

Syntax

```
infeas = infeasibility(constr,pt)
```

Description

Use `infeasibility` to find the numeric value of a constraint violation at a point.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`infeas = infeasibility(constr,pt)` returns the amount of violation of the constraint `constr` at the point `pt`.

Examples

Compute Constraint Violation

Check whether a point satisfies a constraint.

Set up optimization variables and two constraints.

```
x = optimvar('x');
y = optimvar('y');
cons = x + y <= 2;
cons2 = x + y/4 <= 1;
```

Check whether the point `x = 0`, `y = 3` satisfies the constraint named `cons`. A point is feasible when its infeasibility is zero.

```
pt.x = 0;
pt.y = 3;
infeas = infeasibility(cons,pt)
```

```
infeas = 1
```

The point is not feasible with respect to this constraint.

Check the feasibility with respect to the other constraint.

```
infeas = infeasibility(cons2,pt)
```

```
infeas = 0
```

The point is feasible with respect to this constraint.

Compute Multiple Constraint Violations

Check whether a point satisfies a constraint that has multiple conditions.

Set up an optimization variable and a vector of constraints.

```
x = optimvar('x',3,2);
cons = sum(x,2) <= [1;3;2];
```

Check whether the point `pt.x = [1,-1;2,3;3,-1]` satisfies these constraints.

```
pt.x = [1,-1;2,3;3,-1];
infeas = infeasibility(cons,pt)
```

infeas = *3×1*

```
    0
    2
    0
```

The point is not feasible with respect to the second constraint.

Input Arguments

`constr` — Optimization constraint
`OptimizationEquality` object | `OptimizationInequality` object | `OptimizationConstraint` object

Optimization constraint, specified as an `OptimizationEquality` object, `OptimizationInequality` object, or `OptimizationConstraint` object. `constr` can represent a single constraint or an array of constraints.

Example: `constr = x + y <= 1` is a single constraint when x and y are scalar variables.

Example: `constr = sum(x) == 1` is an array of constraints when x is an array of two or more dimensions.

`pt` — Point to evaluate
structure with field names that match the optimization variable names

Point to evaluate, specified as a structure with field names that match the optimization variable names, for optimization variables in the constraint. The size of each field in `pt` must match the size of the corresponding optimization variable.

Example: `pt.x = 5*eye(3)`

Data Types: `struct`

Output Arguments

`infeas` — Infeasibility of constraint
real array

Infeasibility of constraint, returned as a real array. Each zero entry represents a feasible constraint, and each positive entry represents an infeasible constraint. The size of `infeas` is the same as the size of the constraint `constr`. For an example of nonscalar `infeas`, see "Compute Multiple Constraint Violations" on page 16-194.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

See Also

`OptimizationConstraint` | `OptimizationEquality` | `OptimizationInequality` | `evaluate`

Topics
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

intlinprog

Mixed-integer linear programming (MILP)

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,x0)
x = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub,x0,options)
x = intlinprog(problem)
[x,fval,exitflag,output] = intlinprog( ___ )
```

Description

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 2-31.

Note `intlinprog` applies only to the solver-based approach. For a discussion of the two optimization approaches, see "First Choose Problem-Based or Solver-Based Approach" on page 1-3.

`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,x0)` optimizes using an initial feasible point x0. Set lb = [] and ub = [] if no bounds exist.

`x = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub,x0,options)` minimizes using the optimization options specified in `options`. Use `optimoptions` to set these options. Set x0 = [] if no initial point exists.

`x = intlinprog(problem)` uses a `problem` structure to encapsulate all solver inputs. You can import a `problem` structure from an MPS file using `mpsread`. You can also create a `problem` structure from an `OptimizationProblem` object by using `prob2struct`.

`[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
tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).

x = 2×1

    6.5000
    7.0000
```

Solve an MILP with All Types of Constraints

Solve the problem

$$\min_{x}(-3x_1 - 2x_2 - x_3) \text{ subject to} \begin{cases} x_3 \text{ binary} \\ x_1, x_2 \geq 0 \\ x_1 + x_2 + x_3 \leq 7 \\ 4x_1 + 2x_2 + x_3 = 12 \, . \end{cases}$$

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
tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).

x = 3×1

         0
    5.5000
    1.0000
```

Use Initial Point

Compare the number of steps to solve an integer programming problem both with and without an initial feasible point. The problem has eight variables, four linear equality constraints, and has all variables restricted to be positive.

Define the linear equality constraint matrix and vector.

```
Aeq = [22     13     26     33     21      3     14     26
       39     16     22     28     26     30     23     24
       18     14     29     27     30     38     26     26
       41     26     28     36     18     38     16     26];
beq = [ 7872
       10466
       11322
       12058];
```

Set lower bounds that restrict all variables to be nonnegative.

```
N = 8;
lb = zeros(N,1);
```

Specify that all variables are integer-valued.

```
intcon = 1:N;
```

Set the objective function vector f.

```
f = [2     10     13     17      7      5      7      3];
```

Solve the problem without using an initial point, and examine the display to see the number of branch-and-bound nodes.

```
[x1,fval1,exitflag1,output1] = intlinprog(f,intcon,[],[],Aeq,beq,lb);
```

LP: Optimal objective value is 1554.047531.

Cut Generation: Applied 8 strong CG cuts.
                   Lower bound is 1591.000000.

Branch and Bound:

```
  nodes      total   num int       integer        relative
explored   time (s)  solution          fval         gap (%)
  10000       0.90         0             -               -
  18027       1.55         1    2.906000e+03    4.509804e+01
  21859       1.97         2    2.073000e+03    2.270974e+01
  23546       2.14         3    1.854000e+03    1.180593e+01
  24121       2.20         3    1.854000e+03    1.563342e+00
  24294       2.23         3    1.854000e+03    0.000000e+00
```

Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the default value).

For comparison, find the solution using an initial feasible point.

```
x0 = [8 62 23 103 53 84 46 34];
[x2,fval2,exitflag2,output2] = intlinprog(f,intcon,[],[],Aeq,beq,lb,[],x0);
```

LP: Optimal objective value is 1554.047531.

```
Cut Generation:     Applied 8 strong CG cuts.
                    Lower bound is 1591.000000.
                    Relative gap is 59.20%.

Branch and Bound:

   nodes      total   num int       integer       relative
explored   time (s)  solution          fval        gap (%)
    3627       0.35         2   2.154000e+03   2.593968e+01
    5844       0.55         3   1.854000e+03   1.180593e+01
    6204       0.59         3   1.854000e+03   1.455526e+00
    6400       0.60         3   1.854000e+03   0.000000e+00

Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the
optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon
variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the
default value).
```

- Without an initial point, `intlinprog` took about 30,000 branch-and-bound steps.
- Using an initial point, `intlinprog` took about 5,000 steps.

Giving an initial point does not always help. For this problem, giving an initial point saves time and computational steps. However, for some problems, giving an initial point can cause `intlinprog` to take more steps.

Solve an MILP with Nondefault Options

Solve the problem

$$\min_{x}(-3x_1 - 2x_2 - x_3) \text{ subject to} \begin{cases} x_3 \text{ binary} \\ x_1, x_2 \geq 0 \\ x_1 + x_2 + x_3 \leq 7 \\ 4x_1 + 2x_2 + x_3 = 12 \end{cases}$$

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
x0 = [];
```

Specify no display.

```
options = optimoptions('intlinprog','Display','off');
```

Run the solver.

```
x = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub,x0,options)
```

x = *3×1*

```
         0
    5.5000
    1.0000
```

Solve MILP Using Problem-Based Approach

This example shows how to set up a problem using the problem-based approach and then solve it using the solver-based approach. The problem is

$$\min_{x}(-3x_1 - 2x_2 - x_3) \text{ subject to } \begin{cases} x_3 \text{ binary} \\ x_1, x_2 \geq 0 \\ x_1 + x_2 + x_3 \leq 7 \\ 4x_1 + 2x_2 + x_3 = 12 \end{cases}$$

Create an `OptimizationProblem` object named `prob` to represent this problem. To specify a binary variable, create an optimization variable with integer type, a lower bound of 0, and an upper bound of 1.

```
x = optimvar('x',2,'LowerBound',0);
xb = optimvar('xb','LowerBound',0,'UpperBound',1,'Type','integer');
prob = optimproblem('Objective',-3*x(1)-2*x(2)-xb);
cons1 = sum(x) + xb <= 7;
cons2 = 4*x(1) + 2*x(2) + xb == 12;
prob.Constraints.cons1 = cons1;
prob.Constraints.cons2 = cons2;
```

Convert the problem object to a problem structure.

```
problem = prob2struct(prob);
```

Solve the resulting problem structure.

```
[sol,fval,exitflag,output] = intlinprog(problem)
```

```
LP:                Optimal objective value is -12.000000.


Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap
tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).
```

sol = *3×1*

```
         0
```

```
     5.5000
     1.0000


fval = -12

exitflag = 1

output = struct with fields:
        relativegap: 0
        absolutegap: 0
      numfeaspoints: 1
           numnodes: 0
      constrviolation: 0
            message: 'Optimal solution found....'
```

Both `sol(1)` and `sol(3)` are binary-valued. Which value corresponds to the binary optimization variable `xb`?

```
prob.Variables
```

```
ans = struct with fields:
     x: [2x1 optim.problemdef.OptimizationVariable]
    xb: [1x1 optim.problemdef.OptimizationVariable]
```

The variable `xb` appears last in the `Variables` display, so `xb` corresponds to `sol(3) = 1`. See "Algorithms" on page 16-406.

Examine the MILP Solution and Process

Call `intlinprog` with more outputs to see solution details and process.

The goal is to solve the problem

$$\min_{x}(-3x_1 - 2x_2 - x_3) \text{ subject to} \begin{cases} x_3 \text{ binary} \\ x_1, x_2 \geq 0 \\ x_1 + x_2 + x_3 \leq 7 \\ 4x_1 + 2x_2 + x_3 = 12 \, . \end{cases}$$

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
tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).

x = 3×1

         0
    5.5000
    1.0000


fval = -12

exitflag = 1

output = struct with fields:
         relativegap: 0
         absolutegap: 0
       numfeaspoints: 1
            numnodes: 0
       constrviolation: 0
             message: 'Optimal solution found....'
```

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.

Input Arguments

f — Coefficient vector
real vector | real array

Coefficient vector, specified as a real vector or real array. The coefficient vector represents the objective function `f'*x`. The notation assumes that `f` is a column vector, but you are free to use a row vector or array. Internally, `linprog` converts `f` to the column 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 ≤ 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 ≤ 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`

x0 — Initial point
`[]` (default) | real array

Initial point, specified as a real array. The number of elements in `x0` is the same as the number of elements of `f`, when `f` exists. Otherwise, the number is the same as the number of columns of `A` or `Aeq`. Internally, the solver converts an array `x0` into a vector `x0(:)`.

Providing `x0` can change the amount of time `intlinprog` takes to converge. It is difficult to predict how `x0` affects the solver. For suggestions on using appropriate `Heuristics` with `x0`, see "Tips" on page 16-213.

`x0` must be feasible with respect to all constraints. If `x0` is not feasible, the solver errors. If you do not have a feasible `x0`, set `x0 = []`.

Example: `x0 = 100*rand(size(f))`

Data Types: `double`

options — Options for `intlinprog`
options created using `optimoptions`

Options for `intlinprog`, specified as the output of `optimoptions`.

Some options are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

Option	Description	Default
`AbsoluteGapTolerance`	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 `AbsoluteGapTolerance`: `U – L <= AbsoluteGapTolerance`.	0

Option	Description	Default
BranchRule	Rule for choosing the component for branching: • 'maxpscost' — The fractional component with maximum pseudocost. See "Branch and Bound" on page 9-48. • 'strongpscost' — The fractional component with maximum pseudocost and a more accurate estimate of pseudocost than in 'maxpscost'. See "Branch and Bound" on page 9-48. • 'reliability' — The fractional component with maximum pseudocost and an even more accurate estimate of pseudocost than in 'strongpscost'. See "Branch and Bound" on page 9-48. • 'mostfractional' — The component whose fractional part is closest to 1/2. • 'maxfun' — The fractional component with a maximal corresponding component in the absolute value of the objective vector f.	'reliability'
ConstraintTolerance	Real from 1e-9 through 1e-3 that is the maximum discrepancy that linear constraints can have and still be considered satisfied. ConstraintTolerance is not a stopping criterion.	1e-4
CutGeneration	Level of cut generation (see "Cut Generation" on page 9-45): • 'none' — No cuts. Makes CutMaxIterations irrelevant. • 'basic' — Normal cut generation. • 'intermediate' — Use more cut types. • 'advanced' — Use most cut types.	'basic'
CutMaxIterations	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 3-14): • '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 9-46): • `'basic'` • `'intermediate'` • `'advanced'` • `'rss'` • `'rins'` • `'round'` • `'diving'` • `'rss-diving'` • `'rins-diving'` • `'round-diving'` • `'none'`	`'basic'`
HeuristicsMax Nodes	Strictly positive integer that bounds the number of nodes `intlinprog` can explore in its branch-and-bound search for feasible points. Applies only to `'rss'` and `'rins'`. See "Heuristics for Finding Feasible Solutions" on page 9-46.	50
IntegerPrepro cess	Types of integer preprocessing (see "Mixed-Integer Program Preprocessing" on page 9-44): • `'none'` — Use very few integer preprocessing steps. • `'basic'` — Use a moderate number of integer preprocessing steps. • `'advanced'` — Use all available integer preprocessing steps.	`'basic'`
IntegerTolera nce	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. `IntegerTolerance` is not a stopping criterion.	`1e-5`

Option	Description	Default
LPMaxIterations	Strictly positive integer, the maximum number of simplex algorithm iterations per node during the branch-and-bound process.	max(3e4, 10*(numberOfEqualities + numberOfInequalities + numberOfVariables)) In this expression, numberOfEqualities means the number of rows of Aeq, numberOfInequalities means the number of rows of A, and numberOfVariables means the number of elements of f.
LPOptimalityTolerance	Nonnegative real where reduced costs must exceed LPOptimalityTolerance for a variable to be taken into the basis.	1e-7
LPPreprocess	Type of preprocessing for the solution to the relaxed linear program (see "Linear Program Preprocessing" on page 9-44): • '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
MaxFeasiblePoints	Strictly positive integer. intlinprog stops if it finds MaxFeasiblePoints integer feasible points.	Inf
MaxTime	Positive real that is the maximum time in seconds that intlinprog runs.	7200
NodeSelection	Choose the node to explore next. • 'simplebestproj' — Best projection. See "Branch and Bound" on page 9-48. • '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 9-48.	'simplebestproj'
ObjectiveCutOff	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

Option	Description	Default
`ObjectiveImprovementThreshold`	Nonnegative real. `intlinprog` changes the current feasible solution only when it locates another with an objective function value that is at least `ObjectiveImprovementThreshold` lower: (fold – fnew)/(1 + |fold|) > ObjectiveImprovementThreshold.	0
`OutputFcn`	One or more functions that an optimization function calls at events. Specify as `'savemilpsolutions'`, a function handle, or a cell array of function handles. For custom output functions, pass function handles. An output function can stop the solver. • `'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 Function and Plot Function Syntax" on page 15-35.	`[]`
`PlotFcn`	Plots various measures of progress while the algorithm executes; select from predefined plots or write your own. Pass `'optimplotmilp'`, a function handle, or a cell array of function handles. For custom plot functions, pass function handles. The default is none (`[]`): • `'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 Function and Plot Function Syntax" on page 15-35.	`[]`

Option	Description	Default
RelativeGapTolerance	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 RelativeGapTolerance: (U − L) / (abs(U) + 1) <= RelativeGapTolerance. intlinprog automatically modifies the tolerance for large L magnitudes: tolerance = min(1/(1+|L|), RelativeGapTolerance) **Note** Although you specify RelativeGapTolerance as a decimal number, the iterative display and output.relativegap report the gap as a percentage, meaning 100 times the measured relative gap. If the exit message refers to the relative gap, this value is the measured relative gap, not a percentage.	1e-4
RootLPAlgorithm	Algorithm for solving linear programs: • 'dual-simplex' — Dual simplex algorithm • 'primal-simplex' — Primal simplex algorithm	'dual-simplex'
RootLPMaxIterations	Nonnegative integer that is the maximum number of simplex algorithm iterations to solve the initial linear programming problem.	max(3e4, 10*(numberOfEqualities + numberOfInequalities + numberOfVariables)) In this expression, numberOfEqualities means the number of rows of Aeq, numberOfInequalities means the number of rows of A, and numberOfVariables means the number of elements of f.

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
`x0`	Initial feasible point
`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.

3	The solution is feasible with respect to the relative `ConstraintTolerance` tolerance, but is not feasible with respect to the absolute tolerance.
2	`intlinprog` stopped prematurely. Integer feasible point found.
1	`intlinprog` converged to the solution `x`.
0	`intlinprog` stopped prematurely. No integer feasible point found.
-1	`intlinprog` stopped by an output function or plot function.
-2	No feasible point found.
-3	Root LP problem is unbounded.
-9	Solver lost feasibility.

The exit message can give more detailed information on the reason `intlinprog` stopped, such as exceeding a tolerance.

Exitflags 3 and -9 relate to solutions that have large infeasibilities. These usually arise from linear constraint matrices that have large condition number, or problems that have large solution components. To correct these issues, try to scale the coefficient matrices, eliminate redundant linear constraints, or give tighter bounds on the variables.

output — Solution process summary
structure

Solution process summary, returned as a structure containing information about the optimization process.

relativegap	Relative percentage difference between upper (U) and lower (L) bounds of the objective function that `intlinprog` calculates in its branch-and-bound algorithm.
	relativegap = 100*(U - L) / (abs(U) + 1)
	If `intcon = []`, `relativegap = []`.
	Note Although you specify `RelativeGapTolerance` as a decimal number, the iterative display and `output.relativegap` report the gap as a percentage, meaning 100 times the measured relative gap. If the exit message refers to the relative gap, this value is the measured relative gap, not a percentage.
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 IntegerTolerance 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.

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.

- If you include an `x0` argument, `intlinprog` uses that value in the `'rins'` and guided diving heuristics until it finds a better integer-feasible point. So when you provide `x0`, you can obtain good results by setting the `'Heuristics'` option to `'rins-diving'` or another setting that uses `'rins'`.

- 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,x0,options)
    ```

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `intlinprog`.

Compatibility Considerations

Default BranchRule is 'reliability'
Behavior changed in R2019a

The default value of the `BranchRule` option is `'reliability'` instead of `'maxpscost'`. In testing, this value gave better performance on many problems, both in solution times and in number of explored branching nodes.

On a few problems, the previous branch rule performs better. To get the previous behavior, set the `BranchRule` option to `'maxpscost'`.

See Also
Optimize | `linprog` | `mpsread` | `optimoptions` | `prob2struct`

Topics
"Mixed-Integer Linear Programming Basics: Solver-Based" on page 9-54
"Factory, Warehouse, Sales Allocation Model: Solver-Based" on page 9-57
"Traveling Salesman Problem: Solver-Based" on page 9-66

"Solve Sudoku Puzzles Via Integer Programming: Solver-Based" on page 9-89
"Mixed-Integer Quadratic Programming Portfolio Optimization: Solver-Based" on page 9-82
"Optimal Dispatch of Power Generators: Solver-Based" on page 9-72
"Mixed-Integer Linear Programming Algorithms" on page 9-43
"Tuning Integer Linear Programming" on page 9-52
"Solver-Based Optimization Problem Setup"

Introduced in R2014a

linprog

Solve linear programming problems

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,options)
x = linprog(problem)
[x,fval] = linprog( ___ )
[x,fval,exitflag,output] = linprog( ___ )
[x,fval,exitflag,output,lambda] = linprog( ___ )
```

Description

Linear programming solver

Finds the minimum of a problem specified by

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

f, *x*, *b*, *beq*, *lb*, and *ub* are vectors, and *A* and *Aeq* are matrices.

Note `linprog` applies only to the solver-based approach. For a discussion of the two optimization approaches, see "First Choose Problem-Based or Solver-Based Approach" on page 1-3.

`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 `fval` is `[]`.

`x = linprog(f,A,b,Aeq,beq,lb,ub,options)` minimizes with the optimization options specified by `options`. Use `optimoptions` to set these options.

`x = linprog(problem)` finds the minimum for `problem`, a structure described in `problem`.

You can import a `problem` structure from an MPS file using `mpsread`. You can also create a `problem` structure from an `OptimizationProblem` object by using `prob2struct`.

`[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:

$x(1) + x(2) \le 2$

$x(1) + x(2)/4 \le 1$

$x(1) - x(2) \le 2$

$-x(1)/4 - x(2) \le 1$

$-x(1) - x(2) \le -1$

$-x(1) + x(2) \le 2$.

```
A = [1 1
     1 1/4
     1 -1
     -1/4 -1
     -1 -1
     -1 1];
```

```
b = [2 1 2 1 -1 2];
```

Use the objective function $-x(1) - x(2)/3$.

```
f = [-1 -1/3];
```

Solve the linear program.

```
x = linprog(f,A,b)
```

```
Optimal solution found.
```

```
x = 2×1

    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)
```

```
Optimal solution found.
```

```
x = 2×1

     0
     2
```

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:

$$x(1) + x(2) \leq 2$$

$$x(1) + x(2)/4 \le 1$$

$$x(1) - x(2) \le 2$$

$$-x(1)/4 - x(2) \le 1$$

$$-x(1) - x(2) \le -1$$

$$-x(1) + x(2) \le 2.$$

```
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)
```

```
Optimal solution found.
```

```
x = 2×1
```

```
    0.1875
    1.2500
```

Linear Program Using the 'interior-point' Algorithm

Solve a linear program using the 'interior-point' algorithm.

For this example, use these linear inequality constraints:

$$x(1) + x(2) \le 2$$

$$x(1) + x(2)/4 \leq 1$$

$$x(1) - x(2) \leq 2$$

$$-x(1)/4 - x(2) \leq 1$$

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 2.$$

```
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;
```

Set these bounds:

$$-1 \leq x(1) \leq 1.5$$

$$-0.5 \leq x(2) \leq 1.25.$$

```
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 `'interior-point'` algorithm.

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

Solve the linear program using the `'interior-point'` algorithm.

```
x = linprog(f,A,b,Aeq,beq,lb,ub,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.

x = 2×1

    0.1875
    1.2500
```

Solve LP Using Problem-Based Approach for `linprog`

This example shows how to set up a problem using the problem-based approach and then solve it using the solver-based approach. The problem is

$$\max_{x}(x + y/3) \text{ subject to } \begin{cases} x + y \le 2 \\ x + y/4 \le 1 \\ x - y \le 2 \\ x/4 + y \ge -1 \\ x + y \ge 1 \\ -x + y \le 2 \\ x + y/4 = 1/2 \\ -1 \le x \le 1.5 \\ -1/2 \le y \le 1.25 \end{cases}$$

Create an `OptimizationProblem` object named `prob` to represent this problem.

```
x = optimvar('x','LowerBound',-1,'UpperBound',1.5);
y = optimvar('y','LowerBound',-1/2,'UpperBound',1.25);
prob = optimproblem('Objective',x + y/3,'ObjectiveSense','max');
prob.Constraints.c1 = x + y <= 2;
prob.Constraints.c2 = x + y/4 <= 1;
prob.Constraints.c3 = x - y <= 2;
prob.Constraints.c4 = x/4 + y >= -1;
prob.Constraints.c5 = x + y >= 1;
prob.Constraints.c6 = -x + y <= 2;
prob.Constraints.c7 = x + y/4 == 1/2;
```

Convert the problem object to a problem structure.

```
problem = prob2struct(prob);
```

Solve the resulting problem structure.

```
[sol,fval,exitflag,output] = linprog(problem)
```

```
Optimal solution found.

sol = 2×1

    0.1875
    1.2500


fval = -0.6042

exitflag = 1

output = struct with fields:
          iterations: 0
       constrviolation: 0
              message: 'Optimal solution found.'
            algorithm: 'dual-simplex'
         firstorderopt: 0
```

The returned `fval` is negative, even though the solution components are positive. Internally, `prob2struct` turns the maximization problem into a minimization problem of the negative of the objective function. See "Maximizing an Objective" on page 2-30.

Which component of `sol` corresponds to which optimization variable? Examine the `Variables` property of `prob`.

```
prob.Variables
```

```
ans = struct with fields:
    x: [1x1 optim.problemdef.OptimizationVariable]
    y: [1x1 optim.problemdef.OptimizationVariable]
```

As you might expect, `sol(1)` corresponds to x, and `sol(2)` corresponds to y. See "Algorithms" on page 16-406.

Return the Objective Function Value

Calculate the solution and objective function value for a simple linear program.

The inequality constraints are

$$x(1) + x(2) \leq 2$$

$$x(1) + x(2)/4 \leq 1$$

$$x(1) - x(2) \leq 2$$

$$-x(1)/4 - x(2) \leq 1$$

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 2 \, .$$

```
A = [1  1
     1  1/4
     1  -1
     -1/4  -1
     -1  -1
     -1  1];
```

```
b = [2 1 2 1 -1 2];
```

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)
```

```
Optimal solution found.
```

```
x = 2×1
```

```
     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');
```

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,options)

Optimal solution found.

x = 2×1

    0.1875
    1.2500


fval = -0.6042

exitflag = 1

output = struct with fields:
           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" on page 16-222, 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);
```

```
Optimal solution found.
```

Examine the solution and Lagrange multipliers.

```
x,lambda.ineqlin,lambda.lower
```

```
x = 3×1

         0
   15.0000
    3.0000
```

```
ans = 3×1

         0
    1.5000
    0.5000
```

```
ans = 3×1

    1.0000
         0
         0
```

`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 = 3×1

  -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.

Input Arguments

f — Coefficient vector
real vector | real array

Coefficient vector, specified as a real vector or real array. The coefficient vector represents the objective function `f'*x`. The notation assumes that `f` is a column vector, but you can use a row vector or array. Internally, `linprog` converts `f` to the column vector `f(:)`.

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 (length of `f`). 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, consider these inequalities:

$x_1 + 2x_2 \le 10$
$3x_1 + 4x_2 \le 20$
$5x_1 + 6x_2 \le 30.$

Specify the inequalities by entering the following 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`

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 (length of `f`). 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, consider these equalities:

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20.$

Specify the equalities by entering the following 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`

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, consider these inequalities:

$x_1 + 2x_2 \leq 10$
$3x_1 + 4x_2 \leq 20$
$5x_1 + 6x_2 \leq 30.$

Specify the inequalities by entering the following constraints.

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x components sum to 1 or less, use `A = ones(1,N)` and `b = 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 Me-by-N.

For example, consider these equalities:

$x_1 + 2x_2 + 3x_3 = 10$

$2x_1 + 4x_2 + x_3 = 20.$

Specify the equalities by entering the following constraints.

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use `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 length of `f` is equal to the length of `lb`, then `lb` specifies that

`x(i) >= lb(i)` for all `i`.

If `numel(lb) < numel(f)`, 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, use `lb = zeros(size(f))`.

Data Types: `double`

ub — Upper bounds
real vector | real array

Upper bounds, specified as a real vector or real array. If the length of `f` is equal to the length of `ub`, then `ub` specifies that

`x(i) <= ub(i)` for all `i`.

If `numel(ub) < numel(f)`, 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 1, use `ub = 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 15-6 for detailed information.

Some options are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

All Algorithms

Algorithm	Choose the optimization algorithm:

- `'dual-simplex'` (default)
- `'interior-point-legacy'`
- `'interior-point'`

For information on choosing the algorithm, see "Linear Programming Algorithms" on page 2-9.

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 3-14):

- `'final'` (default) displays just the final output.
- `'off'` or `'none'` displays no output.
- `'iter'` displays output at each iteration.

MaxIterations	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

See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9.

For `optimset`, the name is `MaxIter`. See "Current and Legacy Option Names" on page 15-22.

OptimalityTolerance	Termination tolerance on the dual feasibility, a positive scalar. The default is:

- `1e-8` for the `'interior-point-legacy'` algorithm
- `1e-7` for the `'dual-simplex'` algorithm
- `1e-6` for the `'interior-point'` algorithm

For `optimset`, the name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.

interior-point Algorithm

ConstraintTolerance	Feasibility tolerance for constraints, a scalar from `1e-10` through `1e-3`. `ConstraintTolerance` measures primal feasibility tolerance. The default is `1e-6`.
	For `optimset`, the name is `TolCon`. See "Current and Legacy Option Names" on page 15-22.
Preprocess	Level of LP preprocessing prior to algorithm iterations. Specify `'basic'` (default) or `'none'`.

Dual-Simplex Algorithm

ConstraintTolerance	Feasibility tolerance for constraints, a scalar from `1e-10` through `1e-3`. `ConstraintTolerance` measures primal feasibility tolerance. The default is `1e-4`.
	For `optimset`, the name is `TolCon`. See "Current and Legacy Option Names" on page 15-22.
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'`.

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
solver	`'linprog'`
options	Options created with `optimoptions`

You must supply at least the `solver` field in the `problem` structure.

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.

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.

3	The solution is feasible with respect to the relative `ConstraintTolerance` tolerance, but is not feasible with respect to the absolute tolerance.
1	Function converged to a solution x.
0	Number of iterations exceeded `options.MaxIterations` or solution time in seconds exceeded `options.MaxTime`.
-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.
-9	Solver lost feasibility.

Exitflags 3 and -9 relate to solutions that have large infeasibilities. These usually arise from linear constraint matrices that have large condition number, or problems that have large solution components. To correct these issues, try to scale the coefficient matrices, eliminate redundant linear constraints, or give tighter bounds on the variables.

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`

The Lagrange multipliers for linear constraints satisfy this equation with `length(f)` components:

$$f + A^T \lambda_{\text{ineqlin}} + Aeq^T \lambda_{\text{eqlin}} + \lambda_{\text{upper}} - \lambda_{\text{lower}} = 0,$$

based on the Lagrangian

$$f^T x + \lambda_{\text{ineqlin}}^T (Ax - b) + \lambda_{\text{eqlin}}^T (Aeq\, x - beq) + \lambda_{\text{upper}}^T (x - ub) + \lambda_{\text{lower}}^T (lb - x).$$

This sign convention matches that of nonlinear solvers (see "Constrained Optimality Theory" on page 3-12). However, this sign is the opposite of the sign in much linear programming literature, so a `linprog` Lagrange multiplier is the negative of the associated "shadow price."

Algorithms

Dual-Simplex Algorithm

For a description, see "Dual-Simplex Algorithm" on page 9-9.

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 9-6.

The first stage of the algorithm might involve some preprocessing of the constraints (see "Interior-Point-Legacy Linear Programming" on page 9-6). 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 9-6.) 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 < OptimalityTolerance.)`
- `The primal appears to be infeasible (and the dual unbounded). (The dual residual < OptimalityTolerance.)`
- `The dual appears to be infeasible (and the primal unbounded) since the dual residual > sqrt(OptimalityTolerance). (The primal residual < 10*OptimalityTolerance.)`
- `The primal appears to be infeasible (and the dual unbounded) since the primal residual > sqrt(OptimalityTolerance). (The dual residual < 10*OptimalityTolerance.)`
- `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 9-2.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `linprog`.

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

Optimize | intlinprog | mpsread | optimoptions | prob2struct | quadprog

Topics
"Set Up a Linear Program, Solver-Based" on page 1-21
"Typical Linear Programming Problem" on page 9-13
"Maximize Long-Term Investments Using Linear Programming: Solver-Based" on page 9-15
"Solver-Based Optimization Problem Setup"
"Linear Programming Algorithms" on page 9-2

Introduced before R2006a

lsqcurvefit

Solve nonlinear curve-fitting (data-fitting) problems in least-squares sense

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

Nonlinear least-squares solver

Find coefficients *x* that solve the problem

$$\min_{x}\|F(x, xdata) - ydata\|_2^2 = \min_{x}\sum_{i}(F(x, xdata_i) - ydata_i)^2,$$

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 2-31.

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

$$F(x, xdata) = \begin{bmatrix} F(x, xdata(1)) \\ F(x, xdata(2)) \\ \vdots \\ F(x, xdata(k)) \end{bmatrix}.$$

`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 2-57 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 ≤ 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 = 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, a structure described in problem.

[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

ydata = $x(1)\exp(x(2)xdata)$.

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 value of the function tolerance.

x = *1×2*

  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')
```
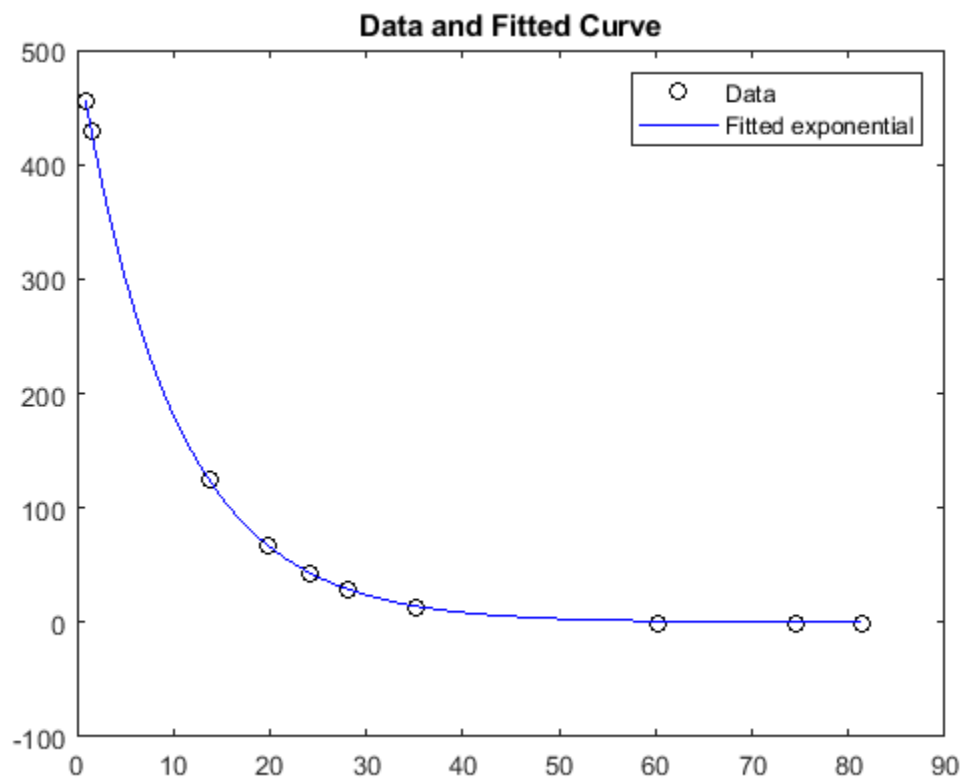

Best Fit with Bound Constraints

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 value of the optimality tolerance.

x = *1×2*

    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')
```
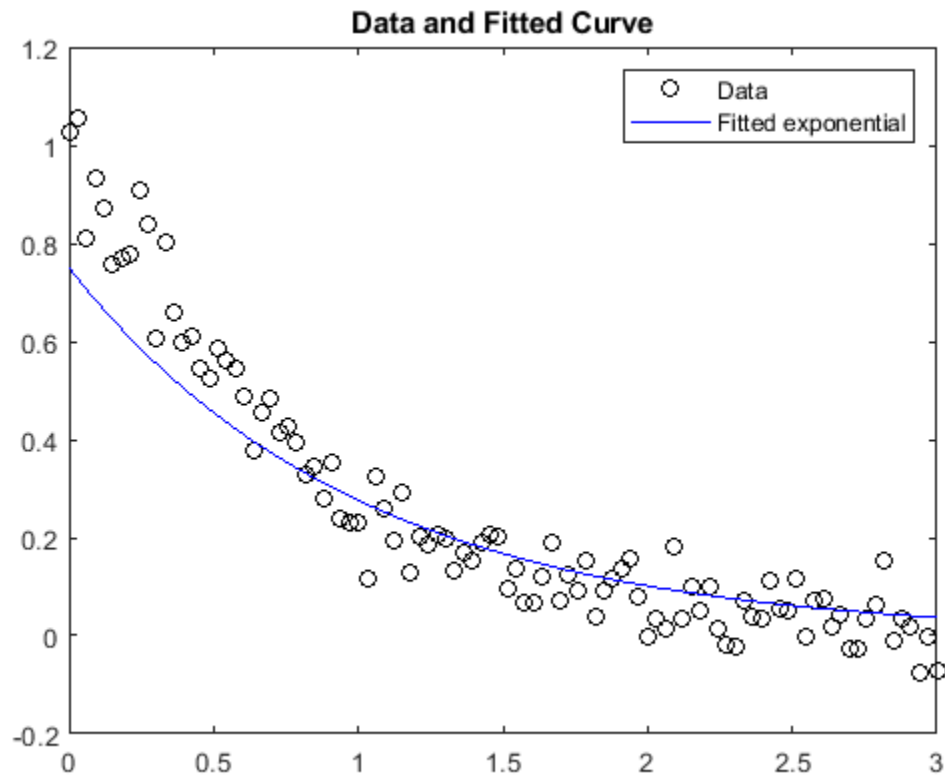
Compare Algorithms

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

$$ydata = x(1)exp(x(2)xdata).$$

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 value of the function tolerance.

x = *1×2*

  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 value of the step size tolerance.

x = *1×2*

  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')
```

Data and Fitted Curve
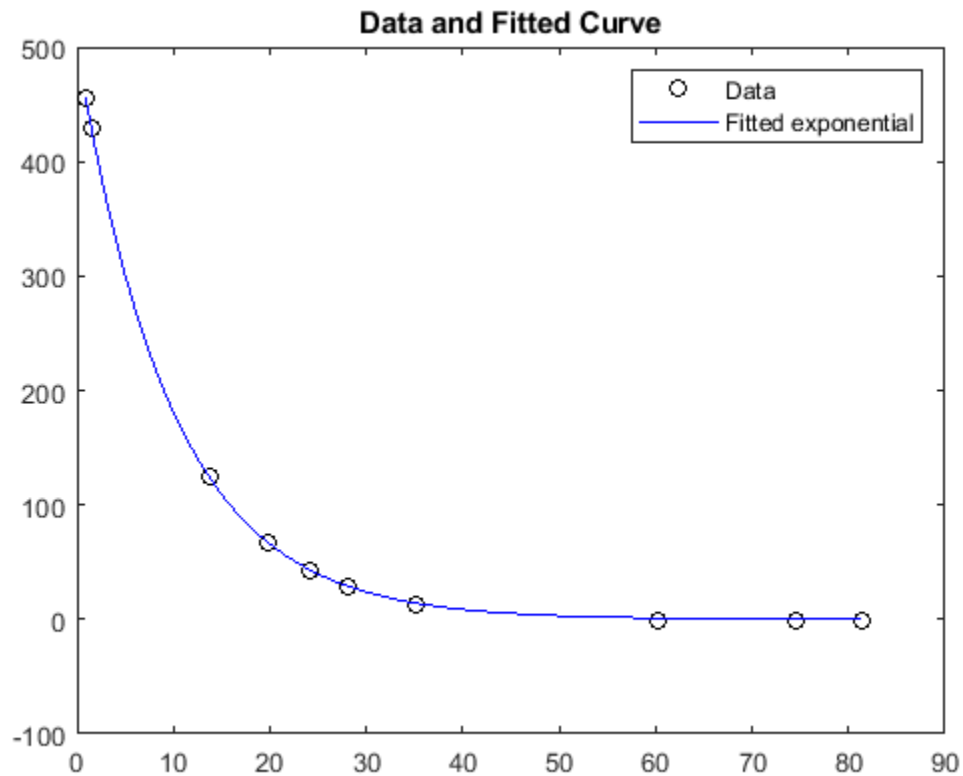
Compare Algorithms and Examine Solution Process

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

$$ydata = x(1)\exp(x(2)xdata).$$

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 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,options);
```

Local minimum possible.
lsqcurvefit stopped because the relative size of the current step is less than
the value of the step size tolerance.

Are the solutions equivalent?

```
norm(x-x2)
```

ans = 2.0626e-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
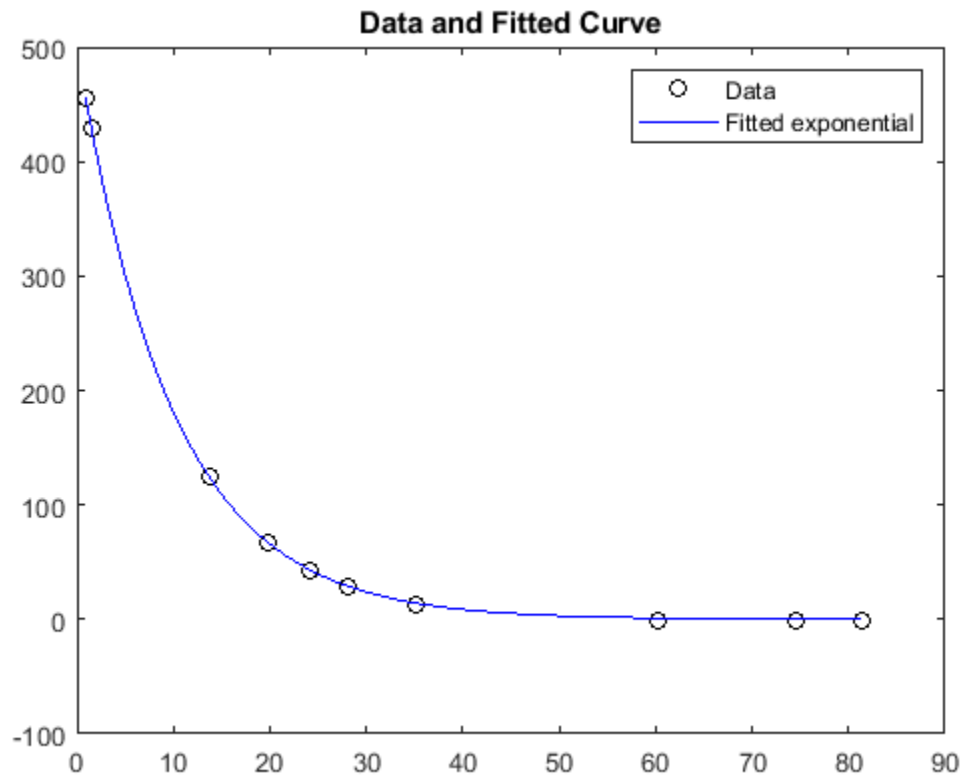
The fit looks good. How large are the residuals?

```
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.
```

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
```

16-243

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 arrays, they are converted to vectors using linear indexing (see "Array 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 16-0   .

If the Jacobian can also be computed *and* the 'SpecifyObjectiveGradient' option is true, set by

```
options = optimoptions('lsqcurvefit','SpecifyObjectiveGradient',true)
```

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 n elements, where n is the number of elements 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 2-26.

Example: @(x,xdata)x(1)*exp(-x(2)*xdata)

Data Types: char | function_handle | string

x0 — Initial point
real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in x0 and the 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 the number of elements in `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)`.

If there are fewer elements in `lb` than in `x0`, solvers issue a warning.

Example: To specify that all x components are positive, use `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 the number of elements in `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)`.

If there are fewer elements in `ub` than in `x0`, solvers issue a warning.

Example: To specify that all x components are less than 1, use `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 15-6 for detailed information.

Some options are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

All Algorithms

Algorithm	Choose between `'trust-region-reflective'` (default) and `'levenberg-marquardt'`.
	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. For more information on choosing the algorithm, see "Choosing the Algorithm" on page 2-6.
CheckGradients	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. Choices are `false` (default) or `true`.
	For `optimset`, the name is `DerivativeCheck` and the values are `'on'` or `'off'`. See "Current and Legacy Option Names" on page 15-22.
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 3-14):
	• `'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.

FiniteDifferenceStepSize	Scalar or vector step size factor for finite differences. When you set FiniteDifferenceStepSize to a vector v, the forward finite differences 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 FiniteDifferenceStepSize expands to a vector. The default is sqrt(eps) for forward finite differences, and eps^(1/3) for central finite differences.
	For optimset, the name is FinDiffRelStep. See "Current and Legacy Option Names" on page 15-22.
FiniteDifferenceType	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.
	For optimset, the name is FinDiffType. See "Current and Legacy Option Names" on page 15-22.
FunctionTolerance	Termination tolerance on the function value, a positive scalar. The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68.
	For optimset, the name is TolFun. See "Current and Legacy Option Names" on page 15-22.
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.
MaxFunctionEvaluations	Maximum number of function evaluations allowed, a positive integer. The default is 100*numberOfVariables. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9.
	For optimset, the name is MaxFunEvals. See "Current and Legacy Option Names" on page 15-22.
MaxIterations	Maximum number of iterations allowed, a positive integer. The default is 400. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9.
	For optimset, the name is MaxIter. See "Current and Legacy Option Names" on page 15-22.

OptimalityTolerance	Termination tolerance on the first-order optimality (a positive scalar). The default is 1e-6. See "First-Order Optimality Measure" on page 3-11.
	Internally, the 'levenberg-marquardt' algorithm uses an optimality tolerance (stopping criterion) of 1e-4 times FunctionTolerance and does not use OptimalityTolerance.
	For optimset, the name is TolFun. See "Current and Legacy Option Names" on page 15-22.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration. Pass a function handle or a cell array of function handles. The default is none ([]). See "Output Function and Plot Function Syntax" on page 15-27.
PlotFcn	Plots various measures of progress while the algorithm executes; select from predefined plots or write your own. Pass a name, a function handle, or a cell array of names or function handles. For custom plot functions, pass 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.
	Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27.
	For optimset, the name is PlotFcns. See "Current and Legacy Option Names" on page 15-22.
SpecifyObjectiveGradient	If false (default), the solver approximates the Jacobian using finite differences. If true, the solver uses a user-defined Jacobian (defined in fun), or Jacobian information (when using JacobMult), for the objective function.
	For optimset, the name is Jacobian, and the values are 'on' or 'off'. See "Current and Legacy Option Names" on page 15-22.
StepTolerance	Termination tolerance on x, a positive scalar. The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68.
	For optimset, the name is TolX. See "Current and Legacy Option Names" on page 15-22.
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.

UseParallel	When `true`, the solver estimates gradients in parallel. Disable by setting to the default, `false`. See "Parallel Computing".

Trust-Region-Reflective Algorithm

JacobianMultiplyFcn	Jacobian multiply function, specified as a function handle. 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 2-57 for information on how to supply values for any additional parameters `jmfun` needs.

Note `'SpecifyObjectiveGradient'` must be set to `true` for the solver to pass `Jinfo` from `fun` to `jmfun`.

See "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-93 and "Jacobian Multiply Function with Linear Least Squares" on page 12-30 for similar examples.

For `optimset`, the name is `JacobMult`. See "Current and Legacy Option Names" on page 15-22.

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 $\partial fun(i)/\partial x(j) \neq 0$.
	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 12-5.
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.
`SubproblemAlgorithm`	Determines how the iteration step is calculated. The default, `'factorization'`, takes a slower but more accurate step than `'cg'`. See "Trust-Region-Reflective Least Squares" on page 12-3.
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 12-6.
ScaleProblem	`'jacobian'` can sometimes improve the convergence of a poorly scaled problem; the default is `'none'`.

Example: `options = optimoptions('lsqcurvefit','FiniteDifferenceType','central')`

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

Field Name	Entry
`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.

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 4-18.

resnorm — Squared norm of the residual
nonnegative real

Squared 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
array

Value of objective function at solution, returned as an array. 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` is less than the specified tolerance, or Jacobian at `x` is undefined.
`3`	Change in the residual is less than the specified tolerance.
`4`	Relative magnitude of search direction is smaller than the step tolerance.
`0`	Number of iterations exceeds `options.MaxIterations` or number of function evaluations exceeded `options.MaxFunctionEvaluations`.
`-1`	A plot function or output function stopped the solver.

-2	Problem is infeasible: the bounds `lb` and `ub` are inconsistent.

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 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.

- `lsqcurvefit` can solve complex-valued problems directly. Noe that bound constraints do not make sense for complex values. For a complex problem with bound constraints, split the variables into real and imaginary parts. See "Fit a Model to Complex-Valued Data" on page 12-50.

- 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.

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 12-3.
- The Levenberg-Marquardt method is described in references [4], [5], and [6]. See "Levenberg-Marquardt Method" on page 12-6.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `lsqcurvefit`.

References

[1] Coleman, T.F. and Y. Li. "An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds." *SIAM Journal on Optimization*, Vol. 6, 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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

- `lsqcurvefit` and `lsqnonlin` support code generation using either the `codegen` function or the MATLAB Coder app. You must have a MATLAB Coder license to generate code.

- The target hardware must support standard double-precision floating-point computations. You cannot generate code for single-precision or fixed-point computations.

- Code generation targets do not use the same math kernel libraries as MATLAB solvers. Therefore, code generation solutions can vary from solver solutions, especially for poorly conditioned problems.

- All code for generation must be MATLAB code. In particular, you cannot use a custom black-box function as an objective function for `lsqcurvefit` or `lsqnonlin`. You can use `coder.ceval` to evaluate a custom function coded in C or C++. However, the custom function must be called in a MATLAB function.

- `lsqcurvefit` and `lsqnonlin` do not support the `problem` argument for code generation.

  ```
  [x,fval] = lsqnonlin(problem) % Not supported
  ```

- You must specify the objective function by using function handles, not strings or character names.

  ```
  x = lsqnonlin(@fun,x0,lb,ub,options) % Supported
  % Not supported: lsqnonlin('fun',...) or lsqnonlin("fun",...)
  ```

- All input matrices `lb` and `ub` must be full, not sparse. You can convert sparse matrices to full by using the `full` function.

- The `lb` and `ub` arguments must have the same number of entries as the `x0` argument or must be empty `[]`.

- For advanced code optimization involving embedded processors, you also need an Embedded Coder license.

- You must include options for `lsqcurvefit` or `lsqnonlin` and specify them using `optimoptions`. The options must include the `Algorithm` option, set to `'levenberg-marquardt'`.

  ```
  options = optimoptions('lsqnonlin','Algorithm','levenberg-marquardt');
  [x,fval,exitflag] = lsqnonlin(fun,x0,lb,ub,options);
  ```

- Code generation supports these options:

  - `Algorithm` — Must be `'levenberg-marquardt'`
  - `FiniteDifferenceStepSize`
  - `FiniteDifferenceType`
  - `FunctionTolerance`
  - `MaxFunctionEvaluations`
  - `MaxIterations`
  - `SpecifyObjectiveGradient`
  - `StepTolerance`
  - `TypicalX`

- Generated code has limited error checking for options. The recommended way to update an option is to use `optimoptions`, not dot notation.

```
opts = optimoptions('lsqnonlin','Algorithm','levenberg-marquardt');
opts = optimoptions(opts,'MaxIterations',1e4); % Recommended
opts.MaxIterations = 1e4; % Not recommended
```

- Do not load options from a file. Doing so can cause code generation to fail. Instead, create options in your code.

- Usually, if you specify an option that is not supported, the option is silently ignored during code generation. However, if you specify a plot function or output function by using dot notation, code generation can issue an error. For reliability, specify only supported options.

- Because output functions and plot functions are not supported, solvers do not return the exit flag – 1.

For an example, see "Generate Code for lsqcurvefit or lsqnonlin" on page 12-89.

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the 'UseParallel' option to true.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see "Using Parallel Computing in Optimization Toolbox" on page 14-5.

See Also
Optimize | fsolve | lsqnonlin | optimoptions

Topics
"Nonlinear Least Squares (Curve Fitting)"
"Solver-Based Optimization Problem Setup"
"Least-Squares (Model Fitting) Algorithms" on page 12-2

Introduced before R2006a

lsqlin

Solve constrained linear least-squares problems

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

Linear least-squares solver with bounds or linear constraints.

Solves least-squares curve fitting problems of the form

$$\min_{x} \frac{1}{2}\|C \cdot x - d\|_2^2 \text{ such that } \begin{cases} A \cdot x \le b, \\ Aeq \cdot x = beq, \\ lb \le x \le ub. \end{cases}$$

Note `lsqlin` applies only to the solver-based approach. For a discussion of the two optimization approaches, see "First Choose Problem-Based or Solver-Based Approach" on page 1-3.

`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`, a structure described in `problem`. Create the `problem` structure using dot notation or the `struct` function. Or create a `problem` structure from an `OptimizationProblem` object by using `prob2struct`.

`[x,resnorm,residual,exitflag,output,lambda] = lsqlin( ___ )`, for any input arguments described above, returns:

- The squared 2-norm of the residual `resnorm` $= \|C \cdot x - d\|_2^2$

- 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)
```

Minimum found that satisfies the constraints.

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

x = 4×1

    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 equality and 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)
```

```
Minimum found that satisfies the constraints.

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

x = 4×1

    -0.1000
    -0.1000
     0.1599
     0.4090
```

Linear Least Squares with Nondefault Options

This example shows how to use nondefault options for linear least squares.

Set options to use the `'interior-point'` algorithm and to give iterative display.

```
options = optimoptions('lsqlin','Algorithm','interior-point','Display','iter');
```

Set up a linear least-squares problem.

```
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];
```

Run the problem.

```
x = lsqlin(C,d,A,b,[],[],[],[],[],options)
```

```
 Iter             Fval  Primal Infeas    Dual Infeas  Complementarity
    0   -7.687420e-02   1.600492e+00   6.150431e-01     1.000000e+00
    1   -7.687419e-02   8.002458e-04   3.075216e-04     2.430833e-01
    2   -3.162837e-01   4.001229e-07   1.537608e-07     5.945636e-02
    3   -3.760545e-01   2.000615e-10   2.036997e-08     1.370933e-02
    4   -3.912129e-01   9.997558e-14   1.006816e-08     2.548273e-03
    5   -3.948062e-01   0.000000e+00   2.955102e-09     4.295807e-04
    6   -3.953277e-01   1.110223e-16   1.237758e-09     3.102850e-05
    7   -3.953581e-01   1.665335e-16   1.645863e-10     1.138719e-07
    8   -3.953582e-01   5.551115e-17   2.400025e-13     5.693290e-11
```

```
Minimum found that satisfies the constraints.

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

```
x = 4×1

    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 value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

x = 4×1

    -0.1000
    -0.1000
     0.2152
     0.3502


resnorm = 0.1672

residual = 5×1

     0.0455
     0.0764
    -0.3562
     0.1620
     0.0784


exitflag = 1

output = struct with fields:
              message: '...'
            algorithm: 'interior-point'
        firstorderopt: 4.3374e-11
        constrviolation: 0
            iterations: 6
          linearsolver: 'dense'
          cgiterations: []
```

```
lambda = struct with fields:
    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 = *3×1*

```
    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 = *1×2*

```
    0.2026    0.2026
```

Now examine the Lagrange multipliers for the lower and upper bound constraints.

```
lambda.lower
```

ans = *4×1*

```
    0.0409
    0.2784
    0.0000
    0.0000
```

```
lambda.upper
```

ans = *4×1*

```
     0
     0
     0
     0
```

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.

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 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

$x_1 + 2x_2 \leq 10$
$3x_1 + 4x_2 \leq 20$
$5x_1 + 6x_2 \leq 30,$

enter 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, use 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, consider these inequalities:

$x_1 + 2x_2 \le 10$
$3x_1 + 4x_2 \le 20$
$5x_1 + 6x_2 \le 30.$

Specify the inequalities by entering the following constraints.

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x components sum to 1 or less, use `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

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20,$

enter these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use `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 Me-by-N.

For example, consider these equalities:

$$x_1 + 2x_2 + 3x_3 = 10$$
$$2x_1 + 4x_2 + x_3 = 20.$$

Specify the equalities by entering the following constraints.

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use Aeq = ones(1,N) and beq = 1.

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, lsqlin 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, lsqlin converts an array ub to the vector ub(:).

Example: ub = [Inf;4;10] means x(2) ≤ 4, x(3) ≤ 10.

Data Types: `double`

x0 — Initial point
[] (default) | real vector or array

Initial point for the solution process, specified as a real vector or array. The `'trust-region-reflective'` and `'active-set'` algorithms use x0 (optional).

If you do not specify x0 for the `'trust-region-reflective'` or `'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` | structure such as created by `optimset`

Options for lsqlin, specified as the output of the `optimoptions` function or as a structure such as created by `optimset`.

Some options are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

All Algorithms

Algorithm	Choose the algorithm:

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

The `'trust-region-reflective'` algorithm allows only upper and lower bounds, no linear inequalities or equalities. If you specify both the `'trust-region-reflective'` algorithm and linear constraints, `lsqlin` uses the `'interior-point'` algorithm.

The `'trust-region-reflective'` algorithm does not allow equal upper and lower bounds.

When the problem has no constraints, `lsqlin` calls `mldivide` internally.

If you have a large number of linear constraints and not a large number of variables, try the `'active-set'` algorithm.

For more information on choosing the algorithm, see "Choosing the Algorithm" on page 2-6.

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.

MaxIterations — Maximum number of iterations allowed, a positive integer. The default value is 2000 for the `'active-set'` algorithm, and 200 for the other algorithms.

For `optimset`, the option name is `MaxIter`. See "Current and Legacy Option Names" on page 15-22.

trust-region-reflective Algorithm Options

FunctionTolerance	Termination tolerance on the function value, a positive scalar. The default is `100*eps`, about `2.2204e-14`.
	For `optimset`, the option name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.
JacobianMultiplyFcn	Jacobian multiply function, specified as a function handle. 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 2-57 for information on how to supply extra parameters if necessary.
	See "Jacobian Multiply Function with Linear Least Squares" on page 12-30 for an example.
	For `optimset`, the option name is `JacobMult`. See "Current and Legacy Option Names" on page 15-22.
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 16-271.
OptimalityTolerance	Termination tolerance on the first-order optimality, a positive scalar. The default is `100*eps`, about `2.2204e-14`. See "First-Order Optimality Measure" on page 3-11.
	For `optimset`, the option name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.
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 16-271.

SubproblemAlgorithm	Determines how the iteration step is calculated. The default, `'cg'`, takes a faster but less accurate step than `'factorization'`. See "Trust-Region-Reflective Least Squares" on page 12-3.
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 variables. 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

ConstraintTolerance	Tolerance on the constraint violation, a positive scalar. The default is `1e-8`.
	For `optimset`, the option name is `TolCon`. See "Current and Legacy Option Names" on page 15-22.
LinearSolver	Type of internal linear solver in algorithm: • `'auto'` (default) — Use `'sparse'` if the C matrix is sparse, `'dense'` otherwise. • `'sparse'` — Use sparse linear algebra. See "Sparse Matrices". • `'dense'` — Use dense linear algebra.
OptimalityTolerance	Termination tolerance on the first-order optimality, a positive scalar. The default is `1e-8`. See "First-Order Optimality Measure" on page 3-11.
	For `optimset`, the option name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.
StepTolerance	Termination tolerance on x, a positive scalar. The default is `1e-12`.
	For `optimset`, the option name is `TolX`. See "Current and Legacy Option Names" on page 15-22.

'active-set' Algorithm Options

ConstraintTolerance	Tolerance on the constraint violation, a positive scalar. The default value is `1e-8`. For `optimset`, the option name is `TolCon`. See "Current and Legacy Option Names" on page 15-22.
ObjectiveLimit	A tolerance (stopping criterion) that is a scalar. If the objective function value goes below `ObjectiveLimit` and the current point is feasible, the iterations halt because the problem is unbounded, presumably. The default value is `-1e20`.
OptimalityTolerance	Termination tolerance on the first-order optimality, a positive scalar. The default value is `1e-8`. See "First-Order Optimality Measure" on page 3-11. For `optimset`, the name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.
StepTolerance	Termination tolerance on x, a positive scalar. The default value is `1e-8`. For `optimset`, the option name is `TolX`. See "Current and Legacy Option Names" on page 15-22.

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`

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.

3	Change in the residual was smaller than the specified tolerance `options.FunctionTolerance`. (`trust-region-reflective` algorithm)
2	Step size smaller than `options.StepTolerance`, constraints satisfied. (`interior-point` algorithm)
1	Function converged to a solution x.
0	Number of iterations exceeded `options.MaxIterations`.
-2	The problem is infeasible. Or, for the `interior-point` algorithm, step size smaller than `options.StepTolerance`, but constraints are not satisfied.
-3	The problem is unbounded.
-4	Ill-conditioning prevents further optimization.
-8	Unable to compute a step direction.

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'`
	• `'trust-region-reflective'`
	• `'mldivide'` for an unconstrained problem
	For an unconstrained problem, `iterations = 0`, and the remaining entries in the `output` structure are empty.

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.
linearsolver	Type of internal linear solver, `'dense'` or `'sparse'` (`'interior-point'` algorithm only)
cgiterations	Number of conjugate gradient iterations the solver performed. Nonempty only for the `'trust-region-reflective'` algorithm.

See "Output Structures" on page 3-21.

`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 3-22.

Tips

- For problems with no constraints, you can use `mldivide` (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.
- If your problem has many linear constraints and few variables, try using the `'active-set'` algorithm. See "Quadratic Programming with Many Linear Constraints" on page 11-64.
- 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

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 12-3, and in particular "Large Scale Linear Least Squares" on page 12-5.

Interior-Point Algorithm

The `'interior-point'` algorithm is based on the `quadprog` `'interior-point-convex'` algorithm. See "Linear Least Squares: Interior-Point or Active-Set" on page 12-2.

Active-Set Algorithm

The `'active-set'` algorithm is based on the `quadprog` `'active-set'` algorithm. For more information, see "Linear Least Squares: Interior-Point or Active-Set" on page 12-2 and "active-set quadprog Algorithm" on page 11-11.

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.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `lsqlin`.

See Also
Optimize | `lsqnonneg` | `mldivide` | `quadprog`

Topics
"Nonnegative Linear Least Squares, Solver-Based" on page 12-25
"Optimize Live Editor Task with lsqlin Solver" on page 12-28
"Jacobian Multiply Function with Linear Least Squares" on page 12-30
"Least-Squares (Model Fitting) Algorithms" on page 12-2

Introduced before R2006a

lsqnonlin

Solve nonlinear least-squares (nonlinear data-fitting) problems

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

Nonlinear least-squares solver

Solves nonlinear least-squares curve fitting problems of the form

$$\min_{x} \|f(x)\|_2^2 = \min_{x} \left(f_1(x)^2 + f_2(x)^2 + \dots + f_n(x)^2 \right)$$

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 2-31.

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

$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix}.$$

`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 (or array) 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 2-57 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, a structure described in problem.

[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 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)')
```
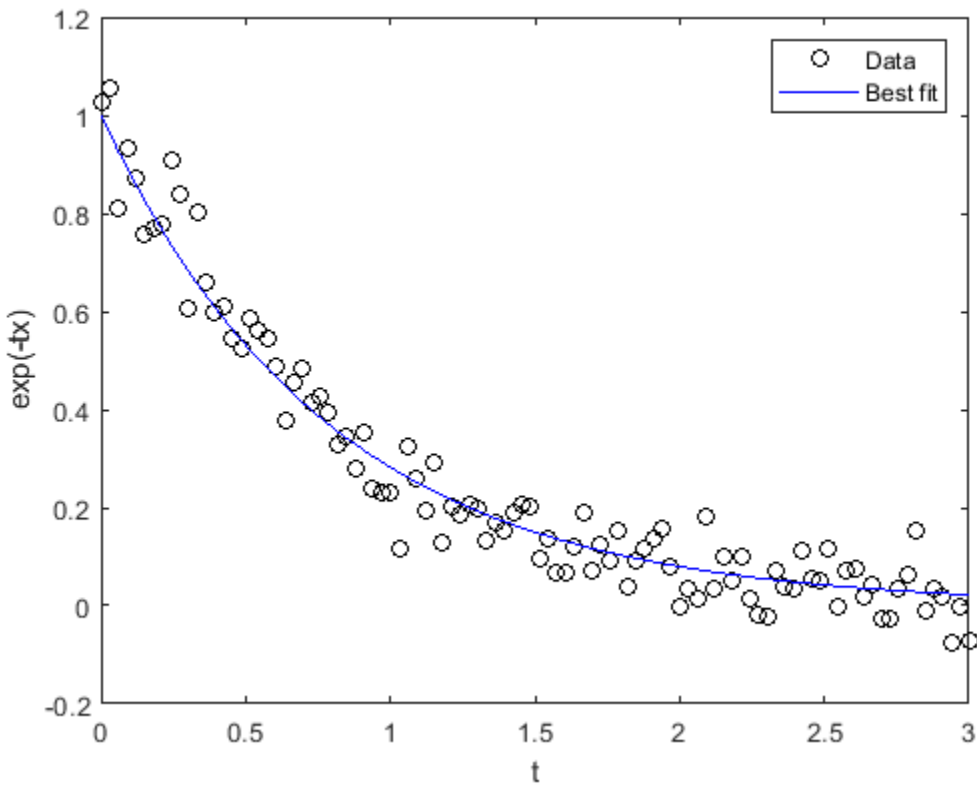

Fit a Problem with Bound Constraints

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,

$$\frac{1}{\sqrt{2\pi}}\exp(-t^2/2).$$

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 value of the function tolerance.

x = 1×2

    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')
```
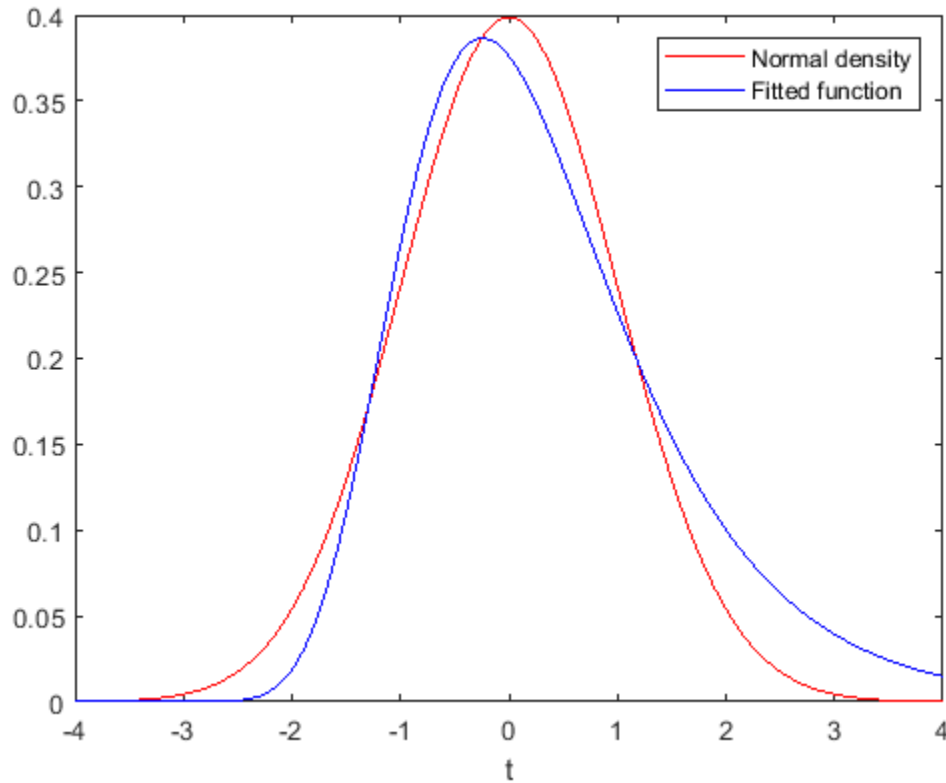
Nonlinear Least Squares with Nondefault Options

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

ydata = $x(1)\exp(x(2)$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. 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 value of the function tolerance.

x = 1×2

  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 value of the step size tolerance.

x = 1×2

  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
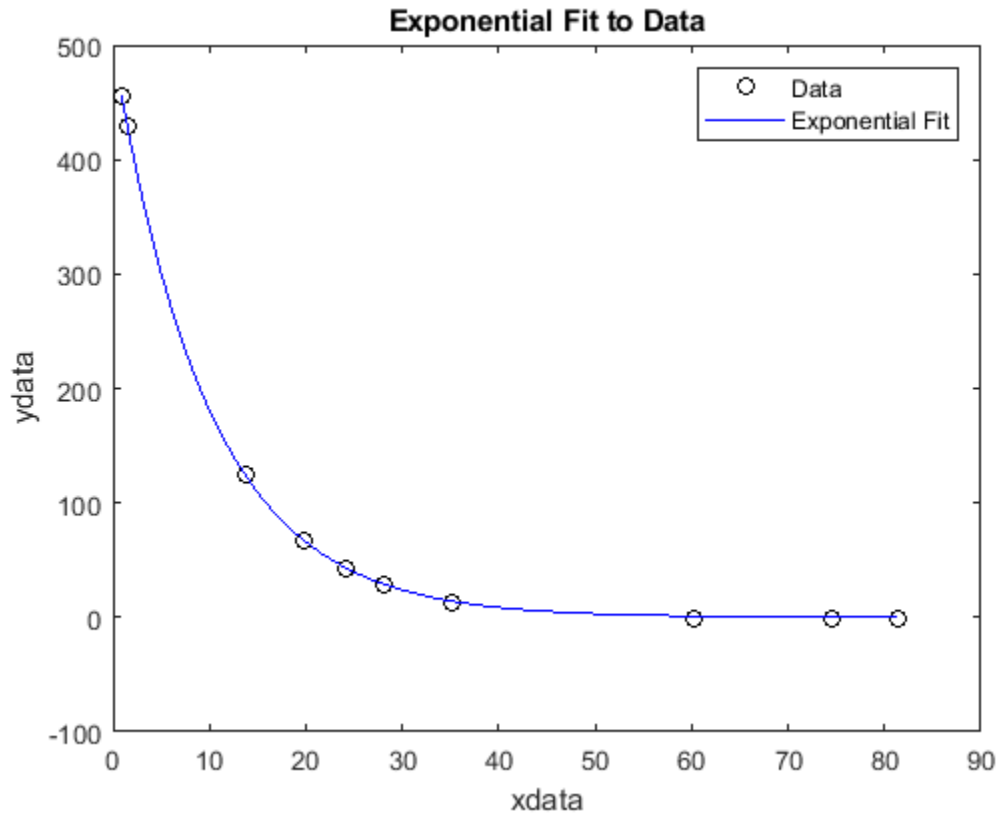
Nonlinear Least Squares Solution and Residual Norm

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).

The `myfun` function that computes the 10-component vector F appears at the end of this example on page 16-0 .

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)
```

```
Local minimum possible.
lsqnonlin stopped because the size of the current step is less than
the value of the step size tolerance.
```

x = *1×2*

```
   0.2578    0.2578
```

resnorm = 124.3622

The `resnorm` output is the squared residual norm, the sum of squares of the function values.

The following function computes the vector-valued objective function.

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

Examine the Solution Process

Examine the solution process both as it occurs (by setting the `Display` option to `'iter'`) and afterward (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

$$ydata = x(1)\exp(x(2)xdata).$$

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. 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
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   0.000462261        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 value of the function tolerance.

Examine the output structure to obtain more information about the solution process.

output

```
output = struct with fields:
    firstorderopt: 0.0114
       iterations: 28
        funcCount: 87
     cgiterations: 0
        algorithm: 'trust-region-reflective'
         stepsize: 4.6226e-04
          message: '...'
```

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
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       0.00944311
     9          42          243568        5.66e+04         100000       0.00821622
    10          45          243424        1.61e+04          10000       0.00777936
    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         4.96e-05          1e-05       5.07028e-05
```

```
Local minimum possible.
lsqnonlin stopped because the relative size of the current step is less than
the value of the step size tolerance.
```

The `'levenberg-marquardt'` converged with fewer iterations, but almost as many function evaluations:

```
output
```

```
output = struct with fields:
       iterations: 19
        funcCount: 72
         stepsize: 5.0703e-05
     cgiterations: []
    firstorderopt: 4.9629e-05
        algorithm: 'levenberg-marquardt'
          message: '...'
```

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 an array `x` and returns an array `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 arrays, they are converted to vectors using linear indexing (see "Array 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 16-0   .

If the Jacobian can also be computed *and* the `'SpecifyObjectiveGradient'` option is `true`, set by

```
options = optimoptions('lsqnonlin','SpecifyObjectiveGradient',true)
```

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 an array of m components and x has n elements, where n is the number of elements 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` | `string`

x0 — Initial point
real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in `x0` and the 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 the number of elements in `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)`.

If there are fewer elements in `lb` than in `x0`, solvers issue a warning.

Example: To specify that all x components are positive, use `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 the number of elements in `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).

If there are fewer elements in ub than in x0, solvers issue a warning.

Example: To specify that all x components are less than 1, use 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 15-6 for detailed information.

Some options are absent from the optimoptions display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

All Algorithms

Algorithm	Choose between 'trust-region-reflective' (default) and 'levenberg-marquardt'.
	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. For more information on choosing the algorithm, see "Choosing the Algorithm" on page 2-6.
CheckGradients	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. Choices are false (default) or true.
	For optimset, the name is DerivativeCheck and the values are 'on' or 'off'. See "Current and Legacy Option Names" on page 15-22.
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 3-14):
	• `'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.
FiniteDifferenceStepSize	Scalar or vector step size factor for finite differences. When you set `FiniteDifferenceStepSize` to a vector `v`, the forward finite differences `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 `FiniteDifferenceStepSize` expands to a vector. The default is `sqrt(eps)` for forward finite differences, and `eps^(1/3)` for central finite differences.
	For `optimset`, the name is `FinDiffRelStep`. See "Current and Legacy Option Names" on page 15-22.
FiniteDifferenceType	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.
	For `optimset`, the name is `FinDiffType`. See "Current and Legacy Option Names" on page 15-22.
FunctionTolerance	Termination tolerance on the function value, a positive scalar. The default is `1e-6`. See "Tolerances and Stopping Criteria" on page 2-68.
	For `optimset`, the name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.
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.

MaxFunctionEvaluations	Maximum number of function evaluations allowed, a positive integer. The default is `100*numberOfVariables`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9. For `optimset`, the name is `MaxFunEvals`. See "Current and Legacy Option Names" on page 15-22.
MaxIterations	Maximum number of iterations allowed, a positive integer. The default is `400`. See "Tolerances and Stopping Criteria" on page 2-68 and "Iterations and Function Counts" on page 3-9. For `optimset`, the name is `MaxIter`. See "Current and Legacy Option Names" on page 15-22.
OptimalityTolerance	Termination tolerance on the first-order optimality (a positive scalar). The default is `1e-6`. See "First-Order Optimality Measure" on page 3-11. Internally, the `'levenberg-marquardt'` algorithm uses an optimality tolerance (stopping criterion) of `1e-4` times `FunctionTolerance` and does not use `OptimalityTolerance`. For `optimset`, the name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.
OutputFcn	Specify one or more user-defined functions that an optimization function calls at each iteration. Pass a function handle or a cell array of function handles. The default is none (`[]`). See "Output Function and Plot Function Syntax" on page 15-27.
PlotFcn	Plots various measures of progress while the algorithm executes; select from predefined plots or write your own. Pass a name, a function handle, or a cell array of names or function handles. For custom plot functions, pass 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. Custom plot functions use the same syntax as output functions. See "Output Functions for Optimization Toolbox™" on page 3-30 and "Output Function and Plot Function Syntax" on page 15-27. For `optimset`, the name is `PlotFcns`. See "Current and Legacy Option Names" on page 15-22.

SpecifyObjectiveGradient	If false (default), the solver approximates the Jacobian using finite differences. If true, the solver uses a user-defined Jacobian (defined in fun), or Jacobian information (when using JacobMult), for the objective function.
	For optimset, the name is Jacobian, and the values are 'on' or 'off'. See "Current and Legacy Option Names" on page 15-22.
StepTolerance	Termination tolerance on x, a positive scalar. The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-68.
	For optimset, the name is TolX. See "Current and Legacy Option Names" on page 15-22.
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.
UseParallel	When true, the solver estimates gradients in parallel. Disable by setting to the default, false. See "Parallel Computing".

Trust-Region-Reflective Algorithm

JacobianMultiplyFcn	Jacobian multiply function, specified as a function handle. 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 2-57 for information on how to supply values for any additional parameters `jmfun` needs.

Note 'SpecifyObjectiveGradient' must be set to `true` for the solver to pass `Jinfo` from `fun` to `jmfun`.

See "Minimization with Dense Structured Hessian, Linear Equalities" on page 6-93 and "Jacobian Multiply Function with Linear Least Squares" on page 12-30 for similar examples.

For `optimset`, the name is `JacobMult`. See "Current and Legacy Option Names" on page 15-22.

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 $\partial fun(i)/\partial x(j) \neq 0$.

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 12-5.
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.
`SubproblemAlgorithm`	Determines how the iteration step is calculated. The default, `'factorization'`, takes a slower but more accurate step than `'cg'`. See "Trust-Region-Reflective Least Squares" on page 12-3.
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 12-6.
ScaleProblem	`'jacobian'` can sometimes improve the convergence of a poorly scaled problem; the default is `'none'`.

Example: `options = optimoptions('lsqnonlin','FiniteDifferenceType','central')`

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.

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 4-18.

resnorm — Squared norm of the residual
nonnegative real

Squared 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
array

Value of objective function at solution, returned as an array. 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 is less than the specified tolerance, or Jacobian at x is undefined.
3	Change in the residual is less than the specified tolerance.
4	Relative magnitude of search direction is smaller than the step tolerance.
0	Number of iterations exceeds options.MaxIterations or number of function evaluations exceeded options.MaxFunctionEvaluations.
-1	A plot function or output function stopped the solver.
-2	Problem is infeasible: the bounds lb and ub are inconsistent.

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 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.

*   `lsqnonlin` can solve complex-valued problems directly. Noe that bound constraints do not make sense for complex values. For a complex problem with bound constraints, split the variables into real and imaginary parts. See "Fit a Model to Complex-Valued Data" on page 12-50.

*   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, `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.

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 12-3.

*   The Levenberg-Marquardt method is described in references [4], [5], and [6]. See "Levenberg-Marquardt Method" on page 12-6.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `lsqnonlin`.

References

[1] Coleman, T.F. and Y. Li. "An Interior, Trust Region Approach for Nonlinear Minimization Subject to Bounds." *SIAM Journal on Optimization*, Vol. 6, 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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

- `lsqcurvefit` and `lsqnonlin` support code generation using either the `codegen` function or the MATLAB Coder app. You must have a MATLAB Coder license to generate code.
- The target hardware must support standard double-precision floating-point computations. You cannot generate code for single-precision or fixed-point computations.
- Code generation targets do not use the same math kernel libraries as MATLAB solvers. Therefore, code generation solutions can vary from solver solutions, especially for poorly conditioned problems.
- All code for generation must be MATLAB code. In particular, you cannot use a custom black-box function as an objective function for `lsqcurvefit` or `lsqnonlin`. You can use `coder.ceval` to evaluate a custom function coded in C or C++. However, the custom function must be called in a MATLAB function.
- `lsqcurvefit` and `lsqnonlin` do not support the `problem` argument for code generation.

```
[x,fval] = lsqnonlin(problem) % Not supported
```

- You must specify the objective function by using function handles, not strings or character names.

```
x = lsqnonlin(@fun,x0,lb,ub,options) % Supported
% Not supported: lsqnonlin('fun',...) or lsqnonlin("fun",...)
```

- All input matrices `lb` and `ub` must be full, not sparse. You can convert sparse matrices to full by using the `full` function.

- The `lb` and `ub` arguments must have the same number of entries as the `x0` argument or must be empty `[]`.

- For advanced code optimization involving embedded processors, you also need an Embedded Coder license.

- You must include options for `lsqcurvefit` or `lsqnonlin` and specify them using `optimoptions`. The options must include the `Algorithm` option, set to `'levenberg-marquardt'`.

```
options = optimoptions('lsqnonlin','Algorithm','levenberg-marquardt');
[x,fval,exitflag] = lsqnonlin(fun,x0,lb,ub,options);
```

- Code generation supports these options:

  - `Algorithm` — Must be `'levenberg-marquardt'`
  - `FiniteDifferenceStepSize`
  - `FiniteDifferenceType`
  - `FunctionTolerance`
  - `MaxFunctionEvaluations`
  - `MaxIterations`
  - `SpecifyObjectiveGradient`
  - `StepTolerance`
  - `TypicalX`

- Generated code has limited error checking for options. The recommended way to update an option is to use `optimoptions`, not dot notation.

```
opts = optimoptions('lsqnonlin','Algorithm','levenberg-marquardt');
opts = optimoptions(opts,'MaxIterations',1e4); % Recommended
opts.MaxIterations = 1e4; % Not recommended
```

- Do not load options from a file. Doing so can cause code generation to fail. Instead, create options in your code.

- Usually, if you specify an option that is not supported, the option is silently ignored during code generation. However, if you specify a plot function or output function by using dot notation, code generation can issue an error. For reliability, specify only supported options.

- Because output functions and plot functions are not supported, solvers do not return the exit flag – 1.

For an example, see "Generate Code for lsqcurvefit or lsqnonlin" on page 12-89.

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the `'UseParallel'` option to `true`.

```
options = optimoptions('solvername','UseParallel',true)
```

For more information, see "Using Parallel Computing in Optimization Toolbox" on page 14-5.

See Also
Optimize | fsolve | lsqcurvefit | optimoptions

Topics
"Nonlinear Least Squares (Curve Fitting)"
"Solver-Based Optimization Problem Setup"
"Least-Squares (Model Fitting) Algorithms" on page 12-2

Introduced before R2006a

lsqnonneg

Solve nonnegative linear least-squares problem

Syntax

```
x = lsqnonneg(C,d)
x = lsqnonneg(C,d,options)
x = lsqnonneg(problem)
[x,resnorm,residual] = lsqnonneg( ___ )
[x,resnorm,residual,exitflag,output] = lsqnonneg( ___ )
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg( ___ )
```

Description

Solve nonnegative least-squares curve fitting problems of the form

$$\min_{x} \lVert C \cdot x - d \rVert_2^2, \text{ where } x \geq 0.$$

Note `lsqnonneg` applies only to the solver-based approach. For a discussion of the two optimization approaches, see "First Choose Problem-Based or Solver-Based Approach" on page 1-3.

`x = lsqnonneg(C,d)` returns the vector `x` that minimizes `norm(C*x-d)` subject to `x ≥ 0`. Arguments `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`, a structure described in `problem`.

`[x,resnorm,residual] = lsqnonneg( ___ )`, for any previous syntax, additionally returns the value of the squared 2-norm of the residual, `norm(C*x-d)^2`, and returns the residual `d-C*x`.

`[x,resnorm,residual,exitflag,output] = lsqnonneg( ___ )` additionally returns a value `exitflag` that describes the exit condition of `lsqnonneg`, and a structure `output` with information about the optimization process.

`[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg( ___ )` additionally returns the Lagrange multiplier vector `lambda`.

Examples

Nonnegative Linear Least Squares

Compute a nonnegative solution to a linear least-squares problem, and compare the result to the solution of an unconstrained problem.

Prepare a `C` matrix and `d` vector for the problem min$||Cx - d||$.

```
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];
```

Compute the constrained and unconstrained solutions.

```
x = lsqnonneg(C,d)
```

x = *2×1*

```
         0
    0.6929
```

```
xunc = C\d
```

xunc = *2×1*

```
   -2.5627
    3.1108
```

All entries in x are nonnegative, but some entries in xunc are negative.

Compute the norms of the residuals for the two solutions.

```
constrained_norm = norm(C*x - d)
```

constrained_norm = 0.9118

```
unconstrained_norm = norm(C*xunc - d)
```

unconstrained_norm = 0.6674

The unconstrained solution has a smaller residual norm because constraints can only increase a residual norm.

Nonnegative Least Squares with Nondefault Options

Set the Display option to 'final' to see output when lsqnonneg finishes.

Create the options.

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

Prepare a C matrix and d vector for the problem $\min||Cx - d||$.

```
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];
```

Call `lsqnonneg` with the options structure.

```
x = lsqnonneg(C,d,options);
```

```
Optimization terminated.
```

Obtain Residuals from Nonnegative Least Squares

Call `lsqnonneg` with outputs to obtain the solution, residual norm, and residual vector.

Prepare a C matrix and d vector for the problem $\min ||Cx - d||$.

```
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];
```

Obtain the solution and residual information.

```
 [x,resnorm,residual] = lsqnonneg(C,d)
```

```
x = 2×1

          0
     0.6929
```

```
resnorm = 0.8315
```

```
residual = 4×1

     0.6599
    -0.3119
    -0.3580
     0.4130
```

Verify that the returned residual norm is the square of the norm of the returned residual vector.

```
 norm(residual)^2
```

```
ans = 0.8315
```

Inspect the Result of Nonnegative Least Squares

Request all output arguments to examine the solution and solution process after `lsqnonneg` finishes.

Prepare a `C` matrix and `d` vector for the problem $\min||Cx - d||$.

```
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];
```

Solve the problem, requesting all output arguments.

```
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(C,d)

x = 2×1

         0
    0.6929


resnorm = 0.8315

residual = 4×1

    0.6599
   -0.3119
   -0.3580
    0.4130


exitflag = 1

output = struct with fields:
    iterations: 1
     algorithm: 'active-set'
       message: 'Optimization terminated.'


lambda = 2×1

   -0.1506
   -0.0000
```

`exitflag` is 1, indicating a correct solution.

`x(1) = 0`, and the corresponding `lambda(1) ≠ 0`, showing the correct duality. Similarly, `x(2) > 0`, and the corresponding `lambda(2) = 0`.

Input Arguments

C — Linear multiplier
real matrix

Linear multiplier, specified as a real matrix. Represents the variable *C* in the problem

$$\min_{x}\|Cx - d\|_2^2.$$

For compatibility, the number of rows of C must equal the length of d.

Example: `C = [1,2;3,-1;-4,4]`

Data Types: `double`

d — Additive term
real vector

Additive term, specified as a real vector. Represents the variable *d* in the problem

$$\min_{x}\|Cx - d\|_2^2.$$

For compatibility, the length of d must equal the number of rows of C.

Example: `d = [1;-6;5]`

Data Types: `double`

options — Optimization options
structure such as `optimset` returns

Optimization options, specified as a structure such as `optimset` returns. You can use `optimset` to set or change the values of these fields in the options structure. See "Optimization Options Reference" on page 15-6 for detailed information.

`Display`	Level of display:
	• `'notify'` (default) displays output only if the function does not converge.
	• `'off'` or `'none'` displays no output.
	• `'final'` displays just the final output.
`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 2-68.

Example: `options = optimset('Display','final')`

Data Types: `struct`

problem — Problem structure
structure

Problem structure, specified as a structure with the following fields.

Field Name	Entry
C	Real matrix
d	Real vector
solver	'lsqnonneg'
options	Options structure such as returned by optimset

Data Types: struct

Output Arguments

x — Solution
real vector

Solution, returned as a real vector. The length of x is the same as the length of d.

resnorm — Squared residual norm
nonnegative scalar

Squared residual norm, returned as a nonnegative scalar. Equal to norm(C*x-d)^2.

residual — Residual
real vector

Residual, returned as a real vector. The residual is d - C*x.

exitflag — Reason lsqnonneg stopped
integer

Reason lsqnonneg stopped, returned as an integer.

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

output — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

iterations	Number of iterations taken
algorithm	'active-set'
message	Exit message

lambda — Lagrange multipliers
real vector

Lagrange multipliers, returned as a real vector. The entries satisfy the complementarity condition x'*lambda = 0. This means lambda(i) < 0 when x(i) is approximately 0, and lambda(i) is approximately 0 when x(i) > 0.

Tips

- For problems where d has length over 20, `lsqlin` might be faster than `lsqnonneg`. When d has length under 20, `lsqnonneg` is generally more efficient.

  To convert between the solvers when `C` has more rows than columns (meaning 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));
  ```

  The only difference is that the corresponding Lagrange multipliers have opposite signs: `lambda = -lambda_lsqlin.ineqlin`.

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` to swap it out of the basis in exchange for another possible candidate. This continues until `lambda ≤ 0`.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for `lsqnonneg`.

References

[1] Lawson, C. L. and R. J. Hanson. *Solving Least-Squares Problems*. Upper Saddle River, NJ: Prentice Hall. 1974. Chapter 23, p. 161.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation:

- You must enable support for variable-size arrays.
- The exit message in the output structure is not translated.

See Also
Optimize | `lsqlin` | `mldivide` | `optimset`

Introduced before R2006a

mapSolution

Package: `optim.problemdef`

Create optimization solution structure from solver outputs

Syntax

```
[sol,fval,exitflag,output,lambda] = mapSolution(prob,xin,fin,eflag,outpt,
lambdain,solver)
```

Description

`[sol,fval,exitflag,output,lambda] = mapSolution(prob,xin,fin,eflag,outpt,
lambdain,solver)` formats an optimization solution in the form that `solve` returns.

Note You can request any subset of the output arguments. If you do, you can include only those input arguments that are required for the output arguments that you request. For example,

```
[sol,fval] = mapSolution(prob,xin,fin)
% or
[sol,~,~,~,lambda] = mapSolution(prob,xin,[],[],[],lambdain,solver)
```

If you request an `output` structure, you must provide a `solver` input.

Examples

Transform Solution

Specify a linear programming problem in the problem framework.

```
x = optimvar('x');
y = optimvar('y');
prob = optimproblem;
prob.Objective = -x - y/3;
prob.Constraints.cons1 = x + y <= 2;
prob.Constraints.cons2 = x + y/4 <= 1;
prob.Constraints.cons3 = x - y <= 2;
prob.Constraints.cons4 = x/4 + y >= -1;
prob.Constraints.cons5 = x + y >= 1;
prob.Constraints.cons6 = -x + y <= 2;
```

Convert the problem to a structure to enable solution by a different solver than `solve`.

```
problem = prob2struct(prob);
```

Solve the problem using the `linprog` solver.

```
options = optimoptions('linprog','Algorithm','dual-simplex');
[xin,fin,eflag,outpt,lambdain] = linprog(problem);
```

```
Optimal solution found.
```

Transform the solution to the form that `solve` returns.

```
[sol,fval,exitflag,output,lambda] = mapSolution(prob,xin,fin,eflag,outpt,lambdain,'linprog')
```

```
sol = struct with fields:
    x: 0.6667
    y: 1.3333


fval = -1.1111

exitflag =
    OptimalSolution


output = struct with fields:
          iterations: 3
       constrviolation: 0
               message: 'Optimal solution found.'
             algorithm: 'dual-simplex'
         firstorderopt: 0
                solver: 'linprog'


lambda = struct with fields:
       Variables: [1x1 struct]
     Constraints: [1x1 struct]
```

Input Arguments

`prob` — Optimization problem
`OptimizationProblem` object

Optimization problem, specified as an `OptimizationProblem` object. Typically, the other input variables such as `xin` come from a solution of the problem as converted to matrix form using `prob2struct`.

`xin` — Optimization solution
real vector

Optimization solution, specified as a real vector. The length of the vector is the sum of the number of elements in the problem variables.

Data Types: `double`

`fin` — Objective function value at solution
real scalar

Objective function value at solution, specified as a real scalar.

If there is an additive constant for the objective function, include it in `fin`. In other words, if the objective function is `a + f'*x`, then include the value `a`, not just `f'*x`, in `fin`.

Data Types: `double`

eflag — Exit flag
integer

Exit flag, specified as an integer.

Data Types: `double`

outpt — Output structure
structure

Output structure, specified as a structure. This structure is copied to the `output` variable, and the `solver` input is appended as the `solver` field of the resulting structure.

Data Types: `struct`

lambdain — Lagrange multipliers
structure

Lagrange multipliers, specified as a structure. For details of the fields of this structure, see "lambda" on page 16-0 .

Data Types: `struct`

solver — Solver name
`'fmincon'` | `'fminunc'` | `'intlinprog'` | `'linprog'` | `'lsqlin'` | `'lsqnonneg'` | `'quadprog'`

Solver name, specified as `'fmincon'`, `'fminunc'`, `'intlinprog'`, `'linprog'`, `'lsqlin'`, `'lsqnonneg'`, or `'quadprog'`.

Note If you request an `output` structure, you must provide a `solver` input.

Data Types: `char` | `string`

Output Arguments

sol — Solution
structure

Solution, returned as a structure. The fields of the structure are the names of the optimization variables. See `optimvar`.

fval — Objective function value at the solution
real number

Objective function value at the solution, returned as a real number.

exitflag — Reason solver stopped
enumeration variable

Reason the solver stopped, returned as an enumeration variable. You can convert `exitflag` to its numeric equivalent using `double(exitflag)`, and to its string equivalent using `string(exitflag)`.

This table describes the exit flags for the `intlinprog` solver.

Exit Flag for `intlinprog`	Numeric Equivalent	Meaning
`OptimalWithPoorFeasibility`	3	The solution is feasible with respect to the relative `ConstraintTolerance` tolerance, but is not feasible with respect to the absolute tolerance.
`IntegerFeasible`	2	`intlinprog` stopped prematurely, and found an integer feasible point.
`OptimalSolution`	1	The solver converged to a solution `x`.
`SolverLimitExceeded`	0	`intlinprog` exceeds one of the following tolerances: • `LPMaxIterations` • `MaxNodes` • `MaxTime` • `RootLPMaxIterations` See "Tolerances and Stopping Criteria" on page 2-68. `solve` also returns this exit flag when it runs out of memory at the root node.
`OutputFcnStop`	-1	`intlinprog` stopped by an output function or plot function.
`NoFeasiblePointFound`	-2	No feasible point found.
`Unbounded`	-3	The problem is unbounded.
`FeasibilityLost`	-9	Solver lost feasibility.

Exitflags 3 and -9 relate to solutions that have large infeasibilities. These usually arise from linear constraint matrices that have large condition number, or problems that have large solution components. To correct these issues, try to scale the coefficient matrices, eliminate redundant linear constraints, or give tighter bounds on the variables.

This table describes the exit flags for the `linprog` solver.

Exit Flag for `linprog`	Numeric Equivalent	Meaning
`OptimalWithPoorFeasibility`	3	The solution is feasible with respect to the relative `ConstraintTolerance` tolerance, but is not feasible with respect to the absolute tolerance.
`OptimalSolution`	1	The solver converged to a solution `x`.
`SolverLimitExceeded`	0	The number of iterations exceeds `options.MaxIterations`.
`NoFeasiblePointFound`	-2	No feasible point found.
`Unbounded`	-3	The problem is unbounded.
`FoundNaN`	-4	`NaN` value encountered during execution of the algorithm.

Exit Flag for `linprog`	Numeric Equivalent	Meaning
`PrimalDualInfeasible`	-5	Both primal and dual problems are infeasible.
`DirectionTooSmall`	-7	The search direction is too small. No further progress can be made.
`FeasibilityLost`	-9	Solver lost feasibility.

Exitflags 3 and -9 relate to solutions that have large infeasibilities. These usually arise from linear constraint matrices that have large condition number, or problems that have large solution components. To correct these issues, try to scale the coefficient matrices, eliminate redundant linear constraints, or give tighter bounds on the variables.

This table describes the exit flags for the `lsqlin` solver.

Exit Flag for `lsqlin`	Numeric Equivalent	Meaning
`FunctionChangeBelowToler ance`	3	Change in the residual is smaller than the specified tolerance `options.FunctionTolerance`. (`trust-region-reflective` algorithm)
`StepSizeBelowTolerance`	2	Step size smaller than `options.StepTolerance`, constraints satisfied. (`interior-point` algorithm)
`OptimalSolution`	1	The solver converged to a solution `x`.
`SolverLimitExceeded`	0	The number of iterations exceeds `options.MaxIterations`.
`NoFeasiblePointFound`	-2	For optimization problems, the problem is infeasible. Or, for the `interior-point` algorithm, step size smaller than `options.StepTolerance`, but constraints are not satisfied. For equation problems, no solution found.
`IllConditioned`	-4	Ill-conditioning prevents further optimization.
`NoDescentDirectionFound`	-8	The search direction is too small. No further progress can be made. (`interior-point` algorithm)

This table describes the exit flags for the `quadprog` solver.

Exit Flag for quadprog	Numeric Equivalent	Meaning
`LocalMinimumFound`	4	Local minimum found; minimum is not unique.
`FunctionChangeBelowToler ance`	3	Change in the objective function value is smaller than the specified tolerance `options.FunctionTolerance`. (`trust-region-reflective` algorithm)

Exit Flag for quadprog	Numeric Equivalent	Meaning
StepSizeBelowTolerance	2	Step size smaller than `options.StepTolerance`, constraints satisfied. (`interior-point-convex` algorithm)
OptimalSolution	1	The solver converged to a solution x.
SolverLimitExceeded	0	The number of iterations exceeds `options.MaxIterations`.
NoFeasiblePointFound	-2	The problem is infeasible. Or, for the `interior-point` algorithm, step size smaller than `options.StepTolerance`, but constraints are not satisfied.
IllConditioned	-4	Ill-conditioning prevents further optimization.
Nonconvex	-6	Nonconvex problem detected. (`interior-point-convex` algorithm)
NoDescentDirectionFound	-8	Unable to compute a step direction. (`interior-point-convex` algorithm)

This table describes the exit flags for the `lsqcurvefit` or `lsqnonlin` solver.

Exit Flag for lsqnonlin	Numeric Equivalent	Meaning
SearchDirectionTooSmall	4	Magnitude of search direction was smaller than `options.StepTolerance`.
FunctionChangeBelowTolerance	3	Change in the residual was less than `options.FunctionTolerance`.
StepSizeBelowTolerance	2	Step size smaller than `options.StepTolerance`.
OptimalSolution	1	The solver converged to a solution x.
SolverLimitExceeded	0	Number of iterations exceeded `options.MaxIterations` or number of function evaluations exceeded `options.MaxFunctionEvaluations`.
OutputFcnStop	-1	Stopped by an output function or plot function.
NoFeasiblePointFound	-2	For optimization problems, problem is infeasible: the bounds lb and ub are inconsistent. For equation problems, no solution found.

This table describes the exit flags for the `fminunc` solver.

Exit Flag for fminunc	Numeric Equivalent	Meaning
NoDecreaseAlongSearchDirection	5	Predicted decrease in the objective function is less than the options.FunctionTolerance tolerance.
FunctionChangeBelowTolerance	3	Change in the objective function value is less than the options.FunctionTolerance tolerance.
StepSizeBelowTolerance	2	Change in x is smaller than the options.StepTolerance tolerance.
OptimalSolution	1	Magnitude of gradient is smaller than the options.OptimalityTolerance tolerance.
SolverLimitExceeded	0	Number of iterations exceeds options.MaxIterations or number of function evaluations exceeds options.MaxFunctionEvaluations.
OutputFcnStop	-1	Stopped by an output function or plot function.
Unbounded	-3	Objective function at current iteration is below options.ObjectiveLimit.

This table describes the exit flags for the fmincon solver.

Exit Flag for fmincon	Numeric Equivalent	Meaning
NoDecreaseAlongSearchDirection	5	Magnitude of directional derivative in search direction is less than 2*options.OptimalityTolerance and maximum constraint violation is less than options.ConstraintTolerance.
SearchDirectionTooSmall	4	Magnitude of the search direction is less than 2*options.StepTolerance and maximum constraint violation is less than options.ConstraintTolerance.
FunctionChangeBelowTolerance	3	Change in the objective function value is less than options.FunctionTolerance and maximum constraint violation is less than options.ConstraintTolerance.
StepSizeBelowTolerance	2	Change in x is less than options.StepTolerance and maximum constraint violation is less than options.ConstraintTolerance.
OptimalSolution	1	First-order optimality measure is less than options.OptimalityTolerance, and maximum constraint violation is less than options.ConstraintTolerance.

Exit Flag for fmincon	Numeric Equivalent	Meaning
SolverLimitExceeded	0	Number of iterations exceeds `options.MaxIterations` or number of function evaluations exceeds `options.MaxFunctionEvaluations`.
OutputFcnStop	-1	Stopped by an output function or plot function.
NoFeasiblePointFound	-2	No feasible point found.
Unbounded	-3	Objective function at current iteration is below `options.ObjectiveLimit` and maximum constraint violation is less than `options.ConstraintTolerance`.

This table describes the exit flags for the `fsolve` solver.

Exit Flag for fsolve	Numeric Equivalent	Meaning
SearchDirectionTooSmall	4	Magnitude of the search direction is less than `options.StepTolerance`, equation solved.
FunctionChangeBelowTolerance	3	Change in the objective function value is less than `options.FunctionTolerance`, equation solved.
StepSizeBelowTolerance	2	Change in x is less than `options.StepTolerance`, equation solved.
OptimalSolution	1	First-order optimality measure is less than `options.OptimalityTolerance`, equation solved.
SolverLimitExceeded	0	Number of iterations exceeds `options.MaxIterations` or number of function evaluations exceeds `options.MaxFunctionEvaluations`.
OutputFcnStop	-1	Stopped by an output function or plot function.
NoFeasiblePointFound	-2	Converged to a point that is not a root.
TrustRegionRadiusTooSmall	-3	Equation not solved. Trust region radius became too small (`trust-region-dogleg` algorithm).

This table describes the exit flags for the `fzero` solver.

Exit Flag for fzero	Numeric Equivalent	Meaning
OptimalSolution	1	Equation solved.
OutputFcnStop	-1	Stopped by an output function or plot function.
FoundNaNInfOrComplex	-4	`NaN`, `Inf`, or complex value encountered during search for an interval containing a sign change.

Exit Flag for `fzero`	Numeric Equivalent	Meaning
`SingularPoint`	-5	Might have converged to a singular point.
`CannotDetectSignChange`	-6	Did not find two points with opposite signs of function value.

`output` — Information about optimization process
structure

Information about the optimization process, returned as a structure. The output structure contains the fields in the relevant underlying solver output field, depending on which solver `solve` called:

- `'fmincon'` output
- `'fminunc'` output
- `'fsolve'` output
- `'fzero'` output
- `'intlinprog'` output
- `'linprog'` output
- `'lsqcurvefit'` or `'lsqnonlin'` output
- `'lsqlin'` output
- `'lsqnonneg'` output
- `'quadprog'` output

`solve` includes the additional field `Solver` in the `output` structure to identify the solver used, such as `'intlinprog'`.

`lambda` — Lagrange multipliers at the solution
structure

Lagrange multipliers at the solution, returned as a structure.

Note `solve` does not return `lambda` for equation-solving problems.

For the `intlinprog` and `fminunc` solvers, `lambda` is empty, `[]`. For the other solvers, `lambda` has these fields:

- `Variables` – Contains fields for each problem variable. Each problem variable name is a structure with two fields:

  - `Lower` – Lagrange multipliers associated with the variable `LowerBound` property, returned as an array of the same size as the variable. Nonzero entries mean that the solution is at the lower bound. These multipliers are in the structure `lambda.Variables.`*`variablename`*`.Lower`.

  - `Upper` – Lagrange multipliers associated with the variable `UpperBound` property, returned as an array of the same size as the variable. Nonzero entries mean that the solution is at the upper bound. These multipliers are in the structure `lambda.Variables.`*`variablename`*`.Upper`.

- Constraints – Contains a field for each problem constraint. Each problem constraint is in a structure whose name is the constraint name, and whose value is a numeric array of the same size as the constraint. Nonzero entries mean that the constraint is active at the solution. These multipliers are in the structure lambda.Constraints.*constraintname*.

Note Elements of a constraint array all have the same comparison (<=, ==, or >=) and are all of the same type (linear, quadratic, or nonlinear).

See Also

OptimizationProblem | prob2struct | solve | varindex

Topics
"Examine Optimization Solution" on page 10-25
"Supply Derivatives in Problem-Based Workflow" on page 7-26

Introduced in R2017b

mpsread

Read MPS file for LP and MILP optimization data

Syntax

```
problem = mpsread(mpsfile)
problem = mpsread(mpsfile,'ReturnNames',true)
```

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.

`problem = mpsread(mpsfile,'ReturnNames',true)` augments the returned `problem` structure with `variableNames` and `constraintNames` fields containing the names of the variables and constraints in `mpsfile`.

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/WebData/instances/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','PlotFcn',@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.AbsoluteGapTolerance = 0 (the default value). The intcon variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the default value).
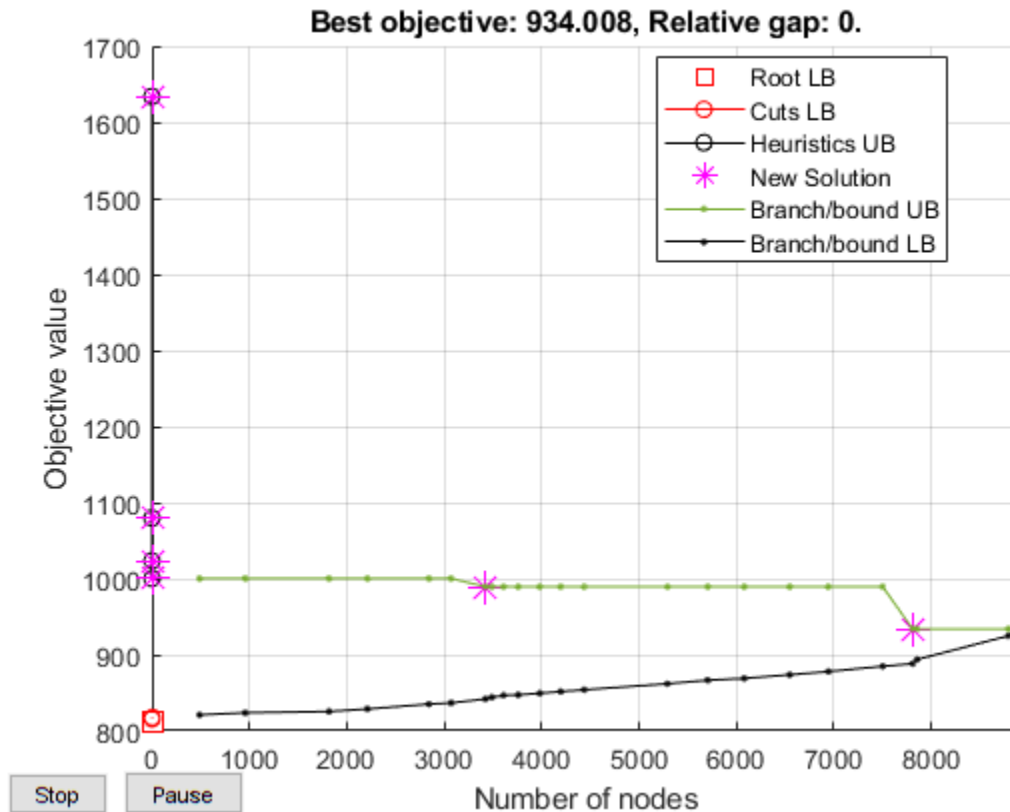
Obtain Variable and Constraint Names

Load an mps file and obtain its variable and constraint names.

Load the eil33-2.mps file from a public repository. View the returned problem structure.

```
gunzip('http://miplib.zib.de/WebData/instances/eil33-2.mps.gz')
problem = mpsread('eil33-2.mps','ReturnNames',true)

problem =

  struct with fields:

            f: [4516×1 double]
        Aineq: [0×4516 double]
        bineq: [0×1 double]
          Aeq: [32×4516 double]
          beq: [32×1 double]
```

```
            lb: [4516×1 double]
            ub: [4516×1 double]
         intcon: [4516×1 double]
         solver: 'intlinprog'
        options: [1×1 optim.options.Intlinprog]
   variableNames: [4516×1 string]
  constraintNames: [1×1 struct]
```

View the first few names of each type.

```
problem.variableNames(1:4)
```

```
ans =

  4×1 string array

    "x1"
    "x2"
    "x3"
    "x4"
```

```
problem.constraintNames.eqlin(1:4)
```

```
ans =

  4×1 string array

    "c1"
    "c2"
    "c3"
    "c4"
```

There are no inequality constraints in the problem.

```
problem.constraintNames.ineqlin
```

```
ans =

  0×1 empty string array
```

Input Arguments

mpsfile — Path to MPS file
character vector | string scalar

Path to MPS file, specified as a character vector or string scalar. `mpsfile` should be a file in the MPS format.

Note

- `mpsread` does not support semicontinuous constraints or SOS constraints.
- `mpsread` supports "fixed format" files.
- `mpsread` does not support extensions such as `objsense` and `objname`.
- `mpsread` silently ignores variables in the `BOUNDS` section that do not previously appear in the `COLUMNS` section of the MPS file.

Example: "documents/optimization/milpproblem.mps"

Data Types: `char` | `string`

ReturnNames — Name-value pair indicating to return variable and constraint names from the MPS file
`false` (default) | `true`

Name-value pair indicating to return variable and constraint names from the MPS file, with the value specified as logical. `false` indicates not to return the names. `true` causes `mpsread` to return two extra fields in the `problem` output structure:

- `problem.variableNames` — String array of variable names
- `problem.constraintNames` — Structure of constraint names:
  - `problem.constraintNames.eqlin` String array of linear equality constraint names
  - `problem.constraintNames.ineqlin` String array of linear inequality constraint names

The `problem` structure inequality constraints `problem.Aineq` and `problem.bineq` have the same order as the names in `problem.constraintNames.ineqlin`. Similarly, the constraints `problem.Aeq` and `problem.beq` have the same order as the names in `problem.constraintNames.eqlin`. The `problem.variableNames` order is the same as the order of the solution variables x after running `linprog` or `intlinprog` on the `problem` structure.

Example: `mpsread('filename','ReturnNames',true)`

Data Types: `logical`

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 ≤ 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'` (if `intcon` is nonempty), or `'linprog'` (if `intcon` is empty)
`options`	Default options, as returned by the command `optimoptions(solver)`

variableNames	String array containing variable names from the MPS file. This field appears only if `ReturnNames` is `true`.
constraintNames	Structure containing constraint names from the MPS file. For a description, see `ReturnNames`. This field appears only if `ReturnNames` is `true`.

`mpsread` returns `problem.Aineq` and `problem.Aeq` as sparse matrices.

See Also
`intlinprog` | `linprog`

Topics
"Linear Programming and Mixed-Integer Linear Programming"

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');
```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Input parameter names must be constant.

See Also
`optimset`

Introduced before R2006a

optimconstr

Create empty optimization constraint array

Syntax

```
constr = optimconstr(N)
constr = optimconstr(cstr)
constr = optimconstr(cstr1,N2,...,cstrk)
constr = optimconstr({cstr1,cstr2,...,cstrk})
constr = optimconstr([N1,N2,...,Nk])
```

Description

Use `optimconstr` to initialize a set of constraint expressions.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2.

`constr = optimconstr(N)` creates an N-by-1 array of empty optimization constraints. Use `constr` to initialize a loop that creates constraint expressions.

`constr = optimconstr(cstr)` creates an array of empty optimization constraints that are indexed by `cstr`, a cell array of character vectors or string vectors.

If `cstr` is 1-by-*ncstr*, where *ncstr* is the number of elements of `cstr`, then `constr` is also 1-by-*ncstr*. Otherwise, `constr` is *ncstr*-by-1.

`constr = optimconstr(cstr1,N2,...,cstrk)` or `constr = optimconstr({cstr1, cstr2,...,cstrk})` or `constr = optimconstr([N1,N2,...,Nk])`, for any combination of `cstr` and `N` arguments, creates an *ncstr1*-by-N2-by-...-by-*ncstrk* array of empty optimization constraints, where *ncstr* is the number of elements in `cstr`.

Examples

Create Constraints in Loop

Create constraints for an inventory model. The stock of goods at the start of each period is equal to the stock at the end of the previous period. During each period, the stock increases by `buy` and decreases by `sell`. The variable `stock` is the stock at the end of the period.

```
N = 12;
stock = optimvar('stock',N,1,'Type','integer','LowerBound',0);
buy = optimvar('buy',N,1,'Type','integer','LowerBound',0);
sell = optimvar('sell',N,1,'Type','integer','LowerBound',0);
initialstock = 100;

stockbalance = optimconstr(N,1);

for t = 1:N
```

```
    if t == 1
        enterstock = initialstock;
    else
        enterstock = stock(t-1);
    end
    stockbalance(t) = stock(t) == enterstock + buy(t) - sell(t);
end

show(stockbalance)
(1, 1)
  -buy(1) + sell(1) + stock(1) == 100
(2, 1)
  -buy(2) + sell(2) - stock(1) + stock(2) == 0
(3, 1)
  -buy(3) + sell(3) - stock(2) + stock(3) == 0
(4, 1)
  -buy(4) + sell(4) - stock(3) + stock(4) == 0
(5, 1)
  -buy(5) + sell(5) - stock(4) + stock(5) == 0
(6, 1)
  -buy(6) + sell(6) - stock(5) + stock(6) == 0
(7, 1)
  -buy(7) + sell(7) - stock(6) + stock(7) == 0
(8, 1)
  -buy(8) + sell(8) - stock(7) + stock(8) == 0
(9, 1)
  -buy(9) + sell(9) - stock(8) + stock(9) == 0
(10, 1)
  -buy(10) + sell(10) - stock(9) + stock(10) == 0
(11, 1)
  -buy(11) + sell(11) - stock(10) + stock(11) == 0
(12, 1)
  -buy(12) + sell(12) - stock(11) + stock(12) == 0
```

Include the constraints in a problem.

```
prob = optimproblem;
prob.Constraints.stockbalance = stockbalance;
```

Instead of using a loop, you can create the same constraints by using matrix operations on the variables.

```
tt = ones(N-1,1);
d = diag(tt,-1); % shift index by -1
stockbalance2 = stock == d*stock + buy - sell;
stockbalance2(1) = stock(1) == initialstock + buy(1) - sell(1);
```

See that the new constraints are the same as the constraints in `stockbalance`:

```
show(stockbalance2)
```

```
(1, 1)

  -buy(1) + sell(1) + stock(1) == 100

(2, 1)

  -buy(2) + sell(2) - stock(1) + stock(2) == 0

(3, 1)

  -buy(3) + sell(3) - stock(2) + stock(3) == 0

(4, 1)

  -buy(4) + sell(4) - stock(3) + stock(4) == 0

(5, 1)

  -buy(5) + sell(5) - stock(4) + stock(5) == 0

(6, 1)

  -buy(6) + sell(6) - stock(5) + stock(6) == 0

(7, 1)

  -buy(7) + sell(7) - stock(6) + stock(7) == 0

(8, 1)

  -buy(8) + sell(8) - stock(7) + stock(8) == 0

(9, 1)

  -buy(9) + sell(9) - stock(8) + stock(9) == 0

(10, 1)

  -buy(10) + sell(10) - stock(9) + stock(10) == 0

(11, 1)
```

```
  -buy(11) + sell(11) - stock(10) + stock(11) == 0

(12, 1)

  -buy(12) + sell(12) - stock(11) + stock(12) == 0
```

Creating constraints in a loop can be more time-consuming than creating constraints by matrix operations. However, you are less likely to create an erroneous constraint by using loops.

Create Indexed Constraints in Loop

Create indexed constraints and variables to represent the calories consumed in a diet. Each meal has a different calorie limit.

```
meals = ["breakfast","lunch","dinner"];
constr = optimconstr(meals);
foods = ["cereal","oatmeal","yogurt","peanut butter sandwich","pizza","hamburger",...
    "salad","steak","casserole","ice cream"];
diet = optimvar('diet',foods,meals,'LowerBound',0);
calories = [200,175,150,450,350,800,150,650,350,300]';
for i = 1:3
    constr(i) = diet(:,i)'*calories <= 250*i;
end
```

Check the constraint for `dinner`.

```
show(constr("dinner"))
```

```
  200*diet('cereal', 'dinner') + 175*diet('oatmeal', 'dinner')
+ 150*diet('yogurt', 'dinner')
+ 450*diet('peanut butter sandwich', 'dinner') + 350*diet('pizza', 'dinner')
+ 800*diet('hamburger', 'dinner') + 150*diet('salad', 'dinner')
+ 650*diet('steak', 'dinner') + 350*diet('casserole', 'dinner')
+ 300*diet('ice cream', 'dinner') <= 750
```

Input Arguments

N — Size of constraint dimension
positive integer

Size of the constraint dimension, specified as a positive integer.

- The size of `constr = optimconstr(N)` is N-by-1.
- The size of `constr = optimconstr(N1,N2)` is N1-by-N2.
- The size of `constr = optimconstr(N1,N2,...,Nk)` is N1-by-N2-by-...-by-Nk.

Example: 5

Data Types: `double`

cstr — Names for indexing
cell array of character vectors | string vector

Names for indexing, specified as a cell array of character vectors or a string vector.

Example: {'red','orange','green','blue'}

Example: ["red";"orange";"green";"blue"]

Data Types: string | cell

Output Arguments

constr — Constraints
empty OptimizationConstraint array

Constraints, returned as an empty OptimizationConstraint array. Use constr to initialize a loop that creates constraint expressions.

For example:

```
x = optimvar('x',8);
constr = optimconstr(4);
for k = 1:4
    constr(k) = 5*k*(x(2*k) - x(2*k-1)) <= 10 - 2*k;
end
```

Limitations

- Each constraint expression in a problem must use the same comparison. For example, the following code leads to an error, because cons1 uses the <= comparison, cons2 uses the >= comparison, and cons1 and cons2 are in the same expression.

```
prob = optimproblem;
x = optimvar('x',2,'LowerBound',0);
cons1 = x(1) + x(2) <= 10;
cons2 = 3*x(1) + 4*x(2) >= 2;
prob.Constraints = [cons1;cons2]; % This line throws an error
```

You can avoid this error by using separate expressions for the constraints.

```
prob.Constraints.cons1 = cons1;
prob.Constraints.cons2 = cons2;
```

Tips

- It is generally more efficient to create constraints by vectorized expressions rather than loops. See "Create Efficient Optimization Problems" on page 10-28.
- You can use optimineq instead of optimconstr to create inequality expressions. Similarly, you can use optimeq instead of optimconstr to create equality expressions.

See Also

OptimizationConstraint | OptimizationExpression | OptimizationProblem | OptimizationVariable | optimeq | optimexpr | optimineq

Topics
"Problem-Based Optimization Setup"

"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

optimeq

Create empty optimization equality array

Syntax

```
eq = optimeq(N)
eq = optimeq(cstr)
eq = optimeq(cstr1,N2,...,cstrk)
eq = optimeq({cstr1,cstr2,...,cstrk})
eq = optimeq([N1,N2,...,Nk])
```

Description

Use `optimeq` to initialize a set of equality expressions.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`eq = optimeq(N)` creates an N-by-1 array of empty optimization equalities. Use `eq` to initialize a loop that creates equalities. Use the resulting equalities as constraints in an optimization problem or as equations in an equation problem.

`eq = optimeq(cstr)` creates an array of empty optimization equalities that are indexed by `cstr`, a cell array of character vectors or string vectors .

If `cstr` is 1-by-*ncstr*, where *ncstr* is the number of elements of `cstr`, then `eq` is also 1-by-*ncstr*. Otherwise, `eq` is *ncstr*-by-1.

`eq = optimeq(cstr1,N2,...,cstrk)` or `eq = optimeq({cstr1,cstr2,...,cstrk})` or `eq = optimeq([N1,N2,...,Nk])`, for any combination of `cstr` and N arguments, creates an *ncstr1*-by-N2-by-...-by-*ncstrk* array of empty optimization equalities, where *ncstr* is the number of elements in `cstr`.

Examples

Create Equality Constraints in Loop

Create equality constraints for an inventory model. The stock of goods at the start of each period is equal to the stock at the end of the previous period. During each period, the stock increases by `buy` and decreases by `sell`. The variable `stock` is the stock at the end of the period.

```
N = 12;
stock = optimvar('stock',N,1,'Type','integer','LowerBound',0);
buy = optimvar('buy',N,1,'Type','integer','LowerBound',0);
sell = optimvar('sell',N,1,'Type','integer','LowerBound',0);
initialstock = 100;
```

```
stockbalance = optimeq(N,1);

for t = 1:N
    if t == 1
        enterstock = initialstock;
    else
        enterstock = stock(t-1);
    end
    stockbalance(t) = stock(t) == enterstock + buy(t) - sell(t);
end

show(stockbalance)
(1, 1)

  -buy(1) + sell(1) + stock(1) == 100

(2, 1)

  -buy(2) + sell(2) - stock(1) + stock(2) == 0

(3, 1)

  -buy(3) + sell(3) - stock(2) + stock(3) == 0

(4, 1)

  -buy(4) + sell(4) - stock(3) + stock(4) == 0

(5, 1)

  -buy(5) + sell(5) - stock(4) + stock(5) == 0

(6, 1)

  -buy(6) + sell(6) - stock(5) + stock(6) == 0

(7, 1)

  -buy(7) + sell(7) - stock(6) + stock(7) == 0

(8, 1)

  -buy(8) + sell(8) - stock(7) + stock(8) == 0

(9, 1)

  -buy(9) + sell(9) - stock(8) + stock(9) == 0

(10, 1)

  -buy(10) + sell(10) - stock(9) + stock(10) == 0

(11, 1)

  -buy(11) + sell(11) - stock(10) + stock(11) == 0

(12, 1)
```

```
-buy(12) + sell(12) - stock(11) + stock(12) == 0
```

Include the constraints in an optimization problem.

```
prob = optimproblem;
prob.Constraints.stockbalance = stockbalance;
```

Instead of using a loop, you can create the same constraints by using matrix operations on the variables.

```
stockbalance2 = optimeq(12, 1);
t = 2:12;
stockbalance2(t) = stock(t) == stock(t-1) + buy(t) - sell(t);
stockbalance2(1) = stock(1) == initialstock + buy(1) - sell(1);
```

Display the new constraints. Note that they are the same as the constraints in `stockbalance`.

```
show(stockbalance2)
```

```
(1, 1)

  -buy(1) + sell(1) + stock(1) == 100

(2, 1)

  -buy(2) + sell(2) - stock(1) + stock(2) == 0

(3, 1)

  -buy(3) + sell(3) - stock(2) + stock(3) == 0

(4, 1)

  -buy(4) + sell(4) - stock(3) + stock(4) == 0

(5, 1)

  -buy(5) + sell(5) - stock(4) + stock(5) == 0

(6, 1)

  -buy(6) + sell(6) - stock(5) + stock(6) == 0

(7, 1)

  -buy(7) + sell(7) - stock(6) + stock(7) == 0

(8, 1)

  -buy(8) + sell(8) - stock(7) + stock(8) == 0

(9, 1)

  -buy(9) + sell(9) - stock(8) + stock(9) == 0

(10, 1)
```

```
  -buy(10) + sell(10) - stock(9) + stock(10) == 0

(11, 1)

  -buy(11) + sell(11) - stock(10) + stock(11) == 0

(12, 1)

  -buy(12) + sell(12) - stock(11) + stock(12) == 0
```

Creating constraints in a loop can be more time consuming than creating constraints by using matrix operations.

Create Indexed Equalities in Loop

Create indexed equalities for a problem that involves shipping goods between airports. First, create indices representing airports.

```
airports = ["LAX" "JFK" "ORD"];
```

Create indices representing goods to be shipped from one airport to another.

```
goods = ["Electronics" "Foodstuffs" "Clothing" "Raw Materials"];
```

Create an array giving the weight of each unit of the goods.

```
weights = [1 20 5 100];
```

Create a variable array representing quantities of goods to be shipped from each airport to each other airport. `quantities(airport1,airport2,goods)` represents the quantity of `goods` being shipped from `airport1` to `airport2`.

```
quantities = optimvar('quantities',airports,airports,goods,'LowerBound',0);
```

Create an equality constraint that the sum of the weights of goods being shipped from each airport is equal to the sum of the weights of goods being shipped to the airport.

```
eq = optimeq(airports);
outweight = optimexpr(size(eq));
inweight = optimexpr(size(eq));
for i = 1:length(airports)
    temp = optimexpr;
    temp2 = optimexpr;
    for j = 1:length(airports)
        for k = 1:length(goods)
            temp = temp + quantities(i,j,k)*weights(k);
            temp2 = temp2 + quantities(j,i,k)*weights(k);
        end
    end
    outweight(i) = temp;
    inweight(i) = temp2;
    eq(i) = outweight(i) == inweight(i);
end
```

Examine the equalities.

```
show(eq)

(1, 'LAX')

  -quantities('JFK', 'LAX', 'Electronics')
- quantities('ORD', 'LAX', 'Electronics')
+ quantities('LAX', 'JFK', 'Electronics')
+ quantities('LAX', 'ORD', 'Electronics')
- 20*quantities('JFK', 'LAX', 'Foodstuffs')
- 20*quantities('ORD', 'LAX', 'Foodstuffs')
+ 20*quantities('LAX', 'JFK', 'Foodstuffs')
+ 20*quantities('LAX', 'ORD', 'Foodstuffs')
- 5*quantities('JFK', 'LAX', 'Clothing')
- 5*quantities('ORD', 'LAX', 'Clothing')
+ 5*quantities('LAX', 'JFK', 'Clothing')
+ 5*quantities('LAX', 'ORD', 'Clothing')
- 100*quantities('JFK', 'LAX', 'Raw Materials')
- 100*quantities('ORD', 'LAX', 'Raw Materials')
+ 100*quantities('LAX', 'JFK', 'Raw Materials')
+ 100*quantities('LAX', 'ORD', 'Raw Materials') == 0

(1, 'JFK')

  quantities('JFK', 'LAX', 'Electronics')
- quantities('LAX', 'JFK', 'Electronics')
- quantities('ORD', 'JFK', 'Electronics')
+ quantities('JFK', 'ORD', 'Electronics')
+ 20*quantities('JFK', 'LAX', 'Foodstuffs')
- 20*quantities('LAX', 'JFK', 'Foodstuffs')
- 20*quantities('ORD', 'JFK', 'Foodstuffs')
+ 20*quantities('JFK', 'ORD', 'Foodstuffs')
+ 5*quantities('JFK', 'LAX', 'Clothing')
- 5*quantities('LAX', 'JFK', 'Clothing')
- 5*quantities('ORD', 'JFK', 'Clothing')
+ 5*quantities('JFK', 'ORD', 'Clothing')
+ 100*quantities('JFK', 'LAX', 'Raw Materials')
- 100*quantities('LAX', 'JFK', 'Raw Materials')
- 100*quantities('ORD', 'JFK', 'Raw Materials')
+ 100*quantities('JFK', 'ORD', 'Raw Materials') == 0

(1, 'ORD')

  quantities('ORD', 'LAX', 'Electronics')
+ quantities('ORD', 'JFK', 'Electronics')
- quantities('LAX', 'ORD', 'Electronics')
- quantities('JFK', 'ORD', 'Electronics')
+ 20*quantities('ORD', 'LAX', 'Foodstuffs')
+ 20*quantities('ORD', 'JFK', 'Foodstuffs')
- 20*quantities('LAX', 'ORD', 'Foodstuffs')
- 20*quantities('JFK', 'ORD', 'Foodstuffs')
+ 5*quantities('ORD', 'LAX', 'Clothing')
+ 5*quantities('ORD', 'JFK', 'Clothing')
- 5*quantities('LAX', 'ORD', 'Clothing')
- 5*quantities('JFK', 'ORD', 'Clothing')
+ 100*quantities('ORD', 'LAX', 'Raw Materials')
+ 100*quantities('ORD', 'JFK', 'Raw Materials')
- 100*quantities('LAX', 'ORD', 'Raw Materials')
- 100*quantities('JFK', 'ORD', 'Raw Materials') == 0
```

To avoid the nested `for` loops, express the equalities using standard MATLAB® operators. Create the array of departing quantities by summing over the arrival airport indices. Squeeze the result to remove the singleton dimension.

```
departing = squeeze(sum(quantities,2));
```

Calculate the weights of the departing quantities.

```
departweights = departing * weights';
```

Similarly, calculate the weights of arriving quantities.

```
arriving = squeeze(sum(quantities,1));
arriveweights = arriving*weights';
```

Create the constraints that the departing weights equal the arriving weights.

```
eq2 = departweights == arriveweights;
```

Include the appropriate index names for the equalities by setting the `IndexNames` property.

```
eq2.IndexNames = {airports,{}};
```

Display the new equalities. Note that they match the previous equalities, but are transposed vectors.

```
show(eq2)
```

```
('LAX', 1)

   -quantities('JFK', 'LAX', 'Electronics')
 - quantities('ORD', 'LAX', 'Electronics')
 + quantities('LAX', 'JFK', 'Electronics')
 + quantities('LAX', 'ORD', 'Electronics')
 - 20*quantities('JFK', 'LAX', 'Foodstuffs')
 - 20*quantities('ORD', 'LAX', 'Foodstuffs')
 + 20*quantities('LAX', 'JFK', 'Foodstuffs')
 + 20*quantities('LAX', 'ORD', 'Foodstuffs')
 - 5*quantities('JFK', 'LAX', 'Clothing')
 - 5*quantities('ORD', 'LAX', 'Clothing')
 + 5*quantities('LAX', 'JFK', 'Clothing')
 + 5*quantities('LAX', 'ORD', 'Clothing')
 - 100*quantities('JFK', 'LAX', 'Raw Materials')
 - 100*quantities('ORD', 'LAX', 'Raw Materials')
 + 100*quantities('LAX', 'JFK', 'Raw Materials')
 + 100*quantities('LAX', 'ORD', 'Raw Materials') == 0

('JFK', 1)

   quantities('JFK', 'LAX', 'Electronics')
 - quantities('LAX', 'JFK', 'Electronics')
 - quantities('ORD', 'JFK', 'Electronics')
 + quantities('JFK', 'ORD', 'Electronics')
 + 20*quantities('JFK', 'LAX', 'Foodstuffs')
 - 20*quantities('LAX', 'JFK', 'Foodstuffs')
 - 20*quantities('ORD', 'JFK', 'Foodstuffs')
 + 20*quantities('JFK', 'ORD', 'Foodstuffs')
 + 5*quantities('JFK', 'LAX', 'Clothing')
 - 5*quantities('LAX', 'JFK', 'Clothing')
 - 5*quantities('ORD', 'JFK', 'Clothing')
```

```
+ 5*quantities('JFK', 'ORD', 'Clothing')
+ 100*quantities('JFK', 'LAX', 'Raw Materials')
- 100*quantities('LAX', 'JFK', 'Raw Materials')
- 100*quantities('ORD', 'JFK', 'Raw Materials')
+ 100*quantities('JFK', 'ORD', 'Raw Materials') == 0

('ORD', 1)

  quantities('ORD', 'LAX', 'Electronics')
+ quantities('ORD', 'JFK', 'Electronics')
- quantities('LAX', 'ORD', 'Electronics')
- quantities('JFK', 'ORD', 'Electronics')
+ 20*quantities('ORD', 'LAX', 'Foodstuffs')
+ 20*quantities('ORD', 'JFK', 'Foodstuffs')
- 20*quantities('LAX', 'ORD', 'Foodstuffs')
- 20*quantities('JFK', 'ORD', 'Foodstuffs')
+ 5*quantities('ORD', 'LAX', 'Clothing')
+ 5*quantities('ORD', 'JFK', 'Clothing')
- 5*quantities('LAX', 'ORD', 'Clothing')
- 5*quantities('JFK', 'ORD', 'Clothing')
+ 100*quantities('ORD', 'LAX', 'Raw Materials')
+ 100*quantities('ORD', 'JFK', 'Raw Materials')
- 100*quantities('LAX', 'ORD', 'Raw Materials')
- 100*quantities('JFK', 'ORD', 'Raw Materials') == 0
```

Creating constraints in a loop can be more time consuming than creating constraints by using matrix operations.

Input Arguments

`N` — Size of constraint dimension
positive integer

Size of the constraint dimension, specified as a positive integer.

- The size of `constr = optimeq(N)` is N-by-1.

- The size of `constr = optimeq(N1,N2)` is N1-by-N2.

- The size of `constr = optimeq(N1,N2,...,Nk)` is N1-by-N2-by-...-by-Nk.

Example: 5

Data Types: `double`

`cstr` — Names for indexing
cell array of character vectors | string vector

Names for indexing, specified as a cell array of character vectors or a string vector.

Example: `{'red','orange','green','blue'}`

Example: `["red";"orange";"green";"blue"]`

Data Types: `string` | `cell`

Output Arguments

eq — Equalities
empty `OptimizationEquality` array

Equalities, returned as an empty `OptimizationEquality` array. Use `eq` to initialize a loop that creates equalities.

For example:

```
x = optimvar('x',8);
eq = optimeq(4);
for k = 1:4
    eq(k) = 5*k*(x(2*k) - x(2*k-1)) == 10 - 2*k;
end
```

Tips

*   You can use `optimconstr` instead of `optimeq` to create equality constraints for optimization problems or equations for equation problems.

See Also

`OptimizationEquality` | `optimconstr` | `optimineq`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2019b

optimexpr

Create empty optimization expression array

Syntax

```
expr = optimexpr(n)
expr = optimexpr(cstr)
expr = optimexpr(cstr1,n2,...,cstrk)
expr = optimexpr([n1,n2,...,nk])
expr = optimexpr({cstr1,cstr2,...,cstrk})
```

Description

Use `optimexpr` to initialize a set of optimization expressions.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`expr = optimexpr(n)` creates an empty n-by-1 `OptimizationExpression` array. Use `expr` as the initial value in a loop that creates optimization expressions.

`expr = optimexpr(cstr)` creates an empty `OptimizationExpression` array that can use the vector `cstr` for indexing. The number of elements of `expr` is the same as the length of `cstr`. When `cstr` is a row vector, then `expr` is a row vector. When `cstr` is a column vector, then `expr` is a column vector.

`expr = optimexpr(cstr1,n2,...,cstrk)` or `expr = optimexpr([n1,n2,...,nk])`or `expr = optimexpr({cstr1,cstr2,...,cstrk})`, for any combination of positive integers n_j and names $cstr_j$, creates an empty array of optimization expressions with dimensions equal to the integers n_j or the lengths of the entries of $cstr_j$.

Examples

Create Optimization Expression Array

Create an empty array of three optimization expressions.

```
expr = optimexpr(3)

expr =
  3x1 OptimizationExpression array with properties:

    IndexNames: {{}  {}}
     Variables: [1x1 struct] containing 0 OptimizationVariables

  See expression formulation with show.
```

Create Optimization Expressions Indexed by Strings

Create a string array of color names, and an optimization expression that is indexed by the color names.

```
strexp = ["red","green","blue","yellow"];
expr = optimexpr(strexp)

expr =
  1x4 OptimizationExpression array with properties:

    IndexNames: {{}  {1x4 cell}}
     Variables: [1x1 struct] containing 0 OptimizationVariables

  See expression formulation with show.
```

You can use a cell array of character vectors instead of strings to get the same effect.

```
strexp = {'red','green','blue','yellow'};
expr = optimexpr(strexp)

expr =
  1x4 OptimizationExpression array with properties:

    IndexNames: {{}  {1x4 cell}}
     Variables: [1x1 struct] containing 0 OptimizationVariables

  See expression formulation with show.
```

If `strexp` is 4-by-1 instead of 1-by-4, then `expr` is also 4-by-1:

```
strexp = ["red";"green";"blue";"yellow"];
expr = optimexpr(strexp)

expr =
  4x1 OptimizationExpression array with properties:

    IndexNames: {{1x4 cell}  {}}
     Variables: [1x1 struct] containing 0 OptimizationVariables

  See expression formulation with show.
```

Create Multidimensional Optimization Expressions

Create an empty 3-by-4-by-2 array of optimization expressions.

```
expr = optimexpr(3,4,2)

expr =
  3x4x2 OptimizationExpression array with properties:
```

```
    IndexNames: {{}  {}  {}}
     Variables: [1x1 struct] containing 0 OptimizationVariables

  See expression formulation with show.
```

Create a 3-by-4 array of optimization expressions, where the first dimension is indexed by the strings "brass", "stainless", and "galvanized", and the second dimension is numerically indexed.

```
bnames = ["brass","stainless","galvanized"];
expr = optimexpr(bnames,4)

expr =
  3x4 OptimizationExpression array with properties:

    IndexNames: {{1x3 cell}  {}}
     Variables: [1x1 struct] containing 0 OptimizationVariables

  See expression formulation with show.
```

Create an expression using a named index indicating that each `stainless` expression is 1.5 times the corresponding `x(galvanized)` value.

```
x = optimvar('x',bnames,4);
expr('stainless',:) = x('galvanized',:)*1.5;
show(expr('stainless',:))

('stainless', 1)

  1.5*x('galvanized', 1)

('stainless', 2)

  1.5*x('galvanized', 2)

('stainless', 3)

  1.5*x('galvanized', 3)

('stainless', 4)

  1.5*x('galvanized', 4)
```

Input Arguments

n — Variable dimension
positive integer

Variable dimension, specified as a positive integer.

Example: 4

Data Types: double

cstr — Index names
string array | cell array of character vectors

Index names, specified as a string array or as a cell array of character vectors.

Example: `expr = optimexpr(["Warehouse","Truck","City"])`

Example: `expr = optimexpr({'Warehouse','Truck','City'})`

Data Types: `string` | `cell`

Output Arguments

expr — Optimization expression
`OptimizationExpression` object

Optimization expression, returned as an `OptimizationExpression` object.

Tips

- You can use `optimexpr` to create empty expressions that you fill programmatically, such as in a `for` loop.

  ```
  x = optimvar('x',8);
  expr = optimexpr(4)
  for k = 1:4
      expr(k) = 5*k*(x(2*k) - x(2*k-1));
  end
  ```

- It is generally more efficient to create expressions by vectorized statements rather than loops. See "Create Efficient Optimization Problems" on page 10-28.

See Also

`OptimizationExpression` | `optimconstr` | `show` | `write`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

optimineq

Create empty optimization inequality array

Syntax

```
constr = optimineq(N)
constr = optimineq(cstr)
constr = optimineq(cstr1,N2,...,cstrk)
constr = optimineq({cstr1,cstr2,...,cstrk})
constr = optimineq([N1,N2,...,Nk])
```

Description

Use `optimineq` to initialize a set of inequality expressions.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2.

`constr = optimineq(N)` creates an N-by-1 array of empty optimization inequalities. Use `constr` to initialize a loop that creates inequality expressions.

`constr = optimineq(cstr)` creates an array of empty optimization constraints that are indexed by `cstr`, a cell array of character vectors or string vectors.

If `cstr` is 1-by-$ncstr$, where $ncstr$ is the number of elements of `cstr`, then `constr` is also 1-by-$ncstr$. Otherwise, `constr` is $ncstr$-by-1.

`constr = optimineq(cstr1,N2,...,cstrk)` or `constr = optimineq({cstr1,cstr2,..., cstrk})` or `constr = optimineq([N1,N2,...,Nk])`, for any combination of `cstr` and `N` arguments, creates an $ncstr1$-by-N2-by-...-by-$ncstrk$ array of empty optimization inequalities, where $ncstr$ is the number of elements in `cstr`.

Examples

Create Inequalities in Loop

Create the constraint that a two-element variable `x` must lie in the intersections of a number of disks whose centers and radii are in the arrays `centers` and `radii`.

```
x = optimvar('x',1,2);
centers = [1 -2;3 -4;-2 3];
radii = [6 7 8];
constr = optimineq(length(radii));
for i = 1:length(constr)
    constr(i) = sum((x - centers(i,:)).^2) <= radii(i)^2;
end
```

View the inequality expressions.

```
show(constr)

  arg_LHS <= arg_RHS

  where:

    arg1 = zeros([3, 1]);
    arg1(1) = sum((x - extraParams{1}).^2);
    arg1(2) = sum((x - extraParams{2}).^2);
    arg1(3) = sum((x - extraParams{3}).^2);
    arg_LHS = arg1(:);
    arg1 = zeros([3, 1]);
    arg1(1) = 36;
    arg1(2) = 49;
    arg1(3) = 64;
    arg_RHS = arg1(:);

    extraParams{1}:

      1    -2

  extraParams{2}:

      3    -4

  extraParams{3}:

    -2     3
```

Instead of using a loop, you can create the same constraints by using matrix operations on the variables.

```
constr2 = sum(([x;x;x] - centers).^2,2) <= radii'.^2;
```

Creating inequalities in a loop can be more time consuming than creating inequalities by using matrix operations.

Create Indexed Inequalities in Loop

Create indexed inequalities and variables to represent the calories consumed in a diet. Each meal has a different calorie limit. Create arrays representing the meals, foods, and calories for each food.

```
meals = ["breakfast","lunch","dinner"];
foods = ["cereal","oatmeal","yogurt","peanut butter sandwich","pizza","hamburger",...
    "salad","steak","casserole","ice cream"];
calories = [200,175,150,450,350,800,150,650,350,300]';
```

Create optimization variables representing the foods for each meal, indexed by food names and meal names.

```
diet = optimvar('diet',foods,meals,'LowerBound',0);
```

Set the inequality constraints that each meal has an upper bound on the calories in the meal.

```
constr = optimineq(meals);
for i = 1:3
```

```
    constr(i) = diet(:,i)'*calories <= 250*i;
end
```

View the inequalities for `dinner`.

```
show(constr("dinner"))
```

```
  200*diet('cereal', 'dinner') + 175*diet('oatmeal', 'dinner')
+ 150*diet('yogurt', 'dinner')
+ 450*diet('peanut butter sandwich', 'dinner') + 350*diet('pizza', 'dinner')
+ 800*diet('hamburger', 'dinner') + 150*diet('salad', 'dinner')
+ 650*diet('steak', 'dinner') + 350*diet('casserole', 'dinner')
+ 300*diet('ice cream', 'dinner') <= 750
```

Instead of using a loop, you can create the same inequalities by using matrix operations on the variables.

```
constr2 = diet'*calories <= 250*(1:3)';
```

Include the appropriate index names for the inequalities by setting the `IndexNames` property.

```
constr2.IndexNames = {meals, {}};
```

Display the new inequalities for `dinner`. Note that they are the same as the previous inequalities.

```
show(constr2("dinner"))
```

```
  200*diet('cereal', 'dinner') + 175*diet('oatmeal', 'dinner')
+ 150*diet('yogurt', 'dinner')
+ 450*diet('peanut butter sandwich', 'dinner') + 350*diet('pizza', 'dinner')
+ 800*diet('hamburger', 'dinner') + 150*diet('salad', 'dinner')
+ 650*diet('steak', 'dinner') + 350*diet('casserole', 'dinner')
+ 300*diet('ice cream', 'dinner') <= 750
```

Creating inequalities in a loop can be more time consuming than creating inequalities by using matrix operations.

Input Arguments

N — Size of constraint dimension
positive integer

Size of the constraint dimension, specified as a positive integer.

- The size of `constr = optimineq(N)` is N-by-1.
- The size of `constr = optimineq(N1,N2)` is N1-by-N2.
- The size of `constr = optimineq(N1,N2,...,Nk)` is N1-by-N2-by-...-by-Nk.

Example: 5

Data Types: `double`

cstr — Names for indexing
cell array of character vectors | string vector

Names for indexing, specified as a cell array of character vectors or a string vector.

Example: `{'red','orange','green','blue'}`

Example: `["red";"orange";"green";"blue"]`

Data Types: `string | cell`

Output Arguments

`constr` — Constraints
empty `OptimizationInequality` array

Constraints, returned as an empty `OptimizationInequality` array. Use `constr` to initialize a loop that creates constraint expressions.

For example,

```
x = optimvar('x',8);
constr = optimineq(4);
for k = 1:4
    constr(k) = 5*k*(x(2*k) - x(2*k-1)) <= 10 - 2*k;
end
```

Tips

- It is generally more efficient to create constraints by vectorized expressions rather than loops. See "Create Efficient Optimization Problems" on page 10-28.

- You can use `optimconstr` instead of `optimineq` to create inequality constraints for optimization problems.

See Also
`OptimizationInequality` | `optimconstr` | `optimeq`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2019b

OptimizationConstraint

Optimization constraints

Description

An `OptimizationConstraint` object contains constraints in terms of `OptimizationVariable` objects or `OptimizationExpression` objects. Each constraint uses one of these comparison operators: ==, <=, or >=.

A single statement can represent an array of constraints. For example, you can express the constraints that each row of a matrix variable x sums to one, as shown in "Create Simple Constraints in Loop" on page 16-340.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

Creation

Create an empty constraint object using `optimconstr`. Typically, you use a loop to fill the expressions in the object.

If you create an optimization expressions from optimization variables using a comparison operators ==, <=, or >=, then the resulting object is either an `OptimizationEquality` or an `OptimizationInequality`. See "Compatibility Considerations" on page 16-341.

Include constraints in the `Constraints` property of an optimization problem by using dot notation.

```
prob = optimproblem;
x = optimvar('x',5,3);
rowsum = sum(x,2);
prob.Constraints.rowsum = rowsum;
```

Properties

IndexNames — Index names
'' (default) | cell array of strings | cell array of character vectors

Index names, specified as a cell array of strings or character vectors. For information on using index names, see "Named Index for Optimization Variables" on page 10-20.

Data Types: `cell`

Variables — Optimization variables in object
structure of `OptimizationVariable` objects

This property is read-only.

Optimization variables in the object, specified as a structure of `OptimizationVariable` objects.

Data Types: `struct`

Object Functions

infeasibility	Constraint violation at a point
show	Display information about optimization object
write	Save optimization object description

Examples

Create Simple Constraints in Loop

Create a 5-by-3 optimization variable `x`.

```
x = optimvar('x',5,3);
```

Create the constraint that each row sums to one by using a loop. Initialize the loop using `optimconstr`.

```
rowsum = optimconstr(5);
for i = 1:5
    rowsum(i) = sum(x(i,:)) == 1;
end
```

Inspect the `rowsum` object.

```
rowsum
```

```
rowsum =
  5x1 Linear OptimizationConstraint array with properties:

    IndexNames: {{}  {}}
     Variables: [1x1 struct] containing 1 OptimizationVariable

  See constraint formulation with show.
```

Show the constraints in `rowsum`.

```
show(rowsum)
```

```
(1, 1)

  x(1, 1) + x(1, 2) + x(1, 3) == 1

(2, 1)

  x(2, 1) + x(2, 2) + x(2, 3) == 1

(3, 1)

  x(3, 1) + x(3, 2) + x(3, 3) == 1

(4, 1)

  x(4, 1) + x(4, 2) + x(4, 3) == 1
```

```
(5, 1)

  x(5, 1) + x(5, 2) + x(5, 3) == 1
```

Compatibility Considerations

`OptimizationConstraint` split into `OptimizationEquality` and `OptimizationInequality`
Behavior changed in R2019b

When you use a comparison operator <=, >=, or == on an optimization expression, the result is no longer an `OptimizationConstraint` object. Instead, the equality comparison == returns an `OptimizationEquality` object, and an inequality comparison <= or >= returns an `OptimizationInequality` object. You can use these new objects for defining constraints in an `OptimizationProblem` object, exactly as you would previously for `OptimizationConstraint` objects. Furthermore, you can use `OptimizationEquality` objects to define equations for an `EquationProblem` object.

The new objects make it easier to distinguish between expressions that are suitable for an `EquationProblem` and those that are suitable only for an `OptimizationProblem`. You can use existing `OptimizationConstraint` objects that represent equality constraints in an `EquationProblem` object. Furthermore, when you use an `OptimizationEquality` or an `OptimizationInequality` as a constraint in an `OptimizationProblem`, the software converts the constraint to an `OptimizationConstraint` object.

See Also
`OptimizationEquality` | `OptimizationExpression` | `OptimizationInequality` | `OptimizationProblem` | `OptimizationVariable` | `infeasibility` | `optimconstr` | `show` | `write`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

OptimizationEquality

Equalities and equality constraints

Description

An `OptimizationEquality` object contains equalities and equality constraints in terms of `OptimizationVariable` objects or `OptimizationExpression` objects. Each equality uses the comparison operator ==.

A single statement can represent an array of equalities. For example, you can express the equalities that each row of a matrix variable x sums to one in this single statement:

```
constrsum = sum(x,2) == 1
```

Use `OptimizationEquality` objects as constraints in an `OptimizationProblem`, or as equations in an `EquationProblem`.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

Creation

Create equalities using optimization expressions with the comparison operator ==.

Include equalities in the `Constraints` property of an optimization problem, or the `Equations` property of an equation problem, by using dot notation.

```
prob = optimproblem;
x = optimvar('x',4,6);
SumToOne = sum(x,2) == 1;
prob.Constraints.SumToOne = SumToOne;
% Or for an equation problem:
eqprob = eqnproblem;
eqprob.Equations.SumToOne = SumToOne;
```

You can also create an empty optimization equality by using `optimeq` or `optimconstr`. Typically, you then set the equalities in a loop. For an example, see "Create Equalities in Loop" on page 16-344. However, for the most efficient problem formulation, avoid setting equalities in loops. See "Create Efficient Optimization Problems" on page 10-28.

Properties

IndexNames — Index names
`''` (default) | cell array of strings | cell array of character vectors

Index names, specified as a cell array of strings or character vectors. For information on using index names, see "Named Index for Optimization Variables" on page 10-20.

Data Types: `cell`

Variables — Optimization variables in object
structure of `OptimizationVariable` objects

This property is read-only.

Optimization variables in the object, specified as a structure of `OptimizationVariable` objects.

Data Types: `struct`

Object Functions

infeasibility	Constraint violation at a point
show	Display information about optimization object
write	Save optimization object description

Examples

Create `OptimizationEquality` Array

Create a 4-by-6 optimization variable matrix named x.

```
x = optimvar('x',4,6);
```

Create the equalities that each row of x sums to one.

```
constrsum = sum(x,2) == 1

constrsum =
  4x1 Linear OptimizationEquality array with properties:

    IndexNames: {{}  {}}
     Variables: [1x1 struct] containing 1 OptimizationVariable

  See equality formulation with show.
```

View the equalities.

```
show(constrsum)

(1, 1)

  x(1, 1) + x(1, 2) + x(1, 3) + x(1, 4) + x(1, 5) + x(1, 6) == 1

(2, 1)

  x(2, 1) + x(2, 2) + x(2, 3) + x(2, 4) + x(2, 5) + x(2, 6) == 1

(3, 1)

  x(3, 1) + x(3, 2) + x(3, 3) + x(3, 4) + x(3, 5) + x(3, 6) == 1

(4, 1)

  x(4, 1) + x(4, 2) + x(4, 3) + x(4, 4) + x(4, 5) + x(4, 6) == 1
```

To include the equalities in an optimization problem, set a `Constraints` property to `constrsum` by using dot notation.

```
prob = optimproblem;
prob.Constraints.constrsum = constrsum

prob =
  OptimizationProblem with properties:

        Description: ''
     ObjectiveSense: 'minimize'
          Variables: [1x1 struct] containing 1 OptimizationVariable
          Objective: [0x0 OptimizationExpression]
        Constraints: [1x1 struct] containing 1 OptimizationConstraint

  See problem formulation with show.
```

Similarly, to include the equalities in an equation problem, set a `Constraints` property to `constrsum` by using dot notation.

```
eqnprob = eqnproblem;
eqnprob.Equations.constrsum = constrsum

eqnprob =
  EquationProblem with properties:

    Description: ''
      Variables: [1x1 struct] containing 1 OptimizationVariable
      Equations: [1x1 struct] containing 1 OptimizationEquality

  See problem formulation with show.
```

Create Equalities in Loop

Create an empty `OptimizationEquality` object.

```
eq1 = optimeq;
```

Create a 5-by-5 optimization variable array named `x`.

```
x = optimvar('x',5,5);
```

Create the equalities that row i of `x` sums to i^2.

```
for i = 1:size(x,1)
    eq1(i) = sum(x(i,:)) == i^2;
end
```

View the resulting equalities.

```
show(eq1)
```

```
(1, 1)
```

```
  x(1, 1) + x(1, 2) + x(1, 3) + x(1, 4) + x(1, 5) == 1

(1, 2)

  x(2, 1) + x(2, 2) + x(2, 3) + x(2, 4) + x(2, 5) == 4

(1, 3)

  x(3, 1) + x(3, 2) + x(3, 3) + x(3, 4) + x(3, 5) == 9

(1, 4)

  x(4, 1) + x(4, 2) + x(4, 3) + x(4, 4) + x(4, 5) == 16

(1, 5)

  x(5, 1) + x(5, 2) + x(5, 3) + x(5, 4) + x(5, 5) == 25
```

To use `eq1` as a constraint in an optimization problem, set `eq1` as a `Constraints` property by using dot notation.

```
prob = optimproblem;
prob.Constraints.eq1 = eq1;
```

Similarly, to use `eq1` as a set of equations in an equation problem, set `eq1` as an `Equations` property by using dot notation.

```
eqprob = eqnproblem;
eqprob.Equations.eq1 = eq1;
```

See Also

EquationProblem | OptimizationConstraint | OptimizationExpression | OptimizationInequality | OptimizationProblem | OptimizationVariable | eqnproblem | infeasibility | optimconstr | optimeq | show | write

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2019b

OptimizationExpression

Arithmetic or functional expression in terms of optimization variables

Description

An `OptimizationExpression` is an arithmetic or functional expression in terms of optimization variables. Use an `OptimizationExpression` as an objective function, or as a part of an inequality or equality in a constraint or equation.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

Creation

Create an optimization expression by performing operations on `OptimizationVariable` objects. Use standard MATLAB arithmetic including taking powers, indexing, and concatenation of optimization variables to create expressions. See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Examples" on page 16-0 .

You can also create an optimization expression from a MATLAB function applied to optimization variables by using `fcn2optimexpr`. For examples, see "Create Expression from Nonlinear Function" on page 16-349 and "Problem-Based Nonlinear Optimization".

Create an empty optimization expression by using `optimexpr`. Typically, you then fill the expression in a loop. For examples, see "Create Optimization Expression by Looping" on page 16-348 and the `optimexpr` function reference page.

After you create an expression, use it as either an objective function, or as part of a constraint or equation. For examples, see the `solve` function reference page.

Properties

`IndexNames` — Index names
`' '` (default) | cell array of strings | cell array of character vectors

Index names, specified as a cell array of strings or character vectors. For information on using index names, see "Named Index for Optimization Variables" on page 10-20.

Data Types: `cell`

`Variables` — Optimization variables in object
structure of `OptimizationVariable` objects

This property is read-only.

Optimization variables in the object, specified as a structure of `OptimizationVariable` objects.

Data Types: `struct`

Object Functions

evaluate	Evaluate optimization expression
show	Display information about optimization object
write	Save optimization object description

Examples

Create Optimization Expressions by Arithmetic Operations

Create optimization expressions by arithmetic operations on optimization variables.

```
x = optimvar('x',3,2);
expr = sum(sum(x))

expr =
  Linear OptimizationExpression

    x(1, 1) + x(2, 1) + x(3, 1) + x(1, 2) + x(2, 2) + x(3, 2)
```

```
f = [2,10,4];
w = f*x;
show(w)

(1, 1)

  2*x(1, 1) + 10*x(2, 1) + 4*x(3, 1)

(1, 2)

  2*x(1, 2) + 10*x(2, 2) + 4*x(3, 2)
```

Create Optimization Expressions by Index and Array Operations

Create an optimization expression by transposing an optimization variable.

```
x = optimvar('x',3,2);
y = x'

y =
  2x3 Linear OptimizationExpression array with properties:

    IndexNames: {{}  {}}
     Variables: [1x1 struct] containing 1 OptimizationVariable

  See expression formulation with show.
```

Simply indexing into an optimization array does not create an expression, but instead creates an optimization variable that references the original variable. To see this, create a variable w that is the first and third row of x. Note that w is an optimization variable, not an optimization expression.

```
w = x([1,3],:)
```

```
w =
  2x2 OptimizationVariable array with properties:

  Read-only array-wide properties:
          Name: 'x'
          Type: 'continuous'
    IndexNames: {{}  {}}

  Elementwise properties:
    LowerBound: [2x2 double]
    UpperBound: [2x2 double]

  Reference to a subset of OptimizationVariable with Name 'x'.

  See variables with show.
  See bounds with showbounds.
```

Create an optimization expression by concatenating optimization variables.

```
y = optimvar('y',4,3);
z = optimvar('z',4,7);
f = [y,z]

f =
  4x10 Linear OptimizationExpression array with properties:

    IndexNames: {{}  {}}
      Variables: [1x1 struct] containing 2 OptimizationVariables

  See expression formulation with show.
```

Create Optimization Expression by Looping

Use `optimexpr` to create an empty expression, then fill the expression in a loop.

```
y = optimvar('y',6,4);
expr = optimexpr(3,2);
for i = 1:3
    for j = 1:2
        expr(i,j) = y(2*i,j) - y(i,2*j);
    end
end
show(expr)

(1, 1)

  y(2, 1) - y(1, 2)

(2, 1)

  y(4, 1) - y(2, 2)

(3, 1)
```

```
  y(6, 1) - y(3, 2)

(1, 2)

  y(2, 2) - y(1, 4)

(2, 2)

  y(4, 2) - y(2, 4)

(3, 2)

  y(6, 2) - y(3, 4)
```

Create Expression from Nonlinear Function

Create an optimization expression corresponding to the objective function

$$f(x) = x^2/10 + \exp(-\exp(-x)).$$

```
x = optimvar('x');
f = x^2/10 + exp(-exp(-x))

f =
  Nonlinear OptimizationExpression

    ((x.^2 ./ 10) + exp(-exp(-x)))
```

Find the point that minimizes fun starting from the point x0 = 0.

```
x0 = struct('x',0);
prob = optimproblem('Objective',f);
[sol,fval] = solve(prob,x0)

Solving problem using fminunc.

Local minimum found.

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

sol = struct with fields:
    x: -0.9595


fval = 0.1656
```

If f were not a supported function, you would convert it using fcn2optimexpr. See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8.

```
f = @(x)x^2/10 + exp(-exp(-x));
fun = fcn2optimexpr(f,x)

fun =
  Nonlinear OptimizationExpression
```

```
    anonymousFunction1(x)

  where:

    anonymousFunction1 = @(x)x^2/10+exp(-exp(-x));
```

```
prob = optimproblem('Objective',fun);
[sol,fval] = solve(prob,x0)
```

```
Solving problem using fminunc.

Local minimum found.

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

sol = struct with fields:
    x: -0.9595


fval = 0.1656
```

Copyright 2018–2020 The MathWorks, Inc.

Evaluate Optimization Expression At Point

Create an optimization expression in two variables.

```
x = optimvar('x',3,2);
y = optimvar('y',1,2);
expr = sum(x,1) - 2*y;
```

Evaluate the expression at a point.

```
xmat = [3,-1;
    0,1;
    2,6];
sol.x = xmat;
sol.y = [4,-3];
val = evaluate(expr,sol)

val = 1×2

    -3    12
```

More About

Arithmetic Operations

For the list of supported operations on optimization expressions, see "Supported Operations on Optimization Variables and Expressions" on page 10-43.

See Also

OptimizationVariable | evaluate | fcn2optimexpr | optimexpr | show | solve | write

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2
"Optimization Expressions" on page 10-6

Introduced in R2017b

OptimizationInequality

Inequality constraints

Description

An `OptimizationInequality` object contains an inequality constraint in terms of `OptimizationVariable` objects or `OptimizationExpression` objects. An inequality constraint uses the comparison operator <= or >=.

A single statement can represent an array of inequalities. For example, you can express the inequalities that each row of a matrix variable x sums to no more than one in this single statement:

```
constrsum = sum(x,2) <= 1
```

Use `OptimizationInequality` objects as constraints in an `OptimizationProblem`.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2.

Creation

Create an inequality using optimization expressions with the comparison operator <= or >=.

Include inequalities in the `Constraints` property of an optimization problem by using dot notation.

```
prob = optimproblem;
x = optimvar('x',4,6);
SumLessThanOne = sum(x,2) <= 1;
prob.Constraints.SumLessThanOne = SumLessThanOne;
```

You can also create an empty optimization inequality by using `optimineq` or `optimconstr`. Typically, you then set the inequalities in a loop. For an example, see "Create Inequalities in Loop" on page 16-354. However, for the most efficient problem formulation, avoid setting inequalities in loops. See "Create Efficient Optimization Problems" on page 10-28.

Properties

IndexNames — Index names
`''` (default) | cell array of strings | cell array of character vectors

Index names, specified as a cell array of strings or character vectors. For information on using index names, see "Named Index for Optimization Variables" on page 10-20.

Data Types: `cell`

Variables — Optimization variables in object
structure of `OptimizationVariable` objects

This property is read-only.

Optimization variables in the object, specified as a structure of `OptimizationVariable` objects.

Data Types: `struct`

Object Functions

infeasibility	Constraint violation at a point
show	Display information about optimization object
write	Save optimization object description

Examples

Create `OptimizationInequality` Array

Create a 4-by-6 optimization variable matrix named x.

```
x = optimvar('x',4,6);
```

Create the inequalities that each row of x sums to no more than one.

```
constrsum = sum(x,2) <= 1

constrsum =
  4x1 Linear OptimizationInequality array with properties:

    IndexNames: {{}  {}}
     Variables: [1x1 struct] containing 1 OptimizationVariable

  See inequality formulation with show.
```

View the inequalities.

```
show(constrsum)

(1, 1)

  x(1, 1) + x(1, 2) + x(1, 3) + x(1, 4) + x(1, 5) + x(1, 6) <= 1

(2, 1)

  x(2, 1) + x(2, 2) + x(2, 3) + x(2, 4) + x(2, 5) + x(2, 6) <= 1

(3, 1)

  x(3, 1) + x(3, 2) + x(3, 3) + x(3, 4) + x(3, 5) + x(3, 6) <= 1

(4, 1)

  x(4, 1) + x(4, 2) + x(4, 3) + x(4, 4) + x(4, 5) + x(4, 6) <= 1
```

To include the inequalities in an optimization problem, set a `Constraints` property to `constrsum` by using dot notation.

```
prob = optimproblem;
prob.Constraints.constrsum = constrsum
```

```
prob =
  OptimizationProblem with properties:

       Description: ''
    ObjectiveSense: 'minimize'
         Variables: [1x1 struct] containing 1 OptimizationVariable
         Objective: [0x0 OptimizationExpression]
       Constraints: [1x1 struct] containing 1 OptimizationConstraint

  See problem formulation with show.
```

Create Inequalities in Loop

Create the constraint that a two-element variable x must lie in the intersections of a number of disks whose centers and radii are in the arrays `centers` and `radii`.

```
x = optimvar('x',1,2);
centers = [1 -2;3 -4;-2 3];
radii = [6 7 8];
constr = optimineq(length(radii));
for i = 1:length(constr)
    constr(i) = sum((x - centers(i,:)).^2) <= radii(i)^2;
end
```

View the inequality expressions.

```
show(constr)
```

```
  arg_LHS <= arg_RHS

  where:

    arg1 = zeros([3, 1]);
    arg1(1) = sum((x - extraParams{1}).^2);
    arg1(2) = sum((x - extraParams{2}).^2);
    arg1(3) = sum((x - extraParams{3}).^2);
    arg_LHS = arg1(:);
    arg1 = zeros([3, 1]);
    arg1(1) = 36;
    arg1(2) = 49;
    arg1(3) = 64;
    arg_RHS = arg1(:);

    extraParams{1}:

      1    -2

    extraParams{2}:

      3    -4

    extraParams{3}:

     -2     3
```

Instead of using a loop, you can create the same constraints by using matrix operations on the variables.

```
constr2 = sum(([x;x;x] - centers).^2,2) <= radii'.^2;
```

Creating inequalities in a loop can be more time consuming than creating inequalities by using matrix operations.

See Also

OptimizationConstraint | OptimizationEquality | OptimizationExpression | OptimizationProblem | OptimizationVariable | infeasibility | optimconstr | optimineq | show | write

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2019b

OptimizationProblem

Optimization problem

Description

An `OptimizationProblem` object describes an optimization problem, including variables for the optimization, constraints, the objective function, and whether the objective is to be maximized or minimized. Solve a complete problem using `solve`.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2.

Creation

Create an `OptimizationProblem` object by using `optimproblem`.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

Properties

Description — Problem label
'' (default) | string | character vector

Problem label, specified as a string or character vector. The software does not use `Description`. It is an arbitrary label that you can use for any reason. For example, you can share, archive, or present a model or problem, and store descriptive information about the model or problem in the `Description` property.

Example: `"Describes a traveling salesman problem"`

Data Types: `char` | `string`

ObjectiveSense — Indication to minimize or maximize
'minimize' (default) | 'min' | 'maximize' | 'max'

Indication to minimize or maximize, specified as `'minimize'` or `'maximize'`. This property affects how `solve` works.

You can use the short name `'min'` for `'minimize'` or `'max'` for `'maximize'`.

Example: `'maximize'`

Data Types: `char` | `string`

Variables — Optimization variables in object
structure of `OptimizationVariable` objects

This property is read-only.

Optimization variables in the object, specified as a structure of `OptimizationVariable` objects.

Data Types: `struct`

`Objective` — Objective function
scalar `OptimizationExpression` | structure containing scalar `OptimizationExpression`

Objective function, specified as a scalar `OptimizationExpression` or as a structure containing a scalar `OptimizationExpression`. Incorporate an objective function into the problem when you create the problem, or later by using dot notation.

```
prob = optimproblem('Objective',5*brownies + 2*cookies)
% or
prob = optimproblem;
prob.Objective = 5*brownies + 2*cookies
```

`Constraints` — Optimization constraints
`OptimizationConstraint` object | `OptimizationEquality` object | `OptimizationInequality` object | structure containing `OptimizationConstraint`, `OptimizationEquality`, or `OptimizationInequality` objects

Optimization constraints, specified as an `OptimizationConstraint` object, an `OptimizationEquality` object, an `OptimizationInequality` object, or as a structure containing one of these objects. Incorporate constraints into the problem when you create the problem, or later by using dot notation:

```
constrs = struct('TrayArea',10*brownies + 20*cookies <= traysize,...
    'TrayWeight',12*brownies + 18*cookies <= maxweight);
prob = optimproblem('Constraints',constrs)
% or
prob.Constraints.TrayArea = 10*brownies + 20*cookies <= traysize
prob.Constraints.TrayWeight = 12*brownies + 18*cookies <= maxweight
```

Remove a constraint by setting it to `[]`.

```
prob.Constraints.TrayArea = [];
```

Object Functions

optimoptions	Create optimization options
prob2struct	Convert optimization problem or equation problem to solver form
show	Display information about optimization object
solve	Solve optimization problem or equation problem
varindex	Map problem variables to solver-based variable index
write	Save optimization object description

Examples

Create and Solve Maximization Problem

Create a linear programming problem for maximization. The problem has two positive variables and three linear inequality constraints.

```
prob = optimproblem('ObjectiveSense','max');
```

Create positive variables. Include an objective function in the problem.

```
x = optimvar('x',2,1,'LowerBound',0);
prob.Objective = x(1) + 2*x(2);
```

Create linear inequality constraints in the problem.

```
cons1 = x(1) + 5*x(2) <= 100;
cons2 = x(1) + x(2) <= 40;
cons3 = 2*x(1) + x(2)/2 <= 60;
prob.Constraints.cons1 = cons1;
prob.Constraints.cons2 = cons2;
prob.Constraints.cons3 = cons3;
```

Review the problem.

```
show(prob)
```

```
  OptimizationProblem :

    Solve for:
       x

    maximize :
       x(1) + 2*x(2)


    subject to cons1:
       x(1) + 5*x(2) <= 100

    subject to cons2:
       x(1) + x(2) <= 40

    subject to cons3:
       2*x(1) + 0.5*x(2) <= 60

    variable bounds:
       0 <= x(1)
       0 <= x(2)
```

Solve the problem.

```
sol = solve(prob);
```

```
Solving problem using linprog.
```

```
Optimal solution found.
```

```
sol.x
```

```
ans = 2×1
```

```
    25.0000
    15.0000
```

See Also

OptimizationConstraint | OptimizationExpression | OptimizationVariable | optimproblem | show | solve | write

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

OptimizationVariable

Variable for optimization

Description

An `OptimizationVariable` object contains variables for optimization expressions. Use expressions to represent an objective function, constraints, or equations. Variables are symbolic in nature, and can be arrays of any size.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

Creation

Create an `OptimizationVariable` object using `optimvar`.

Properties

Array-Wide Properties

Name — Variable name
string | character vector

This property is read-only.

Variable name, specified as a string or character vector.

`Name` gives the variable label to be displayed, such as in `show` or `write`. `Name` also gives the field names in the solution structure that `solve` returns.

Tip To avoid confusion, set `name` to be the MATLAB variable name. For example,

```
metal = optimvar('metal')
```

Data Types: `char` | `string`

Type — Variable type
`'continuous'` (default) | `'integer'`

Variable type, specified as `'continuous'` or `'integer'`.

- `'continuous'` – Real values
- `'integer'` – Integer values

The variable type applies to all variables in the array. To have multiple variable types, create multiple variables.

Tip To specify a binary variable, use the `'integer'` type and specify LowerBound = 0 and UpperBound = 1.

Data Types: `char` | `string`

IndexNames — Index names
`''` (default) | cell array of strings | cell array of character vectors

Index names, specified as a cell array of strings or character vectors. For information on using index names, see "Named Index for Optimization Variables" on page 10-20.

Data Types: `cell`

Element-wise Properties

LowerBound — Lower bound
`-Inf` (default) | real scalar | real array

Lower bound, specified as a real scalar or as a real array having the same dimensions as the `OptimizationVariable` object. Scalar values apply to all elements of the variable.

The `LowerBound` property is always displayed as an array. However, you can set the property as a scalar that applies to all elements. For example,

`var.LowerBound = 0`

Data Types: `double`

UpperBound — Upper bound
`Inf` (default) | real scalar | real array

Upper bound, specified as a real scalar or as a real array having the same dimensions as the `OptimizationVariable` object. Scalar values apply to all elements of the variable.

The `UpperBound` property is always displayed as an array. However, you can set the property as a scalar that applies to all elements. For example

`var.UpperBound = 1`

Data Types: `double`

Object Functions

show   Display information about optimization object
showbounds   Display variable bounds
write   Save optimization object description
writebounds   Save description of variable bounds

Examples

Create Scalar Optimization Variable

Create a scalar optimization variable named `dollars`.

`dollars = optimvar('dollars')`

```
dollars =
  OptimizationVariable with properties:

         Name: 'dollars'
         Type: 'continuous'
    IndexNames: {{}  {}}
    LowerBound: -Inf
    UpperBound: Inf

  See variables with show.
  See bounds with showbounds.
```

Create Optimization Variable Vector

Create a 3-by-1 optimization variable vector named x.

```
x = optimvar('x',3)

x =
  3x1 OptimizationVariable array with properties:

  Array-wide properties:
         Name: 'x'
         Type: 'continuous'
    IndexNames: {{}  {}}

  Elementwise properties:
    LowerBound: [3x1 double]
    UpperBound: [3x1 double]

  See variables with show.
  See bounds with showbounds.
```

Create Optimization Variables Indexed by Strings

Create an integer optimization variable vector named bolts that is indexed by the strings "brass", "stainless", and "galvanized". Use the indices of bolts to create an optimization expression, and experiment with creating bolts using character arrays or in a different orientation.

Create bolts using strings in a row orientation.

```
bnames = ["brass","stainless","galvanized"];
bolts = optimvar('bolts',bnames,'Type','integer')

bolts =
  1x3 OptimizationVariable array with properties:

  Array-wide properties:
         Name: 'bolts'
         Type: 'integer'
    IndexNames: {{}  {1x3 cell}}

  Elementwise properties:
    LowerBound: [-Inf -Inf -Inf]
    UpperBound: [Inf Inf Inf]
```

```
  See variables with show.
  See bounds with showbounds.
```

Create an optimization expression using the string indices.

```
y = bolts("brass") + 2*bolts("stainless") + 4*bolts("galvanized")

y =
  Linear OptimizationExpression

    bolts('brass') + 2*bolts('stainless') + 4*bolts('galvanized')
```

Use a cell array of character vectors instead of strings to get a variable with the same indices as before.

```
bnames = {'brass','stainless','galvanized'};
bolts = optimvar('bolts',bnames,'Type','integer')

bolts =
  1x3 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'bolts'
          Type: 'integer'
    IndexNames: {{}  {1x3 cell}}

  Elementwise properties:
    LowerBound: [-Inf -Inf -Inf]
    UpperBound: [Inf Inf Inf]

  See variables with show.
  See bounds with showbounds.
```

Use a column-oriented version of bnames, 3-by-1 instead of 1-by-3, and observe that bolts has that orientation as well.

```
bnames = ["brass";"stainless";"galvanized"];
bolts = optimvar('bolts',bnames,'Type','integer')

bolts =
  3x1 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'bolts'
          Type: 'integer'
    IndexNames: {{1x3 cell}  {}}

  Elementwise properties:
    LowerBound: [3x1 double]
    UpperBound: [3x1 double]

  See variables with show.
  See bounds with showbounds.
```

Create Multidimensional Optimization Variables

Create a 3-by-4-by-2 array of optimization variables named `xarray`.

```
xarray = optimvar('xarray',3,4,2)

xarray =
  3x4x2 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'xarray'
          Type: 'continuous'
    IndexNames: {{}  {}  {}}

  Elementwise properties:
    LowerBound: [3x4x2 double]
    UpperBound: [3x4x2 double]

  See variables with show.
  See bounds with showbounds.
```

You can also create multidimensional variables indexed by a mixture of names and numeric indices. For example, create a 3-by-4 array of optimization variables where the first dimension is indexed by the strings `'brass'`, `'stainless'`, and `'galvanized'`, and the second dimension is numerically indexed.

```
bnames = ["brass","stainless","galvanized"];
bolts = optimvar('bolts',bnames,4)

bolts =
  3x4 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'bolts'
          Type: 'continuous'
    IndexNames: {{1x3 cell}  {}}

  Elementwise properties:
    LowerBound: [3x4 double]
    UpperBound: [3x4 double]

  See variables with show.
  See bounds with showbounds.
```

Create Binary Optimization Variables

Create an optimization variable named `x` of size 3-by-3-by-3 that represents binary variables.

```
x = optimvar('x',3,3,3,'Type','integer','LowerBound',0,'UpperBound',1)

x =
  3x3x3 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'x'
          Type: 'integer'
```

```
   IndexNames: {{}  {}  {}}

 Elementwise properties:
   LowerBound: [3x3x3 double]
   UpperBound: [3x3x3 double]

 See variables with show.
 See bounds with showbounds.
```

More About

Arithmetic Operations

For the list of supported operations on optimization variables, see "Supported Operations on Optimization Variables and Expressions" on page 10-43.

Tips

• `OptimizationVariable` objects have *handle* copy behavior. See "Handle Object Behavior" and "Comparison of Handle and Value Classes". Handle copy behavior means that a copy of an `OptimizationVariable` points to the original and does not have an independent existence. For example, create a variable x, copy it to y, then set a property of y. Note that x takes on the new property value.

```
x = optimvar('x','LowerBound',1);
y = x;
y.LowerBound = 0;
showbounds(x)


    0 <= x
```

See Also
`OptimizationConstraint` | `OptimizationExpression` | `OptimizationProblem` | `optimvar` | `show` | `showbounds` | `write` | `writebounds`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2
"Supported Operations on Optimization Variables and Expressions" on page 10-43

Introduced in R2017b

Optimize

Optimize or solve equations in the Live Editor

Description

The **Optimize** task lets you interactively optimize linear and nonlinear objective functions subject to constraints of various types, and to solve nonlinear systems of equations. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Choose a solver based on the characteristics of your problem. If you have Global Optimization Toolbox, you can choose to use its solvers as well.
- Specify the objective and constraint functions, either by writing functions or browsing for functions.
- Specify solver options.
- Run the optimization.

For suggestions on how to use **Optimize**, see "Use Optimize Live Editor Task Effectively" on page 1-39. Currently, you cannot use the `fseminf`, `GlobalSearch`, or `MultiStart` solvers with **Optimize**.

For general information about Live Editor tasks, see "Add Interactive Tasks to a Live Script".

Open the Task

To add the **Optimize** task to a live script in the MATLAB Editor, on the Live Editor **Insert** tab, select **Task > Optimize**.

Alternatively, in a code block in the script, type a relevant keyword, such as `optim` or `fmincon`. Select **Optimize** from the suggested command completions.
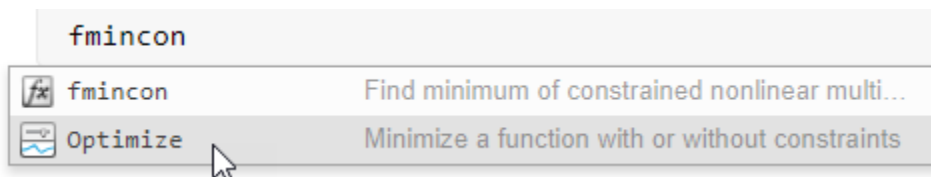
Parameters

`Objective` — Objective function type
`Linear` | `Quadratic` | `Least squares` | `Nonlinear` | `Nonsmooth`

Objective function type, specified by clicking the appropriate labeled button. The selected objective function determines which solvers are available and which solver is recommended for the problem (see `Solver`).

`Constraints` — Constraint types
`Unconstrained` | `Lower bounds` | `Upper bounds` | `Linear inequality` | `Linear equality` | `Second-order cone` | `Nonlinear` | `Integer`

Constraint types, specified by clicking the appropriate labeled buttons. You can specify more than one constraint type. The selected constraints determine which solvers are available and which solver is recommended for the problem (see `Solver`).

Solver — Optimization solver
solver name

Optimization solver that MATLAB uses to solve the problem, specified by selecting a solver from the list of available solvers. The available solvers and the recommended solver depend on your license and the selected `Objective` and `Constraints`.

Available Solvers

Constraint Type	Objective Type				
	Linear	**Quadratic**	**Least Squares**	**Nonlinear**	**Nonsmooth**
Unconstrained	Solution is trivial	quadprog	lsqcurvefit, lsqnonlin, lsqnonneg, lsqlin	fgoalattain, fminsearch, fminimax , fminunc, fsolve, fzero, ga, gamultiobj, paretosearch, patternsearch, particleswarm, simulannealbnd	fminsearch, ga, gamultiobj, paretosearch, patternsearch, particleswarm, simulannealbnd
Bounds Only	linprog	quadprog	lsqcurvefit, lsqnonlin, lsqnonneg, lsqlin	fgoalattain, fminbnd, fmincon, fminimax, ga, gamultiobj, paretosearch, patternsearch, particleswarm, simulannealbnd, surrogateopt	fminbnd, ga, gamultiobj, paretosearch, patternsearch, particleswarm, simulannealbnd, surrogateopt
Linear	linprog	quadprog	lsqlin, fmincon	fgoalattain, fmincon, fminimax, ga, gamultiobj, paretosearch, patternsearch	ga, gamultiobj, paretosearch, patternsearch
Linear + Integer	intlinprog	ga	ga	ga	ga

Constraint Type	Objective Type				
	Linear	**Quadratic**	**Least Squares**	**Nonlinear**	**Nonsmooth**
Second-order cone	coneprog	fgoalattain, fmincon, fminimax, ga, gamultiobj, paretosearch, patternsearch, surrogateopt	fgoalattain, fmincon, fminimax, ga, gamultiobj, paretosearch, patternsearch, surrogateopt	fgoalattain, fmincon, fminimax, ga, gamultiobj, paretosearch, patternsearch, surrogateopt	ga, gamultiobj, paretosearch, patternsearch, surrogateopt
Nonlinear	fgoalattain, fmincon, fminimax, ga, gamultiobj, paretosearch, patternsearch, surrogateopt	fgoalattain, fmincon, fminimax, ga, gamultiobj, paretosearch, patternsearch, surrogateopt	fgoalattain, fmincon, fminimax, ga, gamultiobj, paretosearch, patternsearch, surrogateopt	fgoalattain, fmincon, fminimax, ga, gamultiobj, paretosearch, patternsearch, surrogateopt	ga, gamultiobj, paretosearch, patternsearch, surrogateopt
Nonlinear + Integer	ga, surrogateopt	ga, surrogateopt	ga, surrogateopt	ga, surrogateopt	ga, surrogateopt

Example: `fmincon`

Tips

- For functions with extra inputs, **Optimize** requires you to choose the optimization variable, and to specify which workspace variables contain the fixed data inputs. For example, see "Place Optimization Variables in One Vector and Data in Other Variables" on page 1-40, which contains three function inputs:

  Optimize generates code only after you specify all function inputs.
- **Optimize** cannot parse a function containing the `varargin` input or a function that contains an error.
- If you select an objective or nonlinear constraint function from a file, **Optimize** adds the file location to your MATLAB path.
- If **Optimize** has a parsing error or if multiple local functions have the same name, the list of available local functions is empty.

See Also

Functions
fmincon | intlinprog | patternsearch | surrogateopt

Topics
"Solve a Constrained Nonlinear Problem, Solver-Based" on page 1-11
"Get Started with Optimize Live Editor Task" on page 1-34
"Optimize Live Editor Task with fmincon Solver" on page 6-77
"Optimize Live Editor Task with lsqlin Solver" on page 12-28
"Optimize Using the GPS Algorithm" (Global Optimization Toolbox)
"Minimize Function with Many Local Minima" (Global Optimization Toolbox)
"Pareto Front for Two Objectives" (Global Optimization Toolbox)
"Use Optimize Live Editor Task Effectively" on page 1-39
"Solver-Based Optimization Problem Setup"
How to Use the Optimize Live Editor Task

Introduced in R2020b

optimoptions

Package: optim.problemdef

Create optimization options

Syntax

```
options = optimoptions(SolverName)
options = optimoptions(SolverName,Name,Value)

options = optimoptions(oldoptions,Name,Value)

options = optimoptions(SolverName,oldoptions)

options = optimoptions(prob)
options = optimoptions(prob,Name,Value)
```

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`.

`options = optimoptions(prob)` returns a set of default options for the `prob` optimization problem or equation problem.

`options = optimoptions(prob,Name,Value)` returns options with the named parameters altered with the specified values.

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', 'sqp-legacy', 'trust-region-reflective')
```

```
Set properties:
  No options set.

Default properties:
                Algorithm: 'interior-point'
            CheckGradients: 0
       ConstraintTolerance: 1.0000e-06
                   Display: 'final'
   FiniteDifferenceStepSize: 'sqrt(eps)'
      FiniteDifferenceType: 'forward'
      HessianApproximation: 'bfgs'
                HessianFcn: []
         HessianMultiplyFcn: []
               HonorBounds: 1
     MaxFunctionEvaluations: 3000
             MaxIterations: 1000
             ObjectiveLimit: -1.0000e+20
        OptimalityTolerance: 1.0000e-06
                 OutputFcn: []
                   PlotFcn: []
              ScaleProblem: 0
 SpecifyConstraintGradient: 0
  SpecifyObjectiveGradient: 0
             StepTolerance: 1.0000e-10
        SubproblemAlgorithm: 'factorization'
                  TypicalX: 'ones(numberOfVariables,1)'
                UseParallel: 0

Show options not used by current Algorithm ('interior-point')
```

Create Nondefault Options

Create nondefault options for the `fmincon` solver.

```
options = optimoptions(@fmincon,'Algorithm','sqp','MaxIterations',1500)

options =
  fmincon options:

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

  Set properties:
                Algorithm: 'sqp'
            MaxIterations: 1500

  Default properties:
            CheckGradients: 0
       ConstraintTolerance: 1.0000e-06
                   Display: 'final'
   FiniteDifferenceStepSize: 'sqrt(eps)'
      FiniteDifferenceType: 'forward'
     MaxFunctionEvaluations: '100*numberOfVariables'
```

```
              ObjectiveLimit: -1.0000e+20
          OptimalityTolerance: 1.0000e-06
                     OutputFcn: []
                       PlotFcn: []
                  ScaleProblem: 0
    SpecifyConstraintGradient: 0
     SpecifyObjectiveGradient: 0
                 StepTolerance: 1.0000e-06
                      TypicalX: 'ones(numberOfVariables,1)'
                    UseParallel: 0

   Show options not used by current Algorithm ('sqp')
```

Update Options

Update existing options with new values.

Create options for the `lsqnonlin` solver.

```
oldoptions = optimoptions(@lsqnonlin,'Algorithm','levenberg-marquardt',...
    'MaxFunctionEvaluations',1500)

oldoptions =
  lsqnonlin options:

   Options used by current Algorithm ('levenberg-marquardt'):
   (Other available algorithms: 'trust-region-reflective')

   Set properties:
                     Algorithm: 'levenberg-marquardt'
        MaxFunctionEvaluations: 1500

   Default properties:
                CheckGradients: 0
                       Display: 'final'
      FiniteDifferenceStepSize: 'sqrt(eps)'
          FiniteDifferenceType: 'forward'
             FunctionTolerance: 1.0000e-06
                 MaxIterations: 400
                     OutputFcn: []
                       PlotFcn: []
      SpecifyObjectiveGradient: 0
                 StepTolerance: 1.0000e-06
                      TypicalX: 'ones(numberOfVariables,1)'
                    UseParallel: 0

   Show options not used by current Algorithm ('levenberg-marquardt')
```

Increase `MaxFunctionEvaluations` to 2000.

```
options = optimoptions(oldoptions,'MaxFunctionEvaluations',2000)

options =
  lsqnonlin options:
```

```
        Options used by current Algorithm ('levenberg-marquardt'):
        (Other available algorithms: 'trust-region-reflective')

        Set properties:
                      Algorithm: 'levenberg-marquardt'
            MaxFunctionEvaluations: 2000

        Default properties:
                  CheckGradients: 0
                        Display: 'final'
        FiniteDifferenceStepSize: 'sqrt(eps)'
            FiniteDifferenceType: 'forward'
              FunctionTolerance: 1.0000e-06
                  MaxIterations: 400
                      OutputFcn: []
                        PlotFcn: []
        SpecifyObjectiveGradient: 0
                  StepTolerance: 1.0000e-06
                        TypicalX: 'ones(numberOfVariables,1)'
                    UseParallel: 0

     Show options not used by current Algorithm ('levenberg-marquardt')
```

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',...
    'MaxFunctionEvaluations',1500)

options =
  lsqnonlin options:

   Options used by current Algorithm ('levenberg-marquardt'):
   (Other available algorithms: 'trust-region-reflective')

   Set properties:
                  Algorithm: 'levenberg-marquardt'
        MaxFunctionEvaluations: 1500

   Default properties:
              CheckGradients: 0
                    Display: 'final'
    FiniteDifferenceStepSize: 'sqrt(eps)'
        FiniteDifferenceType: 'forward'
          FunctionTolerance: 1.0000e-06
              MaxIterations: 400
                  OutputFcn: []
                    PlotFcn: []
    SpecifyObjectiveGradient: 0
              StepTolerance: 1.0000e-06
```

```
                    TypicalX: 'ones(numberOfVariables,1)'
                  UseParallel: 0

   Show options not used by current Algorithm ('levenberg-marquardt')
```

Increase `MaxFunctionEvaluations` to 2000 by using dot notation.

```
options.MaxFunctionEvaluations = 2000

options =
  lsqnonlin options:

   Options used by current Algorithm ('levenberg-marquardt'):
   (Other available algorithms: 'trust-region-reflective')

   Set properties:
                      Algorithm: 'levenberg-marquardt'
        MaxFunctionEvaluations: 2000

   Default properties:
                 CheckGradients: 0
                        Display: 'final'
     FiniteDifferenceStepSize: 'sqrt(eps)'
          FiniteDifferenceType: 'forward'
             FunctionTolerance: 1.0000e-06
                   MaxIterations: 400
                       OutputFcn: []
                         PlotFcn: []
      SpecifyObjectiveGradient: 0
                   StepTolerance: 1.0000e-06
                        TypicalX: 'ones(numberOfVariables,1)'
                     UseParallel: 0

   Show options not used by current Algorithm ('levenberg-marquardt')
```

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','MaxIterations',1500)

oldoptions =
  fmincon options:

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

   Set properties:
                      Algorithm: 'sqp'
                   MaxIterations: 1500

   Default properties:
```

```
                CheckGradients: 0
            ConstraintTolerance: 1.0000e-06
                        Display: 'final'
      FiniteDifferenceStepSize: 'sqrt(eps)'
          FiniteDifferenceType: 'forward'
        MaxFunctionEvaluations: '100*numberOfVariables'
                 ObjectiveLimit: -1.0000e+20
            OptimalityTolerance: 1.0000e-06
                      OutputFcn: []
                        PlotFcn: []
                   ScaleProblem: 0
     SpecifyConstraintGradient: 0
      SpecifyObjectiveGradient: 0
                  StepTolerance: 1.0000e-06
                       TypicalX: 'ones(numberOfVariables,1)'
                    UseParallel: 0

   Show options not used by current Algorithm ('sqp')
```

Transfer the applicable options to the `fminunc` solver.

```
options = optimoptions(@fminunc,oldoptions)

options =
  fminunc options:

   Options used by current Algorithm ('quasi-newton'):
   (Other available algorithms: 'trust-region')

   Set properties:
                CheckGradients: 0
          FiniteDifferenceType: 'forward'
                 MaxIterations: 1500
           OptimalityTolerance: 1.0000e-06
                       PlotFcn: []
      SpecifyObjectiveGradient: 0
                 StepTolerance: 1.0000e-06

   Default properties:
                      Algorithm: 'quasi-newton'
                        Display: 'final'
      FiniteDifferenceStepSize: 'sqrt(eps)'
        MaxFunctionEvaluations: '100*numberOfVariables'
                 ObjectiveLimit: -1.0000e+20
                      OutputFcn: []
                       TypicalX: 'ones(numberOfVariables,1)'
                    UseParallel: 0

   Show options not used by current Algorithm ('quasi-newton')
```

Find Solver and Default Options for Optimization Problem

Create an optimization problem and find the default solver and options.

```
rng default
x = optimvar('x',3,'LowerBound',0);
expr = x'*(eye(3) + randn(3))*x - randn(1,3)*x;
prob = optimproblem('Objective',expr);
options = optimoptions(prob)

options =
  quadprog options:

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

   Set properties:
     No options set.

   Default properties:
              Algorithm: 'interior-point-convex'
    ConstraintTolerance: 1.0000e-08
                Display: 'final'
           LinearSolver: 'auto'
          MaxIterations: 200
     OptimalityTolerance: 1.0000e-08
          StepTolerance: 1.0000e-12

   Show options not used by current Algorithm ('interior-point-convex')
```

The default solver is `quadprog`.

Set the options to use iterative display. Find the solution.

```
options.Display = 'iter';
sol = solve(prob,'Options',options);

Solving problem using quadprog.
Your Hessian is not symmetric. Resetting H=(H+H')/2.

 Iter            Fval  Primal Infeas   Dual Infeas  Complementarity
    0    2.018911e+00   0.000000e+00  2.757660e+00     6.535839e-01
    1   -2.170204e+00   0.000000e+00  8.881784e-16     2.586177e-01
    2   -3.405808e+00   0.000000e+00  8.881784e-16     2.244054e-03
    3   -3.438788e+00   0.000000e+00  1.072059e-15     7.261144e-09

Minimum found that satisfies the constraints.

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

```
sol.x

ans = 3×1

    1.6035
    0.0000
    0.8029
```

Input Arguments

SolverName — Solver name
character vector | string | function handle

Solver name, specified as a character vector, string, or function handle.

Example: `'fmincon'`

Example: `@fmincon`

Data Types: `char` | `function_handle` | `string`

oldoptions — Options
options created using `optimoptions`

Options, specified as an options object. The `optimoptions` function creates options objects.

Example: `oldoptions = optimoptions(@fminunc)`

prob — Problem object
`OptimizationProblem` object | `EquationProblem` object

Problem object, specified as an `OptimizationProblem` object or an `EquationProblem` object. Create `prob` using the "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

The syntaxes using `prob` enable you to see what the default solver is for your problem and to modify the algorithm or other options.

Example: `prob = optimproblem('Objective',myobj)`, where `myobj` is an optimization expression

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `optimoptions(@fmincon,'Display','iter','FunctionTolerance',1e-10)` sets `fmincon` options to have iterative display, and to have a `FunctionTolerance` 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 on page 16-161
- `fsolve` options
- `ga` options (in Global Optimization Toolbox)
- `gamultiobj` options (in Global Optimization Toolbox)
- `intlinprog` options

- `linprog` options
- `lsqcurvefit` options
- `lsqlin` options
- `lsqnonlin` options
- `paretosearch` options (in Global Optimization Toolbox)
- `particleswarm` options (in Global Optimization Toolbox)
- `patternsearch` options (in Global Optimization Toolbox)
- `quadprog` options on page 16-421
- `simulannealbnd` options (in Global Optimization Toolbox)
- `surrogateopt` options (in Global Optimization Toolbox)

Output Arguments

`options` — Options object
options object

Options object, returned as the options for the `SolverName` solver.

Alternative Functionality

Live Editor Task

The **Optimize** Live Editor task lets you set options visually. For an example, see "Optimize Live Editor Task with fmincon Solver" on page 6-77.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation supports a limited set of options for each solver. For the supported options, see each solver reference page:

- `fmincon` "Code Generation" on page 16-105
- `fsolve` "Code Generation" on page 16-182
- `lsqcurvefit` "Code Generation" on page 16-254
- `lsqnonlin` "Code Generation" on page 16-291
- `quadprog` "Code Generation" on page 16-427

See Also
EquationProblem | OptimizationProblem | **Optimize** | optimset | resetoptions

Topics
"Set Options"

Introduced in R2013a

optimproblem

Create optimization problem

Syntax

```
prob = optimproblem
prob = optimproblem(Name,Value)
```

Description

Use `optimproblem` to create an optimization problem.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2.

`prob = optimproblem` creates an optimization problem with default properties.

`prob = optimproblem(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, to specify a maximization problem instead of a minimization problem, use `prob = optimproblem('ObjectiveSense','maximize')`.

Examples

Create Optimization Problem

Create an optimization problem with default properties.

```
prob = optimproblem

prob =
  OptimizationProblem with properties:

      Description: ''
   ObjectiveSense: 'minimize'
        Variables: [0x0 struct] containing 0 OptimizationVariables
        Objective: [0x0 OptimizationExpression]
      Constraints: [0x0 struct] containing 0 OptimizationConstraints

  No problem defined.
```

Create and Solve Maximization Problem

Create a linear programming problem for maximization. The problem has two positive variables and three linear inequality constraints.

```
prob = optimproblem('ObjectiveSense','max');
```

Create positive variables. Include an objective function in the problem.

```
x = optimvar('x',2,1,'LowerBound',0);
prob.Objective = x(1) + 2*x(2);
```

Create linear inequality constraints in the problem.

```
cons1 = x(1) + 5*x(2) <= 100;
cons2 = x(1) + x(2) <= 40;
cons3 = 2*x(1) + x(2)/2 <= 60;
prob.Constraints.cons1 = cons1;
prob.Constraints.cons2 = cons2;
prob.Constraints.cons3 = cons3;
```

Review the problem.

```
show(prob)
```

```
  OptimizationProblem :

    Solve for:
       x

    maximize :
       x(1) + 2*x(2)


    subject to cons1:
       x(1) + 5*x(2) <= 100

    subject to cons2:
       x(1) + x(2) <= 40

    subject to cons3:
       2*x(1) + 0.5*x(2) <= 60

    variable bounds:
       0 <= x(1)
       0 <= x(2)
```

Solve the problem.

```
sol = solve(prob);
```

```
Solving problem using linprog.
```

```
Optimal solution found.
```

```
sol.x
```

```
ans = 2×1

    25.0000
    15.0000
```

Input Arguments

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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: To specify a maximization problem, use `prob = optimproblem('ObjectiveSense', 'maximize')`.

`Constraints` — Problem constraints
`OptimizationConstraint` array | structure with `OptimizationConstraint` arrays as fields

Problem constraints, specified as an `OptimizationConstraint` array or a structure with `OptimizationConstraint` arrays as fields.

Example: `prob = optimproblem('Constraints', sum(x,2) == 1)`

`Description` — Problem label
`''` (default) | string | character vector

Problem label, specified as a string or character vector. The software does not use `Description` for computation. `Description` is an arbitrary label that you can use for any reason. For example, you can share, archive, or present a model or problem, and store descriptive information about the model or problem in `Description`.

Example: `"An iterative approach to the Traveling Salesman problem"`

Data Types: `char` | `string`

`Objective` — Objective function
scalar `OptimizationExpression`

Objective function, specified as a scalar `OptimizationExpression` object.

Example: `prob = optimproblem('Objective', sum(sum(x)))` for a 2-D variable x

`ObjectiveSense` — Sense of optimization
`'minimize'` (default) | `'min'` | `'maximize'` | `'max'`

Sense of optimization, specified as `'minimize'` or `'maximize'`. You can also specify `'min'` to obtain `'minimize'` or `'max'` to obtain `'maximize'`. The `solve` function minimizes the objective when `ObjectiveSense` is `'minimize'` and maximizes the objective when `ObjectiveSense` is `'maximize'`.

Example: `prob = optimproblem('ObjectiveSense', 'max')`

Data Types: `char` | `string`

Output Arguments

`prob` — Optimization problem
`OptimizationProblem` object

Optimization problem, returned as an `OptimizationProblem` object. Typically, to complete the problem description, you specify an objective function and constraints. However, you can have a

feasibility problem, which has no objective function, or you can have a problem with no constraints. Solve a complete problem by calling `solve`.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

See Also
`OptimizationProblem` | `optimvar` | `solve`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

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

Note optimoptions is recommended instead of optimset for all solvers except fzero, fminbnd, fminsearch, and lsqnonneg.

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 15-6 provides descriptions of optimization options and which functions use them. optimset uses different names for some options than optimoptions. See "Current and Legacy Option Names" on page 15-22.

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 `MaxIterations` option in the `fmincon` solver is 400 for the trust-region-reflective algorithm, but is 1000 for the interior-point algorithm.

Examples

This statement creates an optimization options structure called `options` in which the `Display` option is set to `'iter'` and the `TolX` option is set to `1e-8`.

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support the syntax that has no input or output arguments:

  ```
  optimset
  ```
- Functions specified in the `options` must be supported for code generation.
- The input argument `optimfun` must be a function that is supported for code generation.
- The fields of the options structure `oldopts` must be fixed-size fields.
- Code generation ignores the `Display` option.
- Code generation does not support the additional options in an options structure created by the Optimization Toolbox `optimset` function. If an input options structure includes the additional Optimization Toolbox options, then the output structure does not include them.

See Also
optimget | optimoptions | optimtool

Introduced before R2006a

optimtool

(To be removed) Select solver and optimization options, run problems

Note `optimtool` will be removed in a future release. Use **Optimize** or the problem-based approach instead.

Syntax

```
optimtool
optimtool(optstruct)
optimtool('solver')
```

Description

Optimization app for selecting solver and options.

Note The Optimization app warns that it will be removed in a future release. Use **Optimize** or the problem-based approach instead.

`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 `coneprog`, and any Global Optimization Toolbox solver except `particleswarm`, `surrogateopt`, `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 character vector, 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 `coneprog`, `GlobalSearch`, `intlinprog`, `MultiStart`, `particleswarm`, and `surrogateopt`.
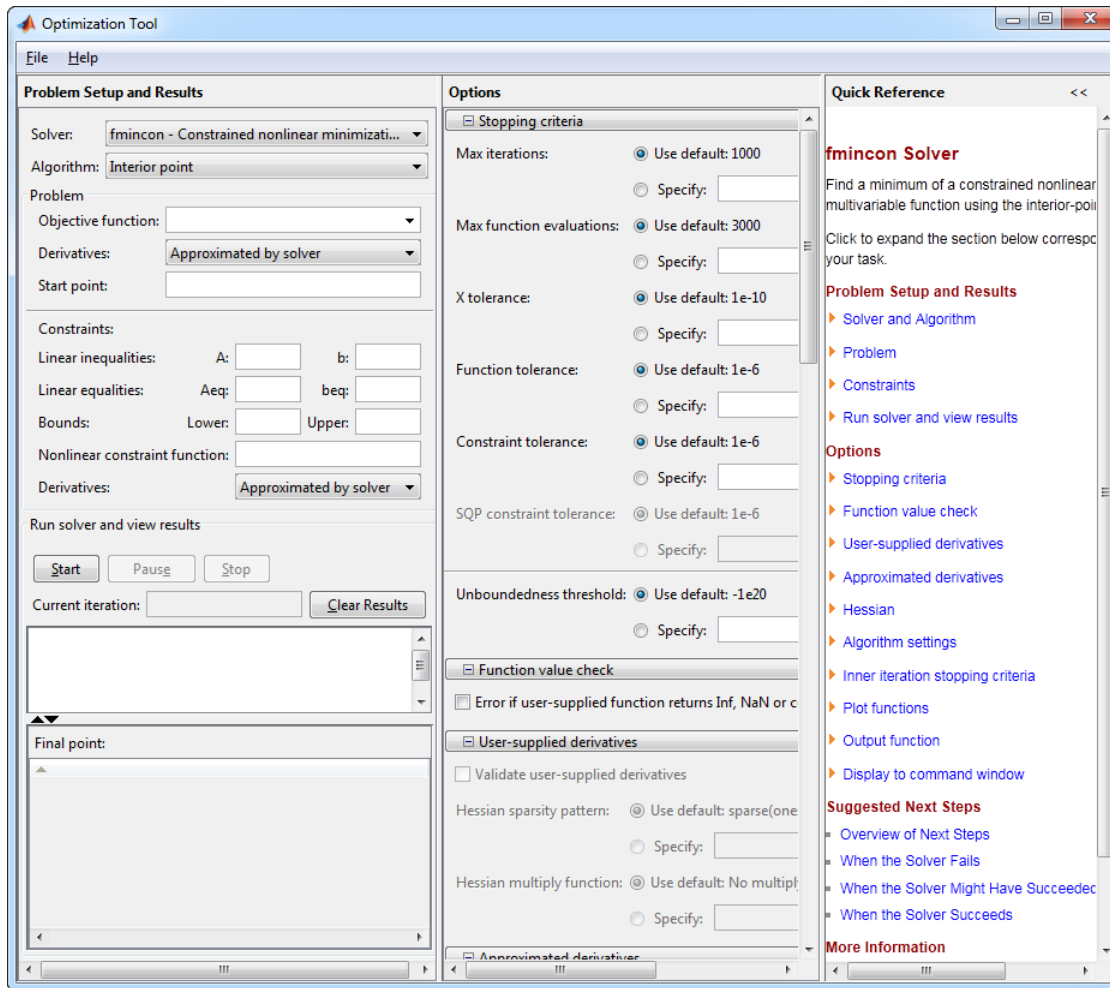
Compatibility Considerations

optimtool will be removed
Warns starting in R2015b

optimtool will be removed in a future release. Use **Optimize** or the problem-based approach instead.

See Also
Optimize | optimoptions | optimset

Topics
"Optimization App" on page 5-2

Introduced in R2006b

optimvar

Create optimization variables

Syntax

```
x = optimvar(name)

x = optimvar(name,n)

x = optimvar(name,cstr)
x = optimvar(name,cstr1,n2,...,cstrk)
x = optimvar(name,{cstr1,cstr2,...,cstrk})
x = optimvar(name,[n1,n2,...,nk])

x = optimvar( ___ ,Name,Value)
```

Description

Use `optimvar` to create optimization variables.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`x = optimvar(name)` creates a scalar optimization variable. An optimization variable is a symbolic object that enables you to create expressions for the objective function and the problem constraints in terms of the variable.

Tip To avoid confusion, set `name` to be the MATLAB variable name. For example,

```
metal = optimvar('metal')
```

`x = optimvar(name,n)` creates an n-by-1 vector of optimization variables.

`x = optimvar(name,cstr)` creates a vector of optimization variables that can use `cstr` for indexing. The number of elements of `x` is the same as the length of the `cstr` vector. The orientation of `x` is the same as the orientation of `cstr`: `x` is a row vector when `cstr` is a row vector, and `x` is a column vector when `cstr` is a column vector.

`x = optimvar(name,cstr1,n2,...,cstrk)` or `x = optimvar(name,{cstr1,cstr2,...,cstrk})` or `x = optimvar(name,[n1,n2,...,nk])`, for any combination of positive integers nj and names `cstrk`, creates an array of optimization variables with dimensions equal to the integers nj and the lengths of the entries cstr1k.

`x = optimvar( ___ ,Name,Value)`, for any previous syntax, uses additional options specified by one or more `Name,Value` pair arguments. For example, to specify an integer variable, use `x = optimvar('x','Type','integer')`.

Examples

Create Scalar Optimization Variable

Create a scalar optimization variable named `dollars`.

```
dollars = optimvar('dollars')

dollars =
  OptimizationVariable with properties:

          Name: 'dollars'
          Type: 'continuous'
     IndexNames: {{}  {}}
     LowerBound: -Inf
     UpperBound: Inf

  See variables with show.
  See bounds with showbounds.
```

Create Optimization Variable Vector

Create a 3-by-1 optimization variable vector named `x`.

```
x = optimvar('x',3)

x =
  3x1 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'x'
          Type: 'continuous'
     IndexNames: {{}  {}}

  Elementwise properties:
     LowerBound: [3x1 double]
     UpperBound: [3x1 double]

  See variables with show.
  See bounds with showbounds.
```

Create Optimization Variables Indexed by Strings

Create an integer optimization variable vector named `bolts` that is indexed by the strings `"brass"`, `"stainless"`, and `"galvanized"`. Use the indices of `bolts` to create an optimization expression, and experiment with creating `bolts` using character arrays or in a different orientation.

Create `bolts` using strings in a row orientation.

```
bnames = ["brass","stainless","galvanized"];
bolts = optimvar('bolts',bnames,'Type','integer')

bolts =
  1x3 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'bolts'
          Type: 'integer'
     IndexNames: {{}  {1x3 cell}}

  Elementwise properties:
    LowerBound: [-Inf -Inf -Inf]
    UpperBound: [Inf Inf Inf]

  See variables with show.
  See bounds with showbounds.
```

Create an optimization expression using the string indices.

```
y = bolts("brass") + 2*bolts("stainless") + 4*bolts("galvanized")

y =
  Linear OptimizationExpression

    bolts('brass') + 2*bolts('stainless') + 4*bolts('galvanized')
```

Use a cell array of character vectors instead of strings to get a variable with the same indices as before.

```
bnames = {'brass','stainless','galvanized'};
bolts = optimvar('bolts',bnames,'Type','integer')

bolts =
  1x3 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'bolts'
          Type: 'integer'
     IndexNames: {{}  {1x3 cell}}

  Elementwise properties:
    LowerBound: [-Inf -Inf -Inf]
    UpperBound: [Inf Inf Inf]

  See variables with show.
  See bounds with showbounds.
```

Use a column-oriented version of bnames, 3-by-1 instead of 1-by-3, and observe that bolts has that orientation as well.

```
bnames = ["brass";"stainless";"galvanized"];
bolts = optimvar('bolts',bnames,'Type','integer')

bolts =
  3x1 OptimizationVariable array with properties:
```

```
   Array-wide properties:
          Name: 'bolts'
          Type: 'integer'
     IndexNames: {{1x3 cell}  {}}

   Elementwise properties:
     LowerBound: [3x1 double]
     UpperBound: [3x1 double]

   See variables with show.
   See bounds with showbounds.
```

Create Multidimensional Optimization Variables

Create a 3-by-4-by-2 array of optimization variables named xarray.

```
xarray = optimvar('xarray',3,4,2)

xarray =
  3x4x2 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'xarray'
          Type: 'continuous'
     IndexNames: {{}  {}  {}}

  Elementwise properties:
     LowerBound: [3x4x2 double]
     UpperBound: [3x4x2 double]

  See variables with show.
  See bounds with showbounds.
```

You can also create multidimensional variables indexed by a mixture of names and numeric indices. For example, create a 3-by-4 array of optimization variables where the first dimension is indexed by the strings 'brass', 'stainless', and 'galvanized', and the second dimension is numerically indexed.

```
bnames = ["brass","stainless","galvanized"];
bolts = optimvar('bolts',bnames,4)

bolts =
  3x4 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'bolts'
          Type: 'continuous'
     IndexNames: {{1x3 cell}  {}}

  Elementwise properties:
     LowerBound: [3x4 double]
     UpperBound: [3x4 double]
```

```
  See variables with show.
  See bounds with showbounds.
```

Create Binary Optimization Variables

Create an optimization variable named x of size 3-by-3-by-3 that represents binary variables.

```
x = optimvar('x',3,3,3,'Type','integer','LowerBound',0,'UpperBound',1)
```

```
x =
  3x3x3 OptimizationVariable array with properties:

  Array-wide properties:
          Name: 'x'
          Type: 'integer'
     IndexNames: {{}  {}  {}}

  Elementwise properties:
    LowerBound: [3x3x3 double]
    UpperBound: [3x3x3 double]

  See variables with show.
  See bounds with showbounds.
```

Input Arguments

name — Variable name
character vector | string

Variable name, specified as a character vector or string.

Tip To avoid confusion about which name relates to which aspect of a variable, set the workspace variable name to the variable name. For example,

```
truck = optimvar('truck');
```

Example: "Warehouse"

Example: 'truck'

Data Types: char | string

n — Variable dimension
positive integer

Variable dimension, specified as a positive integer.

Example: 4

Data Types: `double`

`cstr` — Index names
string array | cell array of character arrays

Index names, specified as a string array or a cell array of character arrays.

Example: `x = optimvar('x',["Warehouse","Truck","City"])`

Example: `x = optimvar('x',{'Warehouse','Truck','City'})`

Data Types: `string` | `cell`

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: Create x as a 3-element nonnegative vector with `x(2) <= 2` and `x(3) <= 4` by the command `x = optimvar('x',3,'LowerBound',0,'UpperBound',[Inf,2,4])`

Type — Variable type
`'continuous'` (default) | `'integer'`

Variable type, specified as `'continuous'` or `'integer'`.

- `'continuous'` – Real values
- `'integer'` – Integer values

The variable type applies to all variables in the array. To have multiple variable types, create multiple variables.

Tip To specify binary variables, use the `'integer'` type with `LowerBound` equal to `0` and `UpperBound` equal to `1`.

Example: `'integer'`

LowerBound — Lower bounds
`-Inf` (default) | array of the same size as `x` | real scalar

Lower bounds, specified as an array of the same size as `x` or as a real scalar. If `LowerBound` is a scalar, the value applies to all elements of `x`.

Example: To set a lower bound of `0` to all elements of `x`, specify the scalar value `0`.

Data Types: `double`

UpperBound — Upper bounds
`Inf` (default) | array of the same size as `x` | real scalar

Upper bounds, specified as an array of the same size as `x` or as a real scalar. If `UpperBound` is a scalar, the value applies to all elements of `x`.

Example: To set an upper bound of `2` to all elements of `x`, specify the scalar value `2`.

Data Types: `double`

Output Arguments

x — Optimization variable
`OptimizationVariable` array

Optimization variable, returned as an `OptimizationVariable` array. The dimensions of the array are the same as those of the corresponding input variables, such as `cstr1`-by-`cstr2`.

Tips

- `OptimizationVariable` objects have *handle* copy behavior. See "Handle Object Behavior" and "Comparison of Handle and Value Classes". Handle copy behavior means that a copy of an `OptimizationVariable` points to the original and does not have an independent existence. For example, create a variable x, copy it to y, then set a property of y. Note that x takes on the new property value.

  ```
  x = optimvar('x','LowerBound',1);
  y = x;
  y.LowerBound = 0;
  showbounds(x)
  ```

  ```
      0 <= x
  ```

See Also
`OptimizationVariable`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2
"Named Index for Optimization Variables" on page 10-20

Introduced in R2017b

prob2struct

Package: `optim.problemdef`

Convert optimization problem or equation problem to solver form

Syntax

```
problem = prob2struct(prob)
problem = prob2struct(prob,x0)
problem = prob2struct( ___ ,Name,Value)
```

Description

Use `prob2struct` to convert an optimization problem or equation problem to solver form.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`problem = prob2struct(prob)` returns an optimization problem structure suitable for a solver-based solution. For nonlinear problems, `prob2struct` creates files for the objective function, and, if necessary, for nonlinear constraint functions and supporting files.

`problem = prob2struct(prob,x0)` also converts the initial point structure `x0` and includes it in `problem`.

`problem = prob2struct( ___ ,Name,Value)`, for any input arguments, specifies additional options using one or more name-value pair arguments. For example, for a nonlinear optimization problem, `problem = prob2struct(prob,'ObjectiveFunctionName','objfun1')` specifies that `prob2struct` creates an objective function file named `objfun1.m` in the current folder.

Examples

Convert Problem to Structure

Convert an optimization problem object to a problem structure.

Input the basic MILP problem from "Mixed-Integer Linear Programming Basics: Problem-Based" on page 10-47.

```
ingots = optimvar('ingots',4,1,'Type','integer','LowerBound',0,'UpperBound',1);
alloys = optimvar('alloys',4,1,'LowerBound',0);

weightIngots = [5,3,4,6];
costIngots = weightIngots.*[350,330,310,280];
costAlloys = [500,450,400,100];
cost = costIngots*ingots + costAlloys*alloys;

steelprob = optimproblem;
```

```
steelprob.Objective = cost;

totalweight = weightIngots*ingots + sum(alloys);

carbonIngots = [5,4,5,3]/100;
molybIngots = [3,3,4,4,]/100;
carbonAlloys = [8,7,6,3]/100;
molybAlloys = [6,7,8,9]/100;

totalCarbon = (weightIngots.*carbonIngots)*ingots + carbonAlloys*alloys;
totalMolyb = (weightIngots.*molybIngots)*ingots + molybAlloys*alloys;

steelprob.Constraints.conswt = totalweight == 25;
steelprob.Constraints.conscarb = totalCarbon == 1.25;
steelprob.Constraints.consmolyb = totalMolyb == 1.25;
```

Convert the problem to an `intlinprog` problem structure.

```
problem = prob2struct(steelprob);
```

Examine the resulting linear equality constraint matrix and vector.

```
Aeq = problem.Aeq

Aeq =
    (1,1)        1.0000
    (2,1)        0.0800
    (3,1)        0.0600
    (1,2)        1.0000
    (2,2)        0.0700
    (3,2)        0.0700
    (1,3)        1.0000
    (2,3)        0.0600
    (3,3)        0.0800
    (1,4)        1.0000
    (2,4)        0.0300
    (3,4)        0.0900
    (1,5)        5.0000
    (2,5)        0.2500
    (3,5)        0.1500
    (1,6)        3.0000
    (2,6)        0.1200
    (3,6)        0.0900
    (1,7)        4.0000
    (2,7)        0.2000
    (3,7)        0.1600
    (1,8)        6.0000
    (2,8)        0.1800
    (3,8)        0.2400


beq = problem.beq

beq = 3×1

   25.0000
    1.2500
    1.2500
```

Examine the bounds.

```
problem.lb
```

ans = *8×1*

```
     0
     0
     0
     0
     0
     0
     0
     0
```

```
problem.ub
```

ans = *8×1*

```
   Inf
   Inf
   Inf
   Inf
     1
     1
     1
     1
```

Solve the problem by calling `intlinprog`.

```
x = intlinprog(problem)
```

```
LP:                Optimal objective value is 8125.600000.

Cut Generation:    Applied 3 mir cuts.
                   Lower bound is 8495.000000.
                   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.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).
```

x = *8×1*

```
   7.2500
        0
   0.2500
   3.5000
   1.0000
   1.0000
        0
   1.0000
```

Convert Nonlinear Problem to Structure

Create a nonlinear problem in the problem-based framework.

```
x = optimvar('x',2);
fun = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
prob = optimproblem('Objective',fun);
mycon = dot(x,x) <= 4;
prob.Constraints.mycon = mycon;
x0.x = [-1;1.5];
```

Convert `prob` to an optimization problem structure. Name the generated objective function file `'rosenbrock'` and the constraint function file `'circle2'`.

```
problem = prob2struct(prob,x0,'ObjectiveFunctionName','rosenbrock',...
    'ConstraintFunctionName','circle2');
```

`prob2struct` creates nonlinear objective and constraint function files in the current folder. To create these files in a different folder, use the `'FileLocation'` name-value pair.

Solve the problem.

```
[x,fval] = fmincon(problem)
```

```
Local minimum found that satisfies the constraints.

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

x = 2×1

    1.0000
    1.0000


fval = 4.6238e-11
```

Input Arguments

prob — Optimization problem or equation problem
OptimizationProblem object | EquationProblem object

Optimization problem or equation problem, specified as an `OptimizationProblem` object or an `EquationProblem` object. Create an optimization problem by using `optimproblem`; create an equation problem by using `eqnproblem`.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

Example: `prob = optimproblem; prob.Objective = obj; prob.Constraints.cons1 = cons1;`

Example: `prob = eqnproblem; prob.Equations = eqs;`

x0 — Initial point
structure

Initial point, specified as a structure with field names equal to the variable names in `prob`.

For an example using `x0` with named index variables, see "Create Initial Point for Optimization with Named Index Variables" on page 10-50.

Example: If `prob` has variables named x and y: `x0.x = [3,2,17]; x0.y = [pi/3,2*pi/3]`.

Data Types: `struct`

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `problem = prob2struct(prob,'FileLocation','C:\Documents\myproblem')`

`ConstraintDerivative` — Indication to use automatic differentiation for constraint functions
`'auto'` (default) | `'finite-differences'`

Indication to use automatic differentiation (AD) for nonlinear constraint functions, specified as the comma-separated pair consisting of `'ConstraintDerivative'` and either `'auto'` (use AD if possible) or `'finite-differences'` (do not use AD). `'auto'` causes the resulting constraint function file to include derivative information provided that the constraint functions are supported, as described in "Supported Operations on Optimization Variables and Expressions" on page 10-43. For an example, see "Supply Derivatives in Problem-Based Workflow" on page 7-26.

Note To use automatic derivatives in a problem converted by `prob2struct`, pass options specifying these derivatives.

```
options = optimoptions('fmincon','SpecifyObjectiveGradient',true,...
    'SpecifyConstraintGradient',true);
problem.options = options;
```

Warning The `'ObjectiveDerivative'` and `'ConstraintDerivative'` name-value pair arguments currently apply only to problems solved by `fmincon` or `fminunc`. If you attempt to use these arguments when solving an equation with `solve` or converting an equation with `prob2struct`, MATLAB issues an error.

Example: `'finite-differences'`

Data Types: `char` | `string`

`ConstraintFunctionName` — Name of nonlinear constraint function file
`'generatedConstraints'` (default) | file name

Name of the nonlinear constraint function file created by `prob2struct` for an optimization problem, specified as the comma-separated pair consisting of `'ConstraintFunctionName'` and a file name. This argument applies to `fmincon` or `fminunc` problems; see `problem`. Do not include the file extension `.m` in the file name. `prob2struct` appends the file extension when it creates the file.

If you do not specify `ConstraintFunctionName`, then `prob2struct` overwrites `'generatedConstraints.m'`. If you do not specify `FileLocation`, then `prob2struct` creates the file in the current folder.

The returned `problem` structure refers to this function file.

Example: `"mynlcons"`

Data Types: `char` | `string`

EquationFunctionName — Name of equation function file
`'generatedEquation'` (default) | file name

Name of the nonlinear equation function file created by `prob2struct` for an equation problem, specified as the comma-separated pair consisting of `'EquationFunctionName'` and a file name. This argument applies to `fsolve`, `fzero`, or `lsqnonlin` equations; see `problem`. Do not include the file extension `.m` in the file name. `prob2struct` appends the file extension when it creates the file.

If you do not specify `EquationFunctionName`, then `prob2struct` overwrites `'generatedEquation.m'`. If you do not specify `FileLocation`, then `prob2struct` creates the file in the current folder.

The returned `problem` structure refers to this function file.

Example: `"myequation"`

Data Types: `char` | `string`

FileLocation — Location for generated files
current folder (default) | path to a writable folder

Location for generated files (objective function, constraint function, and other subfunction files), specified as the comma-separated pair consisting of `'FileLocation'` and a path to a writable folder. All the generated files are stored in this folder; multiple folders are not supported.

Example: `'C:Documents\MATLAB\myproject'`

Data Types: `char` | `string`

ObjectiveDerivative — Indication to use automatic differentiation for objective function
`'auto'` (default) | `'finite-differences'`

Indication to use automatic differentiation (AD) for a nonlinear objective function, specified as the comma-separated pair consisting of `'ObjectiveDerivative'` and either `'auto'` (use AD if possible) or `'finite-differences'` (do not use AD). `'auto'` causes the resulting objective function file to include derivative information provided that the objective function is supported, as described in "Supported Operations on Optimization Variables and Expressions" on page 10-43. For an example, see "Supply Derivatives in Problem-Based Workflow" on page 7-26.

Note To use automatic derivatives in a problem converted by `prob2struct`, pass options specifying these derivatives.

```
options = optimoptions('fmincon','SpecifyObjectiveGradient',true,...
    'SpecifyConstraintGradient',true);
problem.options = options;
```

Warning The 'ObjectiveDerivative' and 'ConstraintDerivative' name-value pair arguments currently apply only to problems solved by fmincon or fminunc. If you attempt to use these arguments when solving an equation with solve or converting an equation with prob2struct, MATLAB issues an error.

Example: 'finite-differences'

Data Types: char | string

ObjectiveFunctionName — Name of objective function file
'generatedObjective' (default) | file name

Name of the objective function file created by prob2struct for an optimization problem, specified as the comma-separated pair consisting of 'ObjectiveFunctionName' and a file name. This argument applies to fmincon or fminunc problems; see problem. Do not include the file extension .m in the file name. prob2struct appends the file extension when it creates the file.

If you do not specify ObjectiveFunctionName, then prob2struct overwrites 'generatedObjective.m'. If you do not specify FileLocation, then prob2struct creates the file in the current folder.

The returned problem structure refers to this function file.

Example: "myobj"

Data Types: char | string

Output Arguments

problem — Problem structure
fmincon problem structure | fminunc problem structure | fsolve problem structure | intlinprog problem structure | linprog problem structure | lsqlin problem structure | lsqnonlin problem structure | quadprog problem structure

Problem structure, returned as an fmincon problem structure, fminunc problem structure, fsolve problem structure, intlinprog problem structure, linprog problem structure, lsqlin problem structure, lsqnonlin problem structure, or quadprog problem structure.

The following table gives the resulting problem type for optimization problems.

Optimization Objective and Constraint Types (Linear Constraints Include Bounds)	Resulting Problem Type
Linear objective and constraint functions. At least one problem variable has the 'integer' type.	intlinprog

Optimization Objective and Constraint Types (Linear Constraints Include Bounds)	Resulting Problem Type
Linear objective and constraint functions. No problem variable has the `'integer'` type.	`linprog`
Linear constraint functions. The objective function is a constant plus a sum of squares of linear expressions.	`lsqlin`
Bound constraints. The objective function is a constant plus a sum of squares of general nonlinear expressions.	`lsqnonlin`
Linear constraint functions. General quadratic objective function.	`quadprog`
General nonlinear objective function. No constraints.	`fminunc`
General nonlinear objective function, and there is at least one constraint of any type. Or, there is at least one general nonlinear constraint function.	`fmincon`

The following table gives the resulting problem type for equation solving problems.

Equation Types	Resulting Problem Type
Linear system with or without bounds	`lsqlin`
Scalar (single) nonlinear equation	`fzero`
Nonlinear system without constraints	`fsolve`
Nonlinear system with bounds	`lsqnonlin`

Note For nonlinear problems, `prob2struct` creates function files for the objective and nonlinear constraint functions. For objective and constraint functions that call supporting functions, `prob2struct` also creates supporting function files and stores them in the `FileLocation` folder. To access extra parameters in generated functions, see "Obtain Generated Function Details" on page 7-35.

For linear and quadratic optimization problems, the problem structure includes an additional field, `f0`, that represents an additive constant for the objective function. If you solve the problem structure using the specified solver, the returned objective function value does not include the `f0` value. If you solve `prob` using the `solve` function, the returned objective function value includes the `f0` value.

If the ObjectiveSense of `prob` is `'max'` or `'maximize'`, then `problem` uses the negative of the objective function in `prob` because solvers minimize. To maximize, they minimize the negative of the original objective function. In this case, the reported optimal function value from the solver is the negative of the value in the original problem. See "Maximizing an Objective" on page 2-30. You cannot use `lsqlin` for a maximization problem.

Tips

- If you call `prob2struct` multiple times in the same MATLAB session for nonlinear problems, use the `ObjectiveFunctionName` or `EquationFunctionName` argument and, if appropriate, the `ConstraintFunctionName` argument. Specifying unique names ensures that the resulting problem structures refer to the correct objective and constraint functions. Otherwise, subsequent calls to `prob2struct` can cause the generated nonlinear function files to overwrite existing files.

- To avoid causing an infinite recursion, do not call `prob2struct` inside an objective or constraint function.

- When calling `prob2struct` in parallel for nonlinear problems, ensure that the resulting objective and constraint function files have unique names. Doing so avoids each pass of the loop writing to the same file or files.

Algorithms

Conversion to Solver Form

The basis for the problem structure is an implicit ordering of all problem variables into a single vector. The order of the problem variables is the same as the order of the `Variables` property in `prob`. See `OptimizationProblem`. You can also find the order by using `varindex`.

For example, suppose that the problem variables are in this order:

- x — a 3-by-2-by-4 array
- y — a 3-by-2 array

In this case, the implicit variable order is the same as if the problem variable is `vars = [x(:);y(:)]`.

The first 24 elements of `vars` are equivalent to `x(:)`, and the next six elements are equivalent to `y(:)`, for a total of 30 elements. The lower and upper bounds correspond to this variable ordering, and each linear constraint matrix has 30 columns.

For problems with general nonlinear objective or constraint functions, `prob2struct` creates function files in the current folder or in the folder specified by `FileLocation`. The returned `problem` structure refers to these function files.

Automatic Differentiation

Automatic differentiation (AD) applies to the `solve` and `prob2struct` functions under the following conditions:

- The objective and constraint functions are supported, as described in "Supported Operations on Optimization Variables and Expressions" on page 10-43. They do not require use of the `fcn2optimexpr` function.

- The solver called by `solve` is either `fmincon` or `fminunc`.

- The `'ObjectiveDerivative'` and `'ConstraintDerivative'` name-value pair arguments for `solve` or `prob2struct` are set to their default value, `'auto'`.

When AD Applies	All Constraint Functions Supported	One or More Constraints Not Supported
Objective Function Supported	AD used for objective and constraints	AD used for objective only
Objective Function Not Supported	AD used for constraints only	AD not used

When these conditions are not satisfied, `solve` estimates gradients by finite differences, and `prob2struct` does not create gradients in its generated function files.

Note To use automatic derivatives in a problem converted by `prob2struct`, pass options specifying these derivatives.

```
options = optimoptions('fmincon','SpecifyObjectiveGradient',true,...
    'SpecifyConstraintGradient',true);
problem.options = options;
```

Currently, AD works only for first derivatives; it does not apply to second or higher derivatives. So, for example, if you want to use an analytic Hessian to speed your optimization, you cannot use `solve` directly, and must instead use the approach described in "Supply Derivatives in Problem-Based Workflow" on page 7-26.

Compatibility Considerations

`Options` Name-Value Pair Will Be Removed
Not recommended starting in R2020b

The `Options` name-value pair will be removed in a future release. To modify options, edit the resulting `problem` structure. For example,

```
problem.options = optimoptions('fmincon',...
    'Display','iter','MaxFunctionEvaluations',5e4);
% Or, to set just one option:
problem.options.MaxFunctionEvaluations = 5e4;
```

The `Options` name-value pair will be removed because it can cause ambiguity in the presence of automatic differentiation.

See Also
`EquationProblem` | `OptimizationProblem` | `varindex`

Topics
"Problem-Based Optimization Workflow" on page 10-2
"Supply Derivatives in Problem-Based Workflow" on page 7-26
"Obtain Generated Function Details" on page 7-35
"Output Function for Problem-Based Optimization" on page 7-38

Introduced in R2017b

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( ___ )
[x,fval,exitflag,output] = quadprog( ___ )
[x,fval,exitflag,output,lambda] = quadprog( ___ )
```

Description

Solver for quadratic objective functions with linear constraints.

quadprog 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.

You can pass *f*, *lb*, and *ub* as vectors or matrices; see "Matrix Arguments" on page 2-31.

Note quadprog applies only to the solver-based approach. For a discussion of the two optimization approaches, see "First Choose Problem-Based or Solver-Based Approach" on page 1-3.

---

x = quadprog(H,f) returns a vector x that minimizes 1/2*x'*H*x + f'*x. The input H must be positive definite for the problem to have a finite minimum. If H is positive definite, then the solution x = H\(-f).

x = quadprog(H,f,A,b) minimizes 1/2*x'*H*x + f'*x subject to the restrictions A*x ≤ b. The input 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 ≤ x ≤ ub. The inputs 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 ≤ x ≤ 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 x0.

Note x0 is a required argument for the 'active-set' algorithm.

---

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, a structure described in problem. Create the problem structure using dot notation or the struct function. Alternatively, create a problem structure from an OptimizationProblem object by using prob2struct.

[x,fval] = quadprog(___), for any input variables, also returns fval, the value of the objective function at x:

fval = 0.5*x'*H*x + f'*x

[x,fval,exitflag,output] = quadprog(___) also returns exitflag, an integer that describes the exit condition of quadprog, and output, a structure that contains information about the optimization.

[x,fval,exitflag,output,lambda] = quadprog(___) also returns lambda, a structure whose fields contain the Lagrange multipliers at the solution x.

Examples

Quadratic Program with Linear Constraints

Find the minimum of

$$f(x) = \frac{1}{2}x_1^2 + x_2^2 - x_1 x_2 - 2x_1 - 6x_2$$

subject to the constraints

$$x_1 + x_2 \leq 2$$
$$-x_1 + 2x_2 \leq 2$$
$$2x_1 + x_2 \leq 3\,.$$

In quadprog syntax, this problem is to minimize

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

where

$$H = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}$$

$$f = \begin{bmatrix} -2 \\ -6 \end{bmatrix},$$

subject to the linear constraints.

To solve this problem, first enter the coefficient matrices.

```
H = [1 -1; -1 2];
f = [-2; -6];
A = [1 1; -1 2; 2 1];
b = [2; 2; 3];
```

Call `quadprog`.

```
[x,fval,exitflag,output,lambda] = ...
    quadprog(H,f,A,b);
```

```
Minimum found that satisfies the constraints.

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

Examine the final point, function value, and exit flag.

```
x,fval,exitflag
```

```
x = 2×1

    0.6667
    1.3333


fval = -8.2222

exitflag = 1
```

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.

Confirm that H is positive definite by checking its eigenvalues.

```
eig(H)
```

```
ans = 2×1

    0.3820
    2.6180
```

Quadratic Program with Linear Equality Constraint

Find the minimum of

$$f(x) = \frac{1}{2}x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2$$

subject to the constraint

$$x_1 + x_2 = 0.$$

In `quadprog` syntax, this problem is to minimize

$$f(x) = \frac{1}{2}x^THx + f^Tx,$$

where

$$H = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}$$

$$f = \begin{bmatrix} -2 \\ -6 \end{bmatrix},$$

subject to the linear constraint.

To solve this problem, first enter the coefficient matrices.

```
H = [1 -1; -1 2];
f = [-2; -6];
Aeq = [1 1];
beq = 0;
```

Call `quadprog`, entering `[]` for the inputs A and b.

```
[x,fval,exitflag,output,lambda] = ...
    quadprog(H,f,[],[],Aeq,beq);
```

```
Minimum found that satisfies the constraints.

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

Examine the final point, function value, and exit flag.

```
x,fval,exitflag
```

```
x = 2×1

   -0.8000
    0.8000


fval = -1.6000

exitflag = 1
```

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.

Confirm that H is positive definite by checking its eigenvalues.

```
eig(H)
```

```
ans = 2×1

    0.3820
    2.6180
```

Quadratic Minimization with Linear Constraints and Bounds

Find the x that minimizes the quadratic expression

$$\frac{1}{2}x^THx + f^Tx$$

where

$$H = \begin{bmatrix} 1 & -1 & 1 \\ -1 & 2 & -2 \\ 1 & -2 & 4 \end{bmatrix}, f = \begin{bmatrix} 2 \\ -3 \\ 1 \end{bmatrix},$$

subject to the constraints

$0 \le x \le 1, \sum x = 1/2.$

To solve this problem, first enter the coefficients.

```
H = [1,-1,1
     -1,2,-2
     1,-2,4];
f = [2;-3;1];
lb = zeros(3,1);
ub = ones(size(lb));
Aeq = ones(1,3);
beq = 1/2;
```

Call `quadprog`, entering `[ ]` for the inputs A and b.

```
x = quadprog(H,f,[],[],Aeq,beq,lb,ub)
```

```
Minimum found that satisfies the constraints.

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

x = 3×1

    0.0000
```

```
    0.5000
    0.0000
```

Quadratic Minimization with Nondefault Options

Set options to monitor the progress of `quadprog`.

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

Define a problem with a quadratic objective and linear inequality constraints.

```
H = [1 -1; -1 2];
f = [-2; -6];
A = [1 1; -1 2; 2 1];
b = [2; 2; 3];
```

To help write the `quadprog` function call, set the unnecessary inputs to `[]`.

```
Aeq = [];
beq = [];
lb = [];
ub = [];
x0 = [];
```

Call `quadprog` to solve the problem.

```
x = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0,options)

 Iter            Fval  Primal Infeas    Dual Infeas  Complementarity
    0  -8.884885e+00   3.214286e+00   1.071429e-01     1.000000e+00
    1  -8.331868e+00   1.321041e-01   4.403472e-03     1.910489e-01
    2  -8.212804e+00   1.676295e-03   5.587652e-05     1.009601e-02
    3  -8.222204e+00   8.381476e-07   2.793826e-08     1.809485e-05
    4  -8.222222e+00   3.019807e-14   1.352696e-12     7.525735e-13

Minimum found that satisfies the constraints.

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

x = 2×1

    0.6667
    1.3333
```

Quadratic Problem from `prob2struct`

Create a `problem` structure using a "Problem-Based Optimization Workflow" on page 10-2. Create an optimization problem equivalent to "Quadratic Program with Linear Constraints" on page 16-409.

```
x = optimvar('x',2);
objec = x(1)^2/2 + x(2)^2 - x(1)*x(2) - 2*x(1) - 6*x(2);
prob = optimproblem('Objective',objec);
prob.Constraints.cons1 = sum(x) <= 2;
prob.Constraints.cons2 = -x(1) + 2*x(2) <= 2;
prob.Constraints.cons3 = 2*x(1) + x(2) <= 3;
```

Convert `prob` to a `problem` structure.

```
problem = prob2struct(prob);
```

Solve the problem using `quadprog`.

```
[x,fval] = quadprog(problem)
```

```
Warning: Your Hessian is not symmetric. Resetting H=(H+H')/2.

Minimum found that satisfies the constraints.

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

x = 2×1

    0.6667
    1.3333


fval = -8.2222
```

Return quadprog Objective Function Value

Solve a quadratic program and return both the solution and the objective function value.

```
H = [1,-1,1
    -1,2,-2
    1,-2,4];
f = [-7;-12;-15];
A = [1,1,1];
b = 3;
[x,fval] = quadprog(H,f,A,b)
```

```
Minimum found that satisfies the constraints.

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

x = 3×1

   -3.5714
    2.9286
    3.6429


fval = -47.1786
```

Check that the returned objective function value matches the value computed from the `quadprog` objective function definition.

```
fval2 = 1/2*x'*H*x + f'*x
```

```
fval2 = -47.1786
```

Examine quadprog Optimization Process

To see the optimization process for `quadprog`, set options to show an iterative display and return four outputs. The problem is to minimize

$$\frac{1}{2}x^T H x + f^T x$$

subject to

$0 \le x \le 1,$

where

$$H = \begin{bmatrix} 2 & 1 & -1 \\ 1 & 3 & \frac{1}{2} \\ -1 & \frac{1}{2} & 5 \end{bmatrix}, f = \begin{bmatrix} 4 \\ -7 \\ 12 \end{bmatrix}.$$

Enter the problem coefficients.

```
H = [2 1 -1
     1 3 1/2
     -1 1/2 5];
f = [4;-7;12];
lb = zeros(3,1);
ub = ones(3,1);
```

Set the options to display iterative progress of the solver.

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

Call `quadprog` with four outputs.

```
[x fval,exitflag,output] = quadprog(H,f,[],[],[],[],lb,ub,[],options)
```

```
 Iter            Fval  Primal Infeas    Dual Infeas  Complementarity
    0    2.691769e+01   1.582123e+00   1.712849e+01     1.680447e+00
    1   -3.889430e+00   0.000000e+00   8.564246e-03     9.971731e-01
    2   -5.451769e+00   0.000000e+00   4.282123e-06     2.710131e-02
    3   -5.499997e+00   0.000000e+00   1.221903e-10     6.939689e-07
    4   -5.500000e+00   0.000000e+00   5.842173e-14     3.469847e-10

Minimum found that satisfies the constraints.

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

```
x = 3×1

    0.0000
    1.0000
    0.0000


fval = -5.5000

exitflag = 1

output = struct with fields:
            message: '...'
          algorithm: 'interior-point-convex'
       firstorderopt: 1.5921e-09
      constrviolation: 0
           iterations: 4
         linearsolver: 'dense'
         cgiterations: []
```

Return quadprog Lagrange Multipliers

Solve a quadratic programming problem and return the Lagrange multipliers.

```
H = [1,-1,1
     -1,2,-2
     1,-2,4];
f = [-7;-12;-15];
A = [1,1,1];
b = 3;
lb = zeros(3,1);
[x,fval,exitflag,output,lambda] = quadprog(H,f,A,b,[],[],lb);

Minimum found that satisfies the constraints.

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

Examine the Lagrange multiplier structure `lambda`.

```
disp(lambda)

    ineqlin: 12.0000
      eqlin: [0x1 double]
      lower: [3x1 double]
      upper: [3x1 double]
```

The linear inequality constraint has an associated Lagrange multiplier of 12.

Display the multipliers associated with the lower bound.

```
disp(lambda.lower)
```

```
    5.0000
    0.0000
    0.0000
```

Only the first component of `lambda.lower` has a nonzero multiplier. This generally means that only the first component of `x` is at the lower bound of zero. Confirm by displaying the components of `x`.

`disp(x)`

```
    0.0000
    1.5000
    1.5000
```

Input Arguments

H — Quadratic objective term
symmetric real matrix

Quadratic objective term, specified as a symmetric real matrix. `H` represents the quadratic in the expression `1/2*x'*H*x + f'*x`. If `H` is not symmetric, `quadprog` issues a warning and uses the symmetrized version `(H + H')/2` instead.

If the quadratic matrix `H` is sparse, then by default, the `'interior-point-convex'` algorithm uses a slightly different algorithm than when `H` is dense. Generally, the sparse algorithm is faster on large, sparse problems, and the dense algorithm is faster on dense or small problems. For more information, see the `LinearSolver` option description and "interior-point-convex quadprog Algorithm" on page 11-2.

Example: `[2,1;1,3]`

Data Types: `double`

f — Linear objective term
real vector

Linear objective term, specified as a real vector. `f` represents the linear term in the expression `1/2*x'*H*x + f'*x`.

Example: `[1;3;2]`

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

$x_1 + 2x_2 \leq 10$
$3x_1 + 4x_2 \leq 20$
$5x_1 + 6x_2 \leq 30$,

enter 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, use `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, consider these inequalities:

$x_1 + 2x_2 \leq 10$
$3x_1 + 4x_2 \leq 20$
$5x_1 + 6x_2 \leq 30$.

Specify the inequalities by entering the following constraints.

```
A = [1,2;3,4;5,6];
b = [10;20;30];
```

Example: To specify that the x components sum to 1 or less, use `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

$x_1 + 2x_2 + 3x_3 = 10$

$2x_1 + 4x_2 + x_3 = 20,$

enter these constraints:

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use `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 `Me`-by-`N`.

For example, consider these equalities:

$x_1 + 2x_2 + 3x_3 = 10$
$2x_1 + 4x_2 + x_3 = 20.$

Specify the equalities by entering the following constraints.

```
Aeq = [1,2,3;2,4,1];
beq = [10;20];
```

Example: To specify that the x components sum to 1, use `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 the number of elements in `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)`.

If there are fewer elements in `lb` than in `x0`, solvers issue a warning.

Example: To specify that all x components are positive, use `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 the number of elements in 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).

If there are fewer elements in ub than in x0, solvers issue a warning.

Example: To specify that all x components are less than 1, use ub = ones(size(x0)).

Data Types: double

x0 — Initial point
real vector

Initial point, specified as a real vector. The length of x0 is the number of rows or columns of H.

x0 applies to the 'trust-region-reflective' algorithm when the problem has only bound constraints. x0 also applies to the 'active-set' algorithm.

Note x0 is a required argument for the 'active-set' algorithm.

If you do not specify 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.

Example: [1;2;1]

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 are absent from the optimoptions display. These options appear in italics in the following table. For details, see "View Options" on page 2-66.

All Algorithms

Algorithm	Choose the algorithm: • `'interior-point-convex'` (default) • `'trust-region-reflective'` • `'active-set'` The `'interior-point-convex'` algorithm handles only convex problems. The `'trust-region-reflective'` algorithm handles problems with only bounds or only linear equality constraints, but not both. The `'active-set'` algorithm handles indefinite problems provided that the projection of H onto the nullspace of Aeq is positive semidefinite. For details, see "Choosing the Algorithm" on page 2-6.
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 3-14): • `'off'` or `'none'` displays no output. • `'final'` displays only the final output (default). The `'interior-point-convex'` and `'active-set'` algorithms allow additional values: • `'iter'` specifies an iterative display. • `'iter-detailed'` specifies an iterative display with a detailed exit message. • `'final-detailed'` displays only the final output with a detailed exit message.
MaxIterations	Maximum number of iterations allowed; a positive integer. • For a `'trust-region-reflective'` equality-constrained problem, the default value is 2*(numberOfVariables – numberOfEqualities). • `'active-set'` has a default of 10*(numberOfVariables + numberOfConstraints). • For all other algorithms and problems, the default value is 200. For optimset, the option name is MaxIter. See "Current and Legacy Option Names" on page 15-22.

OptimalityTolerance   Termination tolerance on the first-order optimality; 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 the `'interior-point-convex'` and `'active-set'` algorithms, the default value is `1e-8`.

See "Tolerances and Stopping Criteria" on page 2-68.

For `optimset`, the option name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.

StepTolerance   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-12`.
- For `'active-set'`, the default value is `1e-8`.

For `optimset`, the option name is `TolX`. See "Current and Legacy Option Names" on page 15-22.

'trust-region-reflective' Algorithm Only

FunctionTolerance	Termination tolerance on the function value; a positive scalar. The default value depends on the problem type: bound-constrained problems use `100*eps`, and linear equality-constrained problems use `1e-6`. See "Tolerances and Stopping Criteria" on page 2-68. For `optimset`, the option name is `TolFun`. See "Current and Legacy Option Names" on page 15-22.
HessianMultiplyFcn	Hessian multiply function, specified as a function handle. 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 potentially some additional parameters) contain the matrices used to compute H*Y. See "Quadratic Minimization with Dense, Structured Hessian" on page 11-24 for an example that uses this option. For `optimset`, the option name is `HessMult`. See "Current and Legacy Option Names" on page 15-22.
MaxPCGIter	Maximum number of PCG (preconditioned conjugate gradient) iterations; a positive scalar. The default is `max(1,floor(numberOfVariables/2))` for bound-constrained problems. For equality-constrained problems, `quadprog` ignores `MaxPCGIter` and uses `MaxIterations` to limit the number of PCG iterations. For more information, see "Preconditioned Conjugate Gradient Method" on page 11-9.
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.
SubproblemAlgorithm	Determines how the iteration step is calculated. The default, `'cg'`, takes a faster but less accurate step than `'factorization'`. See "trust-region-reflective quadprog Algorithm" on page 11-7.
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

ConstraintTolerance	Tolerance on the constraint violation; a positive scalar. The default is `1e-8`. For `optimset`, the option name is `TolCon`. See "Current and Legacy Option Names" on page 15-22.
LinearSolver	Type of internal linear solver in the algorithm: • `'auto'` (default) — Use `'sparse'` if the H matrix is sparse and `'dense'` otherwise. • `'sparse'` — Use sparse linear algebra. See "Sparse Matrices". • `'dense'` — Use dense linear algebra.

`'active-set'` Algorithm Only

ConstraintTolerance	Tolerance on the constraint violation; a positive scalar. The default value is `1e-8`. For `optimset`, the option name is `TolCon`. See "Current and Legacy Option Names" on page 15-22.
ObjectiveLimit	A tolerance (stopping criterion) that is a scalar. If the objective function value goes below `ObjectiveLimit` and the current point is feasible, the iterations halt because the problem is unbounded, presumably. The default value is `-1e20`.

problem — Problem structure
structure

Problem structure, specified as a structure with these fields:

H	Symmetric matrix in `1/2*x'*H*x`
f	Vector in linear term `f'*x`
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
x0	Initial point for x
solver	`'quadprog'`
options	Options created using `optimoptions` or `optimset`

The required fields are H, f, `solver`, and `options`. When solving, `quadprog` ignores any fields in `problem` other than those listed.

Data Types: `struct`

Output Arguments

x — Solution
real vector

Solution, returned as a real vector. x is the 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 4-22.

fval — Objective function value at solution
real scalar

Objective function value at the solution, returned as a real scalar. fval is the value of 1/2*x'*H*x + f'*x at the solution x.

exitflag — Reason quadprog stopped
integer

Reason quadprog stopped, returned as an integer described in this table.

All Algorithms

1	Function converged to the solution x.
0	Number of iterations exceeded options.MaxIterations.
-2	Problem is infeasible. Or, for 'interior-point-convex', the step size was smaller than options.StepTolerance, but constraints were not satisfied.
-3	Problem is unbounded.

'interior-point-convex' Algorithm

2	Step size was smaller than options.StepTolerance, constraints were satisfied.
-6	Nonconvex problem detected.
-8	Unable to compute a step direction.

'trust-region-reflective' Algorithm

4	Local minimum found; minimum is not unique.
3	Change in the objective function value was smaller than options.FunctionTolerance.
-4	Current search direction was not a direction of descent. No further progress could be made.

'active-set' Algorithm

-6	Nonconvex problem detected; projection of H onto the nullspace of Aeq is not positive semidefinite.

Note Occasionally, the 'active-set' algorithm halts with exit flag 0 when the problem is, in fact, unbounded. Setting a higher iteration limit also results in exit flag 0.

output — Information about optimization process
structure

Information about the optimization process, returned as a structure with these fields:

`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
`linearsolver`	Type of internal linear solver, `'dense'` or `'sparse'` (`'interior-point-convex'` algorithm only)
`message`	Exit message

`lambda` — Lagrange multipliers at solution
structure

Lagrange multipliers at the solution, returned as a structure with these fields:

`lower`	Lower bounds `lb`
`upper`	Upper bounds `ub`
`ineqlin`	Linear inequalities
`eqlin`	Linear equalities

For details, see "Lagrange Multiplier Structures" on page 3-22.

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.

The algorithm has different implementations for a sparse Hessian matrix H and for a dense matrix. Generally, the sparse implementation is faster on large, sparse problems, and the dense implementation is faster on dense or small problems. For more information, see "interior-point-convex quadprog Algorithm" on page 11-2.

`'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 11-7.

`'active-set'`

The `'active-set'` algorithm is a projection method, similar to the one described in [2]. The algorithm is not large-scale; see "Large-Scale vs. Medium-Scale Algorithms" on page 2-10. For more information, see "active-set quadprog Algorithm" on page 11-11.

Alternative Functionality

App

The **Optimize** Live Editor task provides a visual interface for quadprog.

References

[1] Coleman, T. F., and Y. Li. "A Reflective Newton Method for Minimizing a Quadratic Function Subject to Bounds on Some of the Variables." *SIAM Journal on Optimization*. Vol. 6, Number 4, 1996, pp. 1040–1058.

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

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- quadprog supports code generation using either the codegen function or the MATLAB Coder app. You must have a MATLAB Coder license to generate code.
- The target hardware must support standard double-precision floating-point computations. You cannot generate code for single-precision or fixed-point computations.
- Code generation targets do not use the same math kernel libraries as MATLAB solvers. Therefore, code generation solutions can vary from solver solutions, especially for poorly conditioned problems.
- quadprog does not support the problem argument for code generation.

  ```
  [x,fval] = quadprog(problem) % Not supported
  ```
- All quadprog input matrices such as A, Aeq, lb, and ub must be full, not sparse. You can convert sparse matrices to full by using the full function.
- The lb and ub arguments must have the same number of entries as the number of columns in H or must be empty [].
- For advanced code optimization involving embedded processors, you also need an Embedded Coder license.
- You must include options for quadprog and specify them using optimoptions. The options must include the Algorithm option, set to 'active-set'.

  ```
  options = optimoptions('quadprog','Algorithm','active-set');
  [x,fval,exitflag] = quadprog(H,f,A,b,Aeq,beq,lb,ub,x0,options);
  ```
- Code generation supports these options:

  - Algorithm — Must be 'active-set'
  - ConstraintTolerance

- `MaxIterations`
- `ObjectiveLimit`
- `OptimalityTolerance`
- `StepTolerance`

- Generated code has limited error checking for options. The recommended way to update an option is to use `optimoptions`, not dot notation.

```matlab
opts = optimoptions('quadprog','Algorithm','active-set');
opts = optimoptions(opts,'MaxIterations',1e4); % Recommended
opts.MaxIterations = 1e4; % Not recommended
```

- Do not load options from a file. Doing so can cause code generation to fail. Instead, create options in your code.

- If you specify an option that is not supported, the option is typically ignored during code generation. For reliable results, specify only supported options.

For an example, see "Generate Code for quadprog" on page 11-60.

See Also
Optimize | linprog | lsqlin | optimoptions | prob2struct

Topics
"Solver-Based Optimization Problem Setup"
"Optimization Results"
"Quadratic Programming and Cone Programming"
"Mixed-Integer Quadratic Programming Portfolio Optimization: Solver-Based" on page 9-82

Introduced before R2006a

resetoptions

Reset options

Syntax

```
options2 = resetoptions(options,optionname)
options2 = resetoptions(options,multioptions)
```

Description

`options2 = resetoptions(options,optionname)` resets the specified option back to its default value.

Tip If you want only one set of options, use `options` as the output argument instead of `options2`.

`options2 = resetoptions(options,multioptions)` resets multiple options back to their default values.

Examples

Reset One Option

Create options with some nondefault settings. Examine the `MaxIterations` setting.

```
options = optimoptions('fmincon','Algorithm','sqp','MaxIterations',2e4,...
    'SpecifyObjectiveGradient',true);
options.MaxIterations
```

```
ans =

      20000
```

Reset the `MaxIterations` option to its default value.

```
options2 = resetoptions(options,'MaxIterations');
options2.MaxIterations
```

```
ans =

   400
```

The default value of the `MaxIterations` option is 400 for the `'sqp'` algorithm.

Reset Multiple Options

Create options with some nondefault settings. Examine the `MaxIterations` setting.

```
options = optimoptions('fmincon','Algorithm','sqp','MaxIterations',2e4,...
    'SpecifyObjectiveGradient',true);
options.MaxIterations
```

```
ans =

    20000
```

Reset the `MaxIterations` and `Algorithm` options to their default values. Examine the `MaxIterations` setting.

```
multiopts = {'MaxIterations','Algorithm'};
options2 = resetoptions(options,multiopts);
options2.MaxIterations
```

```
ans =

    1000
```

The default value of the `MaxIterations` option is 1000 for the default `'interior-point'` algorithm.

Input Arguments

`options` — Optimization options
object as created by `optimoptions`

Optimization options, specified as an object as created by `optimoptions`.

Example: `optimoptions('fmincon','Algorithm','sqp','SpecifyObjectiveGradient',true)`

`optionname` — Option name
name in single quote marks

Option names, specified as a name in single quote marks. The allowable option names for each solver are listed in the `options` section of the function reference page.

Example: `'Algorithm'`

Data Types: `char`

`multioptions` — Multiple options
cell array of names

Multiple options, specified as a cell array of names.

Example: `{'Algorithm','OptimalityTolerance'}`

Data Types: `cell`

Output Arguments

options2 — Optimization options
object as created by `optimoptions`

Optimization options, returned as an object as created by `optimoptions`.

See Also

`optimoptions`

Topics
"Set Options"

Introduced in R2016a

secondordercone

Create second-order cone constraint

Syntax

```
socConstraint = secondordercone(A,b,d,gamma)
```

Description

The `secondordercone` function creates a second-order cone constraint representing the inequality

$$\|A \cdot x - b\| \le d^T \cdot x - \gamma$$

from the input matrices A, b, d, and `gamma`.

`socConstraint = secondordercone(A,b,d,gamma)` creates a second-order cone constraint object `socConstraint`.

Solve problems with second-order cone constraints by using the `coneprog` function. To represent multiple cone constraints, pass an array of these constraints to `coneprog` as shown in the example "Several Cone Constraints" on page 16-433.

Examples

Single Cone Constraint

To set up a problem with a second-order cone constraint, create a second-order cone constraint object.

```
A = diag([1,1/2,0]);
b = zeros(3,1);
d = [0;0;1];
gamma = 0;
socConstraints = secondordercone(A,b,d,gamma);
```

Create an objective function vector.

```
f = [-1,-2,0];
```

The problem has no linear constraints. Create empty matrices for these constraints.

```
Aineq = [];
bineq = [];
Aeq = [];
beq = [];
```

Set upper and lower bounds on `x(3)`.

```
lb = [-Inf,-Inf,0];
ub = [Inf,Inf,2];
```

Solve the problem by using the `coneprog` function.

```
[x,fval] = coneprog(f,socConstraints,Aineq,bineq,Aeq,beq,lb,ub)
```

Optimal solution found.

x = *3×1*

```
    0.4851
    3.8806
    2.0000
```

fval = -8.2462

The solution component `x(3)` is at its upper bound. The cone constraint is active at the solution:

```
norm(A*x-b) - d'*x % Near 0 when the constraint is active
```

ans = -2.5677e-08

Several Cone Constraints

To set up a problem with several second-order cone constraints, create an array of constraint objects. To save time and memory, create the highest-index constraint first.

```
A = diag([1,2,0]);
b = zeros(3,1);
d = [0;0;1];
gamma = -1;
socConstraints(3) = secondordercone(A,b,d,gamma);

A = diag([3,0,1]);
d = [0;1;0];
socConstraints(2) = secondordercone(A,b,d,gamma);

A = diag([0;1/2;1/2]);
d = [1;0;0];
socConstraints(1) = secondordercone(A,b,d,gamma);
```

Create the linear objective function vector.

```
f = [-1;-2;-4];
```

Solve the problem by using the `coneprog` function.

```
[x,fval] = coneprog(f,socConstraints)
```

Optimal solution found.

x = *3×1*

```
    0.4238
    1.6477
    2.3225
```

```
fval = -13.0089
```

Input Arguments

A — Linear factor of cone
real matrix

Linear factor of the cone, specified as a real matrix. The number of columns in A must equal the number of elements in d, and the number of rows in A must equal the number of elements in b.

Example: `diag([1,1/2,0])`

Data Types: `double`

b — Center of cone
real vector

Center of the cone, specified as a real vector. The number of elements in b must equal the number of rows in A.

Example: `zeros(3,1)`

Data Types: `double`

d — Linear bound
real vector

Linear bound, specified as a real vector. The number of elements in d must equal the number of columns in A.

Example: `[0;0;1]`

Data Types: `double`

gamma — Bound
real scalar

Bound, specified as a real scalar. Smaller values of `gamma` correspond to looser constraints.

Example: `-1`

Data Types: `double`

Output Arguments

`socConstraint` — Second-order cone constraint
`SecondOrderConeConstraint` object

Second-order cone constraint, returned as a `SecondOrderConeConstraint` object. Use this object as a constraint for the `coneprog` solver. If you have multiple cone constraints, pass a vector of constraints to `coneprog`; see "Several Cone Constraints" on page 16-433.

See Also
`SecondOrderConeConstraint` | `coneprog`

Topics
"Quadratic Programming and Cone Programming"

Introduced in R2020b

SecondOrderConeConstraint

Second-order cone constraint object

Description

`SecondOrderConeConstraint` represents the second-order cone constraint

$$\|A \cdot x - b\| \le d^T \cdot x - \gamma$$

- The A matrix represents the linear factor of the cone.
- The b vector represents the center of the cone.
- The d vector represents a linear bound.
- The γ scalar represents a bound.

Solve problems with second-order cone constraints by using the `coneprog` function.

Creation

Create a `SecondOrderConeConstraint` object by using the `secondordercone` function.

Properties

A — Linear factor of cone
real matrix

Linear factor of the cone, specified as a real matrix.

Data Types: `double`

b — Center of cone
real vector

Center of the cone, specified as a real vector.

Data Types: `double`

d — Linear bound
real vector

Linear bound, specified as a real vector.

Data Types: `double`

gamma — Bound
real scalar

Bound, specified as a real scalar. Smaller values of `gamma` correspond to looser constraints.

Data Types: `double`

Object Functions

Examples

Single Cone Constraint

To set up a problem with a second-order cone constraint, create a second-order cone constraint object.

```
A = diag([1,1/2,0]);
b = zeros(3,1);
d = [0;0;1];
gamma = 0;
socConstraints = secondordercone(A,b,d,gamma);
```

Create an objective function vector.

```
f = [-1,-2,0];
```

The problem has no linear constraints. Create empty matrices for these constraints.

```
Aineq = [];
bineq = [];
Aeq = [];
beq = [];
```

Set upper and lower bounds on `x(3)`.

```
lb = [-Inf,-Inf,0];
ub = [Inf,Inf,2];
```

Solve the problem by using the `coneprog` function.

```
[x,fval] = coneprog(f,socConstraints,Aineq,bineq,Aeq,beq,lb,ub)
```

```
Optimal solution found.

x = 3×1

    0.4851
    3.8806
    2.0000


fval = -8.2462
```

The solution component `x(3)` is at its upper bound. The cone constraint is active at the solution:

```
norm(A*x-b) - d'*x % Near 0 when the constraint is active
```

```
ans = -2.5677e-08
```

Several Cone Constraints

To set up a problem with several second-order cone constraints, create an array of constraint objects. To save time and memory, create the highest-index constraint first.

```
A = diag([1,2,0]);
b = zeros(3,1);
d = [0;0;1];
gamma = -1;
socConstraints(3) = secondordercone(A,b,d,gamma);

A = diag([3,0,1]);
d = [0;1;0];
socConstraints(2) = secondordercone(A,b,d,gamma);

A = diag([0;1/2;1/2]);
d = [1;0;0];
socConstraints(1) = secondordercone(A,b,d,gamma);
```

Create the linear objective function vector.

```
f = [-1;-2;-4];
```

Solve the problem by using the `coneprog` function.

```
[x,fval] = coneprog(f,socConstraints)
```

```
Optimal solution found.

x = 3×1

    0.4238
    1.6477
    2.3225


fval = -13.0089
```

See Also
coneprog | secondordercone

Topics
"Quadratic Programming and Cone Programming"

Introduced in R2020b

show

Package: `optim.problemdef`

Display information about optimization object

Syntax

```
show(obj)
```

Description

Use `show` to display information about an optimization object.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`show(obj)` displays information about `obj` at the command line. If the object display is large, consider using `write` instead to save the information in a text file.

Examples

Examine Problem-Based Setup

Examine the various stages of problem construction for optimizing the Rosenbrock function confined to the unit disk (see "Solve a Constrained Nonlinear Problem, Problem-Based" on page 1-5).

Create a 2-D optimization variable `x`. Show the variable.

```
x = optimvar('x',2);
show(x)

    [ x(1) ]
    [ x(2) ]
```

Create an expression for the objective function. Show the expression.

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

  ((100 .* (x(2) - x(1).^2).^2) + (1 - x(1)).^2)
```

Create an expression for the constraint. Show the constraint.

```
cons = x(1)^2 + x(2)^2 <= 1;
show(cons)

  (x(1).^2 + x(2).^2) <= 1
```

Create an optimization problem that has `obj` as the objective function and `cons` as the constraint. Show the problem.

```
prob = optimproblem("Objective",obj,"Constraints",cons);
show(prob)

  OptimizationProblem :

    Solve for:
       x

    minimize :
       ((100 .* (x(2) - x(1).^2).^2) + (1 - x(1)).^2)


    subject to :
       (x(1).^2 + x(2).^2) <= 1
```

Finally, create an initial point `[0 0]` and solve the problem starting at the initial point.

```
x0.x = [0 0];
[sol,fval,exitflag] = solve(prob,x0)

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol = struct with fields:
    x: [2x1 double]


fval = 0.0457

exitflag =
    OptimalSolution
```

Examine the solution point.

```
sol.x

ans = 2×1

    0.7864
    0.6177
```

Input Arguments

obj — Optimization object
OptimizationProblem object | EquationProblem object | OptimizationExpression object |
OptimizationVariable object | OptimizationConstraint object | OptimizationEquality
object | OptimizationInequality object

Optimization object, specified as one of the following:

- OptimizationProblem object — show(obj) displays the variables for the solution, objective function, constraints, and variable bounds.
- EquationProblem object — show(obj) displays the variables for the solution, equations for the solution, and variable bounds.
- OptimizationExpression object — show(obj) displays the optimization expression.
- OptimizationVariable object — show(obj) displays the optimization variables. This display does not indicate variable types or bounds; it shows only the variable dimensions and index names (if any).
- OptimizationConstraint object — show(obj) displays the constraint expression.
- OptimizationEquality object — show(obj) displays the equality expression.
- OptimizationInequality object — show(obj) displays the inequality expression.

See Also
showbounds | write

Topics
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2019b

showbounds

Package: `optim.problemdef`

Display variable bounds

Syntax

`showbounds(var)`

Description

Use `showbounds` to display the bounds on optimization variables.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`showbounds(var)` displays the bounds for `var`.

Examples

Display Optimization Variable Bounds

Show bounds for various optimization variables.

Create a continuous optimization variable array and display its bounds.

```
x = optimvar('x',2,2);
showbounds(x)

    x is unbounded.
```

Set lower bounds of 0 on all elements of x, and set upper bounds on the first row.

```
x.LowerBound = 0;
x.UpperBound(1,:) = [3,5];
showbounds(x)

    0 <= x(1, 1) <= 3
    0 <= x(2, 1)
    0 <= x(1, 2) <= 5
    0 <= x(2, 2)
```

Create a binary optimization variable array and display its bounds.

```
binvar = optimvar('binvar',2,2,'Type','integer',...
    'LowerBound',0,'UpperBound',1);
showbounds(binvar)

    0 <= binvar(1, 1) <= 1
    0 <= binvar(2, 1) <= 1
```

```
   0 <= binvar(1, 2) <= 1
   0 <= binvar(2, 2) <= 1
```

Create a large optimization variable that has few bounded elements, and display the variable bounds.

```
bigvar = optimvar('bigvar',100,10,50);
bigvar.LowerBound(55,4,3) = -20;
bigvar.LowerBound(20,5,30) = -40;
bigvar.UpperBound(35,3,35) = -200;
showbounds(bigvar)

   -20 <= bigvar(55, 4, 3)
   -40 <= bigvar(20, 5, 30)
          bigvar(35, 3, 35) <= -200
```

Input Arguments

var — Optimization variable
OptimizationVariable object

Optimization variable, specified as an `OptimizationVariable` object. Create `var` using `optimvar`.

Example: `var = optimvar('var',4,6)`

Tips

- For a variable that has many bounds, use `writebounds` to generate a text file containing the bound information.

See Also
OptimizationVariable | optimvar | writebounds

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

showconstr

Package: `optim.problemdef`

(Not recommended) Display optimization constraint

Syntax

`showconstr(constr)`

Description

`showconstr` is not recommended. Use `show` instead.

`showconstr(constr)` displays the optimization constraint `constr` at the MATLAB Command Window.

Examples

Display Optimization Constraint

Display an array of optimization constraints.

```
x = optimvar('x',3,2);
constr = sum(x,2) <= [1;3;2];
showconstr(constr)
```

```
(1, 1)

  x(1, 1) + x(1, 2) <= 1

(2, 1)

  x(2, 1) + x(2, 2) <= 3

(3, 1)

  x(3, 1) + x(3, 2) <= 2
```

Input Arguments

constr — Optimization constraint
`OptimizationEquality` object | `OptimizationInequality` object | `OptimizationConstraint` object

Optimization constraint, specified as an `OptimizationEquality` object, `OptimizationInequality` object, or `OptimizationConstraint` object. `constr` can represent a single constraint or an array of constraints.

Example: `constr = x + y <= 1` is a single constraint when x and y are scalar variables.

Example: `constr = sum(x) == 1` is an array of constraints when `x` is an array of two or more dimensions.

Tips

- For a large or complicated constraint, use `writeconstr` to generate a text file containing the constraint information.

Compatibility Considerations

`showconstr` is not recommended
Not recommended starting in R2019b

The `showconstr` function is not recommended. Instead, use `show`. The `show` function replaces `showconstr` and many other problem-based functions.

There are no plans to remove `showconstr` at this time.

See Also
`OptimizationConstraint` | `show` | `write`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

showexpr

Package: `optim.problemdef`

(Not recommended) Display optimization expression

Syntax

```
showexpr(expr)
```

Description

`showexpr` is not recommended. Use `show` instead.

`showexpr(expr)` displays the optimization expression `expr` at the MATLAB Command Window.

Examples

Display Optimization Expression

Create an optimization variable and an expression.

```
x = optimvar('x',3,3);
A = magic(3);
expr = sum(sum(A.*x));
```

Display the expression.

```
showexpr(expr)
```

```
  8*x(1, 1) + 3*x(2, 1) + 4*x(3, 1) + x(1, 2) + 5*x(2, 2) + 9*x(3, 2)
+ 6*x(1, 3) + 7*x(2, 3) + 2*x(3, 3)
```

Input Arguments

expr — Optimization expression
OptimizationExpression object

Optimization expression, specified as an `OptimizationExpression` object.

Example: `sum(sum(x))`

Tips

- For an expression that has many terms, use `writeexpr` to generate a text file containing the expression information.

Compatibility Considerations

showexpr is not recommended
Not recommended starting in R2019b

The showexpr function is not recommended. Instead, use show. The show function replaces showexpr and many other problem-based functions.

There are no plans to remove showexpr at this time.

See Also

OptimizationExpression | show | write

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

showproblem

Package: `optim.problemdef`

(Not recommended) Display optimization problem

Syntax

`showproblem(prob)`

Description

`showproblem` is not recommended. Use `show` instead.

`showproblem(prob)` displays the objective function, constraints, and bounds of `prob`.

Examples

Display Optimization Problem

Create an optimization problem, including an objective function and constraints, and display the problem.

Create the problem in "Mixed-Integer Linear Programming Basics: Problem-Based" on page 10-47.

```
steelprob = optimproblem;
ingots = optimvar('ingots',4,1,'Type','integer','LowerBound',0,'UpperBound',1);
alloys = optimvar('alloys',4,1,'LowerBound',0);
weightIngots = [5,3,4,6];
costIngots = weightIngots.*[350,330,310,280];
costAlloys = [500,450,400,100];
cost = costIngots*ingots + costAlloys*alloys;
steelprob.Objective = cost;
totalweight = weightIngots*ingots + sum(alloys);
carbonIngots = [5,4,5,3]/100;
carbonAlloys = [8,7,6,3]/100;
totalCarbon = (weightIngots.*carbonIngots)*ingots + carbonAlloys*alloys;
molybIngots = [3,3,4,4,]/100;
molybAlloys = [6,7,8,9]/100;
totalMolyb = (weightIngots.*molybIngots)*ingots + molybAlloys*alloys;
steelprob.Constraints.conswt = totalweight == 25;
steelprob.Constraints.conscarb = totalCarbon == 1.25;
steelprob.Constraints.consmolyb = totalMolyb == 1.25;
```

Display the problem.

```
showproblem(steelprob)

  OptimizationProblem :

    Solve for:
       alloys, ingots
```

```
minimize :
    1750*ingots(1) + 990*ingots(2) + 1240*ingots(3) + 1680*ingots(4)
 + 500*alloys(1) + 450*alloys(2) + 400*alloys(3) + 100*alloys(4)


subject to conswt:
    5*ingots(1) + 3*ingots(2) + 4*ingots(3) + 6*ingots(4) + alloys(1)
 + alloys(2) + alloys(3) + alloys(4) == 25

subject to conscarb:
    0.25*ingots(1) + 0.12*ingots(2) + 0.2*ingots(3) + 0.18*ingots(4)
 + 0.08*alloys(1) + 0.07*alloys(2) + 0.06*alloys(3) + 0.03*alloys(4) == 1.25

subject to consmolyb:
    0.15*ingots(1) + 0.09*ingots(2) + 0.16*ingots(3) + 0.24*ingots(4)
 + 0.06*alloys(1) + 0.07*alloys(2) + 0.08*alloys(3) + 0.09*alloys(4) == 1.25

variable bounds:
    0 <= alloys(1)
    0 <= alloys(2)
    0 <= alloys(3)
    0 <= alloys(4)

    0 <= ingots(1) <= 1
    0 <= ingots(2) <= 1
    0 <= ingots(3) <= 1
    0 <= ingots(4) <= 1
```

Input Arguments

prob — Optimization problem or equation problem
`OptimizationProblem` object | `EquationProblem` object

Optimization problem or equation problem, specified as an `OptimizationProblem` object or an `EquationProblem` object. Create an optimization problem by using `optimproblem`; create an equation problem by using `eqnproblem`.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

Example: `prob = optimproblem; prob.Objective = obj; prob.Constraints.cons1 = cons1;`

Example: `prob = eqnproblem; prob.Equations = eqs;`

Tips

- `showproblem` is equivalent to calling all of the following:

  - `showexpr(prob.Objective)`

- • showconstr on each constraint in `prob.Constraints`
- • showbounds on all the variables in `prob`
- For a problem that has many bounds or constraints, use `writeproblem` to generate a text file containing the objective, constraint, and bound information.

Compatibility Considerations

showproblem is not recommended
Not recommended starting in R2019b

The `showproblem` function is not recommended. Instead, use `show`. The `show` function replaces `showproblem` and many other problem-based functions.

There are no plans to remove `showproblem` at this time.

See Also
OptimizationProblem | show | showbounds | write

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

showvar

Package: `optim.problemdef`

(Not recommended) Display optimization variable

Syntax

```
showvar(var)
```

Description

`showvar` is not recommended. Use `show` instead.

`showvar(var)` displays the optimization variable `var` at the Command Window.

Examples

Display Optimization Variable

Create an optimization variable and display it.

```
var = optimvar('var',8,3,'Type','integer');
showvar(var)

    [ var(1, 1)     var(1, 2)     var(1, 3) ]
    [ var(2, 1)     var(2, 2)     var(2, 3) ]
    [ var(3, 1)     var(3, 2)     var(3, 3) ]
    [ var(4, 1)     var(4, 2)     var(4, 3) ]
    [ var(5, 1)     var(5, 2)     var(5, 3) ]
    [ var(6, 1)     var(6, 2)     var(6, 3) ]
    [ var(7, 1)     var(7, 2)     var(7, 3) ]
    [ var(8, 1)     var(8, 2)     var(8, 3) ]
```

Input Arguments

var — Optimization variable
`OptimizationVariable` object

Optimization variable, specified as an `OptimizationVariable` object. Create `var` using `optimvar`.

Example: `var = optimvar('var',4,6)`

Compatibility Considerations

showvar is not recommended
Not recommended starting in R2019b

The `showvar` function is not recommended. Instead, use `show`. The `show` function replaces `showvar` and many other problem-based functions.

There are no plans to remove `showvar` at this time.

See Also
`OptimizationVariable` | `optimvar` | `show` | `write`

Topics
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

solve

Package: optim.problemdef

Solve optimization problem or equation problem

Syntax

```
sol = solve(prob)
sol = solve(prob,x0)
sol = solve( ___ ,Name,Value)
[sol,fval] = solve( ___ )
[sol,fval,exitflag,output,lambda] = solve( ___ )
```

Description

Use `solve` to find the solution of an optimization problem or equation problem.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`sol = solve(prob)` solves the optimization problem or equation problem `prob`.

`sol = solve(prob,x0)` solves `prob` starting from the point `x0`.

`sol = solve( ___ ,Name,Value)` modifies the solution process using one or more name-value pair arguments in addition to the input arguments in previous syntaxes.

`[sol,fval] = solve( ___ )` also returns the objective function value at the solution using any of the input arguments in previous syntaxes.

`[sol,fval,exitflag,output,lambda] = solve( ___ )` also returns an exit flag describing the exit condition, an `output` structure containing additional information about the solution process, and, for non-integer optimization problems, a Lagrange multiplier structure.

Examples

Solve Linear Programming Problem

Solve a linear programming problem defined by an optimization problem.

```
x = optimvar('x');
y = optimvar('y');
prob = optimproblem;
prob.Objective = -x - y/3;
prob.Constraints.cons1 = x + y <= 2;
prob.Constraints.cons2 = x + y/4 <= 1;
prob.Constraints.cons3 = x - y <= 2;
```

```
prob.Constraints.cons4 = x/4 + y >= -1;
prob.Constraints.cons5 = x + y >= 1;
prob.Constraints.cons6 = -x + y <= 2;

sol = solve(prob)

Solving problem using linprog.

Optimal solution found.

sol = struct with fields:
    x: 0.6667
    y: 1.3333
```

Solve Nonlinear Programming Problem Using Problem-Based Approach

Find a minimum of the peaks function, which is included in MATLAB®, in the region $x^2 + y^2 \leq 4$. To do so, create optimization variables x and y.

```
x = optimvar('x');
y = optimvar('y');
```

Create an optimization problem having peaks as the objective function.

```
prob = optimproblem("Objective",peaks(x,y));
```

Include the constraint as an inequality in the optimization variables.

```
prob.Constraints = x^2 + y^2 <= 4;
```

Set the initial point for x to 1 and y to –1, and solve the problem.

```
x0.x = 1;
x0.y = -1;
sol = solve(prob,x0)

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol = struct with fields:
    x: 0.2283
    y: -1.6255
```

Unsupported Functions Require `fcn2optimexpr`

If your objective or nonlinear constraint functions are not entirely composed of elementary functions, you must convert the functions to optimization expressions using `fcn2optimexpr`. See "Convert Nonlinear Function to Optimization Expression" on page 7-8 and "Supported Operations on Optimization Variables and Expressions" on page 10-43.

To convert the present example:

```
convpeaks = fcn2optimexpr(@peaks,x,y);
prob.Objective = convpeaks;
sol2 = solve(prob,x0)

Solving problem using fmincon.

Local minimum found that satisfies the constraints.

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

sol2 = struct with fields:
    x: 0.2283
    y: -1.6255
```

Copyright 2018–2020 The MathWorks, Inc.

Solve Mixed-Integer Linear Program Starting from Initial Point

Compare the number of steps to solve an integer programming problem both with and without an initial feasible point. The problem has eight integer variables and four linear equality constraints, and all variables are restricted to be positive.

```
prob = optimproblem;
x = optimvar('x',8,1,'LowerBound',0,'Type','integer');
```

Create four linear equality constraints and include them in the problem.

```
Aeq = [22    13    26    33    21     3    14    26
       39    16    22    28    26    30    23    24
       18    14    29    27    30    38    26    26
       41    26    28    36    18    38    16    26];
beq = [ 7872
       10466
       11322
       12058];
cons = Aeq*x == beq;
prob.Constraints.cons = cons;
```

Create an objective function and include it in the problem.

```
f = [2    10    13    17     7     5     7     3];
prob.Objective = f*x;
```

Solve the problem without using an initial point, and examine the display to see the number of branch-and-bound nodes.

```
[x1,fval1,exitflag1,output1] = solve(prob);

Solving problem using intlinprog.
LP:                Optimal objective value is 1554.047531.

Cut Generation:    Applied 8 strong CG cuts.
```

```
                  Lower bound is 1591.000000.

Branch and Bound:

   nodes      total   num int       integer       relative
explored   time (s)  solution          fval        gap (%)
   10000       0.76         0             -              -
   18027       1.32         1   2.906000e+03   4.509804e+01
   21859       1.72         2   2.073000e+03   2.270974e+01
   23546       1.85         3   1.854000e+03   1.180593e+01
   24121       1.90         3   1.854000e+03   1.563342e+00
   24294       1.92         3   1.854000e+03   0.000000e+00

Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the
optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon
variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the
default value).
```

For comparison, find the solution using an initial feasible point.

```
x0.x = [8 62 23 103 53 84 46 34]';
[x2,fval2,exitflag2,output2] = solve(prob,x0);
```

```
Solving problem using intlinprog.
LP:                Optimal objective value is 1554.047531.

Cut Generation:    Applied 8 strong CG cuts.
                   Lower bound is 1591.000000.
                   Relative gap is 59.20%.

Branch and Bound:

   nodes      total   num int       integer       relative
explored   time (s)  solution          fval        gap (%)
    3627       0.41         2   2.154000e+03   2.593968e+01
    5844       0.66         3   1.854000e+03   1.180593e+01
    6204       0.69         3   1.854000e+03   1.455526e+00
    6400       0.72         3   1.854000e+03   0.000000e+00

Optimal solution found.

Intlinprog stopped because the objective value is within a gap tolerance of the
optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon
variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the
default value).
```

```
fprintf('Without an initial point, solve took %d steps.\nWith an initial point, solve took %d ste
```

```
Without an initial point, solve took 24294 steps.
With an initial point, solve took 6400 steps.
```

Giving an initial point does not always improve the problem. For this problem, using an initial point saves time and computational steps. However, for some problems, an initial point can cause solve to take more steps.

Solve Integer Programming Problem with Nondefault Options

Solve the problem

$$\min_{x}(-3x_1 - 2x_2 - x_3) \text{ subject to} \begin{cases} x_3 \text{ binary} \\ x_1, x_2 \geq 0 \\ x_1 + x_2 + x_3 \leq 7 \\ 4x_1 + 2x_2 + x_3 = 12 \end{cases}$$

without showing iterative display.

```
x = optimvar('x',2,1,'LowerBound',0);
x3 = optimvar('x3','Type','integer','LowerBound',0,'UpperBound',1);
prob = optimproblem;
prob.Objective = -3*x(1) - 2*x(2) - x3;
prob.Constraints.cons1 = x(1) + x(2) + x3 <= 7;
prob.Constraints.cons2 = 4*x(1) + 2*x(2) + x3 == 12;

options = optimoptions('intlinprog','Display','off');

sol = solve(prob,'Options',options)

sol = struct with fields:
     x: [2x1 double]
    x3: 1
```

Examine the solution.

```
sol.x

ans = 2×1

         0
    5.5000
```

```
sol.x3

ans = 1
```

Use `intlinprog` to Solve a Linear Program

Force `solve` to use `intlinprog` as the solver for a linear programming problem.

```
x = optimvar('x');
y = optimvar('y');
prob = optimproblem;
prob.Objective = -x - y/3;
prob.Constraints.cons1 = x + y <= 2;
prob.Constraints.cons2 = x + y/4 <= 1;
prob.Constraints.cons3 = x - y <= 2;
prob.Constraints.cons4 = x/4 + y >= -1;
prob.Constraints.cons5 = x + y >= 1;
```

```
prob.Constraints.cons6 = -x + y <= 2;

sol = solve(prob,'Solver', 'intlinprog')

Solving problem using intlinprog.
LP:                 Optimal objective value is -1.111111.


Optimal solution found.

No integer variables specified. Intlinprog solved the linear problem.

sol = struct with fields:
    x: 0.6667
    y: 1.3333
```

Return All Outputs

Solve the mixed-integer linear programming problem described in "Solve Integer Programming Problem with Nondefault Options" on page 16-456 and examine all of the output data.

```
x = optimvar('x',2,1,'LowerBound',0);
x3 = optimvar('x3','Type','integer','LowerBound',0,'UpperBound',1);
prob = optimproblem;
prob.Objective = -3*x(1) - 2*x(2) - x3;
prob.Constraints.cons1 = x(1) + x(2) + x3 <= 7;
prob.Constraints.cons2 = 4*x(1) + 2*x(2) + x3 == 12;

[sol,fval,exitflag,output] = solve(prob)

Solving problem using intlinprog.
LP:                 Optimal objective value is -12.000000.


Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap
tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).

sol = struct with fields:
     x: [2x1 double]
    x3: 1


fval = -12

exitflag =
    OptimalSolution


output = struct with fields:
        relativegap: 0
        absolutegap: 0
```

```
      numfeaspoints: 1
           numnodes: 0
      constrviolation: 0
             message: 'Optimal solution found....'
              solver: 'intlinprog'
```

For a problem without any integer constraints, you can also obtain a nonempty Lagrange multiplier structure as the fifth output.

View Solution with Index Variables

Create and solve an optimization problem using named index variables. The problem is to maximize the profit-weighted flow of fruit to various airports, subject to constraints on the weighted flows.

```matlab
rng(0) % For reproducibility
p = optimproblem('ObjectiveSense', 'maximize');
flow = optimvar('flow', ...
    {'apples', 'oranges', 'bananas', 'berries'}, {'NYC', 'BOS', 'LAX'}, ...
    'LowerBound',0,'Type','integer');
p.Objective = sum(sum(rand(4,3).*flow));
p.Constraints.NYC = rand(1,4)*flow(:,'NYC') <= 10;
p.Constraints.BOS = rand(1,4)*flow(:,'BOS') <= 12;
p.Constraints.LAX = rand(1,4)*flow(:,'LAX') <= 35;
sol = solve(p);
```

```
Solving problem using intlinprog.
LP:                Optimal objective value is -1027.472366.

Heuristics:        Found 1 solution using ZI round.
                   Upper bound is -1027.233133.
                   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.AbsoluteGapTolerance = 0 (the default
value). The intcon variables are integer within tolerance,
options.IntegerTolerance = 1e-05 (the default value).
```

Find the optimal flow of oranges and berries to New York and Los Angeles.

```matlab
[idxFruit,idxAirports] = findindex(flow, {'oranges','berries'}, {'NYC', 'LAX'})
```

```
idxFruit = 1×2

     2     4


idxAirports = 1×2

     1     3


orangeBerries = sol.flow(idxFruit, idxAirports)
```

```
orangeBerries = 2×2

         0   980.0000
   70.0000          0
```

This display means that no oranges are going to NYC, 70 berries are going to NYC, 980 oranges are going to LAX, and no berries are going to LAX.

List the optimal flow of the following:

```
Fruit Airports

----- --------

Berries NYC

Apples BOS

Oranges LAX
```

```
idx = findindex(flow, {'berries', 'apples', 'oranges'}, {'NYC', 'BOS', 'LAX'})
```

```
idx = 1×3

     4     5    10
```

```
optimalFlow = sol.flow(idx)
```

```
optimalFlow = 1×3

   70.0000   28.0000   980.0000
```

This display means that 70 berries are going to NYC, 28 apples are going to BOS, and 980 oranges are going to LAX.

Solve Nonlinear System of Equations, Problem-Based

To solve the nonlinear system of equations

$$\exp(-\exp(-(x_1 + x_2))) = x_2\left(1 + x_1^2\right)$$

$$x_1\cos(x_2) + x_2\sin(x_1) = \frac{1}{2}$$

using the problem-based approach, first define x as a two-element optimization variable.

```
x = optimvar('x',2);
```

Create the first equation as an optimization equality expression.

```
eq1 = exp(-exp(-(x(1) + x(2)))) == x(2)*(1 + x(1)^2);
```

Similarly, create the second equation as an optimization equality expression.

```
eq2 = x(1)*cos(x(2)) + x(2)*sin(x(1)) == 1/2;
```

Create an equation problem, and place the equations in the problem.

```
prob = eqnproblem;
prob.Equations.eq1 = eq1;
prob.Equations.eq2 = eq2;
```

Review the problem.

```
show(prob)
```

```
  EquationProblem :

    Solve for:
       x


  eq1:
       exp(-exp(-(x(1) + x(2)))) == (x(2) .* (1 + x(1).^2))

  eq2:
       ((x(1) .* cos(x(2))) + (x(2) .* sin(x(1)))) == 0.5
```

Solve the problem starting from the point [0,0]. For the problem-based approach, specify the initial point as a structure, with the variable names as the fields of the structure. For this problem, there is only one variable, x.

```
x0.x = [0 0];
[sol,fval,exitflag] = solve(prob,x0)
```

```
Solving problem using fsolve.

Equation solved.

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

sol = struct with fields:
    x: [2x1 double]


fval = struct with fields:
    eq1: -2.4069e-07
    eq2: -3.8253e-08


exitflag =
    EquationSolved
```

View the solution point.

```
disp(sol.x)
```

```
    0.3532
    0.6061
```

Unsupported Functions Require `fcn2optimexpr`

If your equation functions are not composed of elementary functions, you must convert the functions to optimization expressions using `fcn2optimexpr`. For the present example:

```
ls1 = fcn2optimexpr(@(x)exp(-exp(-(x(1)+x(2)))),x);
eq1 = ls1 == x(2)*(1 + x(1)^2);
ls2 = fcn2optimexpr(@(x)x(1)*cos(x(2))+x(2)*sin(x(1)),x);
eq2 = ls2 == 1/2;
```

See "Supported Operations on Optimization Variables and Expressions" on page 10-43 and "Convert Nonlinear Function to Optimization Expression" on page 7-8.

Input Arguments

`prob` — Optimization problem or equation problem
`OptimizationProblem` object | `EquationProblem` object

Optimization problem or equation problem, specified as an `OptimizationProblem` object or an `EquationProblem` object. Create an optimization problem by using `optimproblem`; create an equation problem by using `eqnproblem`.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

Example: `prob = optimproblem; prob.Objective = obj; prob.Constraints.cons1 = cons1;`

Example: `prob = eqnproblem; prob.Equations = eqs;`

`x0` — Initial point
structure

Initial point, specified as a structure with field names equal to the variable names in `prob`.

For an example using `x0` with named index variables, see "Create Initial Point for Optimization with Named Index Variables" on page 10-50.

Example: If `prob` has variables named x and y: `x0.x = [3,2,17]; x0.y = [pi/3,2*pi/3]`.

Data Types: `struct`

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `solve(prob,'options',opts)`

`options` — Optimization options
object created by `optimoptions` | options structure

Optimization options, specified as the comma-separated pair consisting of `'options'` and an object created by `optimoptions` or an options structure such as created by `optimset`.

Internally, the `solve` function calls a relevant solver as detailed in the `'solver'` argument reference. Ensure that `options` is compatible with the solver. For example, `intlinprog` does not allow options to be a structure, and `lsqnonneg` does not allow options to be an object.

For suggestions on options settings to improve an `intlinprog` solution or the speed of a solution, see "Tuning Integer Linear Programming" on page 9-52. For `linprog`, the default `'dual-simplex'` algorithm is generally memory-efficient and speedy. Occasionally, `linprog` solves a large problem faster when the `Algorithm` option is `'interior-point'`. For suggestions on options settings to improve a nonlinear problem's solution, see "Options in Common Use: Tuning and Troubleshooting" on page 2-61 and "Improve Results".

Example: `options = optimoptions('intlinprog','Display','none')`

solver — Optimization solver
`'intlinprog'` | `'linprog'` | `'lsqlin'` | `'lsqcurvefit'` | `'lsqnonlin'` | `'lsqnonneg'` | `'quadprog'` | `'fminunc'` | `'fmincon'` | `'fzero'` | `'fsolve'`

Optimization solver, specified as the comma-separated pair consisting of `'solver'` and the name of a listed solver. For optimization problems, this table contains the available solvers for each problem type.

Problem Type	Default Solver	Other Allowed Solvers
Linear objective, linear constraints	`linprog`	`intlinprog`, `quadprog`, `fmincon`, `fminunc` (`fminunc` is not recommended because unconstrained linear programs are either constant or unbounded)
Linear objective, linear and integer constraints	`intlinprog`	`linprog` (integer constraints ignored)
Quadratic objective, linear constraints	`quadprog`	`fmincon`, `fminunc` (with no constraints)
Minimize $\|\|C*x - d\|\|^2$ subject to linear constraints	`lsqlin` when the objective is a constant plus a sum of squares of linear expressions	`quadprog`, `lsqnonneg` (Constraints other than x >= 0 are ignored for `lsqnonneg`), `fmincon`, `fminunc` (with no constraints)
Minimize $\|\|C*x - d\|\|^2$ subject to x >= 0	`lsqlin`	`quadprog`, `lsqnonneg`
Minimize `sum(e(i).^2)`, where `e(i)` is an optimization expression, subject to bound constraints	`lsqnonlin` when the objective has the form given in "Write Objective Function for Problem-Based Least Squares" on page 12-85	`lsqcurvefit`, `fmincon`, `fminunc` (with no constraints)
Minimize general nonlinear function $f(x)$	`fminunc`	`fmincon`

Problem Type	Default Solver	Other Allowed Solvers
Minimize general nonlinear function $f(x)$ subject to some constraints, or minimize any function subject to nonlinear constraints	fmincon	(none)

Note If you choose lsqcurvefit as the solver for a least-squares problem, solve uses lsqnonlin. The lsqcurvefit and lsqnonlin solvers are identical for solve.

Caution For maximization problems (prob.ObjectiveSense is "max" or "maximize"), do not specify a least-squares solver (one with a name beginning lsq). If you do, solve throws an error, because these solvers cannot maximize.

For equation solving, this table contains the available solvers for each problem type. In the table,

- ***** indicates the default solver for the problem type.
- **Y** indicates an available solver.
- **N** indicates an unavailable solver.

Supported Solvers for Equations

Equation Type	lsqlin	lsqnonneg	fzero	fsolve	lsqnonlin
Linear	*	N	Y (scalar only)	Y	Y
Linear plus bounds	*	Y	N	N	Y
Scalar nonlinear	N	N	*	Y	Y
Nonlinear system	N	N	N	*	Y
Nonlinear system plus bounds	N	N	N	N	*

Example: 'intlinprog'

Data Types: char | string

ObjectiveDerivative — Indication to use automatic differentiation for objective function
'auto' (default) | 'finite-differences'

Indication to use automatic differentiation (AD) for nonlinear objective function, specified as the comma-separated pair consisting of 'ObjectiveDerivative' and either 'auto' (use AD if possible) or 'finite-differences' (do not use AD). 'auto' causes the underlying solver (fmincon or fminunc) to use gradient information when solving the problem provided that the objective function is supported, as described in "Supported Operations on Optimization Variables and Expressions" on page 10-43. For an example, see "Effect of Automatic Differentiation in Problem-Based Optimization" on page 7-23.

Warning The 'ObjectiveDerivative' and 'ConstraintDerivative' name-value pair arguments currently apply only to problems solved by fmincon or fminunc. If you attempt to use

these arguments when solving an equation with `solve` or converting an equation with `prob2struct`, MATLAB issues an error.

Example: `'finite-differences'`

Data Types: `char` | `string`

`ConstraintDerivative` — Indication to use automatic differentiation for constraint functions
`'auto'` (default) | `'finite-differences'`

Indication to use automatic differentiation (AD) for nonlinear constraint functions, specified as the comma-separated pair consisting of `'ConstraintDerivative'` and either `'auto'` (use AD if possible) or `'finite-differences'` (do not use AD). `'auto'` causes the underlying solver (`fmincon`) to use gradient information when solving the problem provided that the constraint functions are supported, as described in "Supported Operations on Optimization Variables and Expressions" on page 10-43. For an example, see "Effect of Automatic Differentiation in Problem-Based Optimization" on page 7-23.

Warning The `'ObjectiveDerivative'` and `'ConstraintDerivative'` name-value pair arguments currently apply only to problems solved by `fmincon` or `fminunc`. If you attempt to use these arguments when solving an equation with `solve` or converting an equation with `prob2struct`, MATLAB issues an error.

Example: `'finite-differences'`

Data Types: `char` | `string`

Output Arguments

`sol` — Solution
structure

Solution, returned as a structure. The fields of the structure are the names of the optimization variables. See `optimvar`.

`fval` — Objective function value at the solution
real number | real vector

Objective function value at the solution, returned as a real number, or, for systems of equations, a real vector. For least-squares problems, `fval` is the sum of squares of the residuals at the solution. For equation-solving problems, `fval` is the function value at the solution, meaning the left-hand side minus the right-hand side of the equations.

Tip If you neglect to ask for `fval` for an optimization problem, you can calculate it using:

```
fval = evaluate(prob.Objective,sol)
```

`exitflag` — Reason solver stopped
enumeration variable

Reason the solver stopped, returned as an enumeration variable. You can convert `exitflag` to its numeric equivalent using `double(exitflag)`, and to its string equivalent using `string(exitflag)`.

This table describes the exit flags for the `intlinprog` solver.

Exit Flag for `intlinprog`	Numeric Equivalent	Meaning
`OptimalWithPoorFeasibility`	3	The solution is feasible with respect to the relative `ConstraintTolerance` tolerance, but is not feasible with respect to the absolute tolerance.
`IntegerFeasible`	2	`intlinprog` stopped prematurely, and found an integer feasible point.
`OptimalSolution`	1	The solver converged to a solution x.
`SolverLimitExceeded`	0	`intlinprog` exceeds one of the following tolerances: • `LPMaxIterations` • `MaxNodes` • `MaxTime` • `RootLPMaxIterations` See "Tolerances and Stopping Criteria" on page 2-68. `solve` also returns this exit flag when it runs out of memory at the root node.
`OutputFcnStop`	-1	`intlinprog` stopped by an output function or plot function.
`NoFeasiblePointFound`	-2	No feasible point found.
`Unbounded`	-3	The problem is unbounded.
`FeasibilityLost`	-9	Solver lost feasibility.

Exitflags 3 and -9 relate to solutions that have large infeasibilities. These usually arise from linear constraint matrices that have large condition number, or problems that have large solution components. To correct these issues, try to scale the coefficient matrices, eliminate redundant linear constraints, or give tighter bounds on the variables.

This table describes the exit flags for the `linprog` solver.

Exit Flag for `linprog`	Numeric Equivalent	Meaning
`OptimalWithPoorFeasibility`	3	The solution is feasible with respect to the relative `ConstraintTolerance` tolerance, but is not feasible with respect to the absolute tolerance.
`OptimalSolution`	1	The solver converged to a solution x.

Exit Flag for `linprog`	Numeric Equivalent	Meaning
`SolverLimitExceeded`	0	The number of iterations exceeds `options.MaxIterations`.
`NoFeasiblePointFound`	-2	No feasible point found.
`Unbounded`	-3	The problem is unbounded.
`FoundNaN`	-4	`NaN` value encountered during execution of the algorithm.
`PrimalDualInfeasible`	-5	Both primal and dual problems are infeasible.
`DirectionTooSmall`	-7	The search direction is too small. No further progress can be made.
`FeasibilityLost`	-9	Solver lost feasibility.

Exitflags 3 and -9 relate to solutions that have large infeasibilities. These usually arise from linear constraint matrices that have large condition number, or problems that have large solution components. To correct these issues, try to scale the coefficient matrices, eliminate redundant linear constraints, or give tighter bounds on the variables.

This table describes the exit flags for the `lsqlin` solver.

Exit Flag for `lsqlin`	Numeric Equivalent	Meaning
`FunctionChangeBelowTolerance`	3	Change in the residual is smaller than the specified tolerance `options.FunctionTolerance`. (`trust-region-reflective` algorithm)
`StepSizeBelowTolerance`	2	Step size smaller than `options.StepTolerance`, constraints satisfied. (`interior-point` algorithm)
`OptimalSolution`	1	The solver converged to a solution `x`.
`SolverLimitExceeded`	0	The number of iterations exceeds `options.MaxIterations`.
`NoFeasiblePointFound`	-2	For optimization problems, the problem is infeasible. Or, for the `interior-point` algorithm, step size smaller than `options.StepTolerance`, but constraints are not satisfied. For equation problems, no solution found.
`IllConditioned`	-4	Ill-conditioning prevents further optimization.
`NoDescentDirectionFound`	-8	The search direction is too small. No further progress can be made. (`interior-point` algorithm)

This table describes the exit flags for the `quadprog` solver.

Exit Flag for quadprog	Numeric Equivalent	Meaning
LocalMinimumFound	4	Local minimum found; minimum is not unique.
FunctionChangeBelowTolerance	3	Change in the objective function value is smaller than the specified tolerance `options.FunctionTolerance`. (`trust-region-reflective` algorithm)
StepSizeBelowTolerance	2	Step size smaller than `options.StepTolerance`, constraints satisfied. (`interior-point-convex` algorithm)
OptimalSolution	1	The solver converged to a solution x.
SolverLimitExceeded	0	The number of iterations exceeds `options.MaxIterations`.
NoFeasiblePointFound	-2	The problem is infeasible. Or, for the `interior-point` algorithm, step size smaller than `options.StepTolerance`, but constraints are not satisfied.
IllConditioned	-4	Ill-conditioning prevents further optimization.
Nonconvex	-6	Nonconvex problem detected. (`interior-point-convex` algorithm)
NoDescentDirectionFound	-8	Unable to compute a step direction. (`interior-point-convex` algorithm)

This table describes the exit flags for the `lsqcurvefit` or `lsqnonlin` solver.

Exit Flag for `lsqnonlin`	Numeric Equivalent	Meaning
SearchDirectionTooSmall	4	Magnitude of search direction was smaller than `options.StepTolerance`.
FunctionChangeBelowTolerance	3	Change in the residual was less than `options.FunctionTolerance`.
StepSizeBelowTolerance	2	Step size smaller than `options.StepTolerance`.
OptimalSolution	1	The solver converged to a solution x.
SolverLimitExceeded	0	Number of iterations exceeded `options.MaxIterations` or number of function evaluations exceeded `options.MaxFunctionEvaluations`.
OutputFcnStop	-1	Stopped by an output function or plot function.
NoFeasiblePointFound	-2	For optimization problems, problem is infeasible: the bounds lb and ub are inconsistent. For equation problems, no solution found.

This table describes the exit flags for the `fminunc` solver.

Exit Flag for `fminunc`	Numeric Equivalent	Meaning
`NoDecreaseAlongSearchDirection`	5	Predicted decrease in the objective function is less than the `options.FunctionTolerance` tolerance.
`FunctionChangeBelowTolerance`	3	Change in the objective function value is less than the `options.FunctionTolerance` tolerance.
`StepSizeBelowTolerance`	2	Change in `x` is smaller than the `options.StepTolerance` tolerance.
`OptimalSolution`	1	Magnitude of gradient is smaller than the `options.OptimalityTolerance` tolerance.
`SolverLimitExceeded`	0	Number of iterations exceeds `options.MaxIterations` or number of function evaluations exceeds `options.MaxFunctionEvaluations`.
`OutputFcnStop`	-1	Stopped by an output function or plot function.
`Unbounded`	-3	Objective function at current iteration is below `options.ObjectiveLimit`.

This table describes the exit flags for the `fmincon` solver.

Exit Flag for `fmincon`	Numeric Equivalent	Meaning
`NoDecreaseAlongSearchDirection`	5	Magnitude of directional derivative in search direction is less than 2*`options.OptimalityTolerance` and maximum constraint violation is less than `options.ConstraintTolerance`.
`SearchDirectionTooSmall`	4	Magnitude of the search direction is less than 2*`options.StepTolerance` and maximum constraint violation is less than `options.ConstraintTolerance`.
`FunctionChangeBelowTolerance`	3	Change in the objective function value is less than `options.FunctionTolerance` and maximum constraint violation is less than `options.ConstraintTolerance`.
`StepSizeBelowTolerance`	2	Change in `x` is less than `options.StepTolerance` and maximum constraint violation is less than `options.ConstraintTolerance`.
`OptimalSolution`	1	First-order optimality measure is less than `options.OptimalityTolerance`, and maximum constraint violation is less than `options.ConstraintTolerance`.

Exit Flag for fmincon	**Numeric Equivalent**	**Meaning**
SolverLimitExceeded	0	Number of iterations exceeds `options.MaxIterations` or number of function evaluations exceeds `options.MaxFunctionEvaluations`.
OutputFcnStop	-1	Stopped by an output function or plot function.
NoFeasiblePointFound	-2	No feasible point found.
Unbounded	-3	Objective function at current iteration is below `options.ObjectiveLimit` and maximum constraint violation is less than `options.ConstraintTolerance`.

This table describes the exit flags for the `fsolve` solver.

Exit Flag for fsolve	**Numeric Equivalent**	**Meaning**
SearchDirectionTooSmall	4	Magnitude of the search direction is less than `options.StepTolerance`, equation solved.
FunctionChangeBelowTolerance	3	Change in the objective function value is less than `options.FunctionTolerance`, equation solved.
StepSizeBelowTolerance	2	Change in x is less than `options.StepTolerance`, equation solved.
OptimalSolution	1	First-order optimality measure is less than `options.OptimalityTolerance`, equation solved.
SolverLimitExceeded	0	Number of iterations exceeds `options.MaxIterations` or number of function evaluations exceeds `options.MaxFunctionEvaluations`.
OutputFcnStop	-1	Stopped by an output function or plot function.
NoFeasiblePointFound	-2	Converged to a point that is not a root.
TrustRegionRadiusTooSmall	-3	Equation not solved. Trust region radius became too small (`trust-region-dogleg` algorithm).

This table describes the exit flags for the `fzero` solver.

Exit Flag for fzero	**Numeric Equivalent**	**Meaning**
OptimalSolution	1	Equation solved.
OutputFcnStop	-1	Stopped by an output function or plot function.
FoundNaNInfOrComplex	-4	`NaN`, `Inf`, or complex value encountered during search for an interval containing a sign change.

Exit Flag for `fzero`	Numeric Equivalent	Meaning
`SingularPoint`	-5	Might have converged to a singular point.
`CannotDetectSignChange`	-6	Did not find two points with opposite signs of function value.

`output` — Information about optimization process
structure

Information about the optimization process, returned as a structure. The output structure contains the fields in the relevant underlying solver output field, depending on which solver `solve` called:

*   `'fmincon'` output
*   `'fminunc'` output
*   `'fsolve'` output
*   `'fzero'` output
*   `'intlinprog'` output
*   `'linprog'` output
*   `'lsqcurvefit'` or `'lsqnonlin'` output
*   `'lsqlin'` output
*   `'lsqnonneg'` output
*   `'quadprog'` output

`solve` includes the additional field `Solver` in the `output` structure to identify the solver used, such as `'intlinprog'`.

`lambda` — Lagrange multipliers at the solution
structure

Lagrange multipliers at the solution, returned as a structure.

Note `solve` does not return `lambda` for equation-solving problems.

For the `intlinprog` and `fminunc` solvers, `lambda` is empty, `[]`. For the other solvers, `lambda` has these fields:

*   `Variables` – Contains fields for each problem variable. Each problem variable name is a structure with two fields:

    *   `Lower` – Lagrange multipliers associated with the variable `LowerBound` property, returned as an array of the same size as the variable. Nonzero entries mean that the solution is at the lower bound. These multipliers are in the structure `lambda.Variables.variablename.Lower`.
    *   `Upper` – Lagrange multipliers associated with the variable `UpperBound` property, returned as an array of the same size as the variable. Nonzero entries mean that the solution is at the upper bound. These multipliers are in the structure `lambda.Variables.variablename.Upper`.

- `Constraints` – Contains a field for each problem constraint. Each problem constraint is in a structure whose name is the constraint name, and whose value is a numeric array of the same size as the constraint. Nonzero entries mean that the constraint is active at the solution. These multipliers are in the structure `lambda.Constraints.`*`constraintname`*.

Note Elements of a constraint array all have the same comparison (<=, ==, or >=) and are all of the same type (linear, quadratic, or nonlinear).

Algorithms

Conversion to Solver Form

Internally, the `solve` function solves optimization problems by calling a solver:

- `linprog` for linear objective and linear constraints
- `intlinprog` for linear objective and linear constraints and integer constraints
- `quadprog` for quadratic objective and linear constraints
- `lsqlin` or `lsqnonneg` for linear least-squares with linear constraints
- `lsqcurvefit` or `lsqnonlin` for nonlinear least-squares with bound constraints
- `fminunc` for problems without any constraints (not even variable bounds) and with a general nonlinear objective function
- `fmincon` for problems with a nonlinear constraint, or with a general nonlinear objective and at least one constraint
- `fzero` for a scalar nonlinear equation
- `lsqlin` for systems of linear equations, with or without bounds
- `fsolve` for systems of nonlinear equations without constraints
- `lsqnonlin` for systems of nonlinear equations with bounds

Before `solve` can call these functions, the problems must be converted to solver form, either by `solve` or some other associated functions or objects. This conversion entails, for example, linear constraints having a matrix representation rather than an optimization variable expression.

The first step in the algorithm occurs as you place optimization expressions into the problem. An `OptimizationProblem` object has an internal list of the variables used in its expressions. Each variable has a linear index in the expression, and a size. Therefore, the problem variables have an implied matrix form. The `prob2struct` function performs the conversion from problem form to solver form. For an example, see "Convert Problem to Structure" on page 16-398.

For nonlinear optimization problems that use the `fmincon` or `fminunc` solvers, `solve` uses automatic differentiation to compute the gradients of the objective function and nonlinear constraint functions. These derivatives apply when the objective and constraint functions are composed of "Supported Operations on Optimization Variables and Expressions" on page 10-43 and do not use the `fcn2optimexpr` function. When automatic differentiation does not apply, solvers estimate derivatives using finite differences. For details of automatic differentiation, see "Automatic Differentiation Background" on page 10-37.

For the default and allowed solvers that `solve` calls, depending on the problem objective and constraints, see `'solver'`. You can override the default by using the `'solver'` name-value pair argument when calling `solve`.

For the algorithm that `intlinprog` uses to solve MILP problems, see "intlinprog Algorithm" on page 9-43. For the algorithms that `linprog` uses to solve linear programming problems, see "Linear Programming Algorithms" on page 9-2. For the algorithms that `quadprog` uses to solve quadratic programming problems, see "Quadratic Programming Algorithms" on page 11-2. For linear or nonlinear least-squares solver algorithms, see "Least-Squares (Model Fitting) Algorithms" on page 12-2. For nonlinear solver algorithms, see "Unconstrained Nonlinear Optimization Algorithms" on page 6-2 and "Constrained Nonlinear Optimization Algorithms" on page 6-19.

For nonlinear equation solving, `solve` internally represents each equation as the difference between the left and right sides. Then `solve` attempts to minimize the sum of squares of the equation components. For the algorithms for solving nonlinear systems of equations, see "Equation Solving Algorithms" on page 13-2. When the problem also has bounds, `solve` calls `lsqnonlin` to minimize the sum of squares of equation components. See "Least-Squares (Model Fitting) Algorithms" on page 12-2.

Note If your objective function is a sum of squares, and you want `solve` to recognize it as such, write it as `sum(expr.^2)`, and not as `expr'*expr` or any other form. The internal parser recognizes only explicit sums of squares. For details, see "Write Objective Function for Problem-Based Least Squares" on page 12-85. For an example, see "Nonnegative Linear Least Squares, Problem-Based" on page 12-40.

Automatic Differentiation

Automatic differentiation (AD) applies to the `solve` and `prob2struct` functions under the following conditions:

- The objective and constraint functions are supported, as described in "Supported Operations on Optimization Variables and Expressions" on page 10-43. They do not require use of the `fcn2optimexpr` function.
- The solver called by `solve` is either `fmincon` or `fminunc`.
- The `'ObjectiveDerivative'` and `'ConstraintDerivative'` name-value pair arguments for `solve` or `prob2struct` are set to their default value, `'auto'`.

When AD Applies	All Constraint Functions Supported	One or More Constraints Not Supported
Objective Function Supported	AD used for objective and constraints	AD used for objective only
Objective Function Not Supported	AD used for constraints only	AD not used

When these conditions are not satisfied, `solve` estimates gradients by finite differences, and `prob2struct` does not create gradients in its generated function files.

Note To use automatic derivatives in a problem converted by `prob2struct`, pass options specifying these derivatives.

```
options = optimoptions('fmincon','SpecifyObjectiveGradient',true,...
    'SpecifyConstraintGradient',true);
problem.options = options;
```

Currently, AD works only for first derivatives; it does not apply to second or higher derivatives. So, for example, if you want to use an analytic Hessian to speed your optimization, you cannot use `solve` directly, and must instead use the approach described in "Supply Derivatives in Problem-Based Workflow" on page 7-26.

Compatibility Considerations

`solve(prob,solver)`, `solve(prob,options)`, and `solve(prob,solver,options)` syntaxes have been removed
Errors starting in R2018b

To choose options or the underlying solver for `solve`, use name-value pairs. For example,

```
sol = solve(prob,'options',opts,'solver','quadprog');
```

The previous syntaxes were not as flexible, standard, or extensible as name-value pairs.

See Also
`EquationProblem` | `OptimizationProblem` | `evaluate` | `fcn2optimexpr` | `optimoptions` | `prob2struct`

Topics
"Problem-Based Optimization Workflow" on page 10-2
"Problem-Based Workflow for Solving Equations" on page 10-4
"Create Initial Point for Optimization with Named Index Variables" on page 10-50

Introduced in R2017b

varindex

Package: optim.problemdef

Map problem variables to solver-based variable index

Syntax

```
idx = varindex(prob)
idx = varindex(prob,varname)
```

Description

`idx = varindex(prob)` returns the linear indices of problem variables as a structure or an integer vector. If you convert `prob` to a problem structure by using `prob2struct`, `idx` gives the variable indices in the resulting problem structure that correspond to the variables in `prob`.

`idx = varindex(prob,varname)` returns the linear indices of elements of `varname`.

Examples

Obtain Problem Indices

Create an optimization problem.

```
x = optimvar('x',3);
y = optimvar('y',3,3);
prob = optimproblem('Objective',x'*y*x);
```

Convert the problem to a structure.

```
problem = prob2struct(prob);
```

Obtain the linear indices in `problem` of all `prob` variables.

```
idx = varindex(prob);
disp(idx.x)

     1     2     3

disp(idx.y)

     4     5     6     7     8     9    10    11    12
```

Obtain the y indices only.

```
idxy = varindex(prob,'y')

idxy = 1×9

     4     5     6     7     8     9    10    11    12
```

Input Arguments

`prob` — Optimization problem or equation problem
`OptimizationProblem` object | `EquationProblem` object

Optimization problem or equation problem, specified as an `OptimizationProblem` object or an `EquationProblem` object. Create an optimization problem by using `optimproblem`; create an equation problem by using `eqnproblem`.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

Example: `prob = optimproblem; prob.Objective = obj; prob.Constraints.cons1 = cons1;`

Example: `prob = eqnproblem; prob.Equations = eqs;`

`varname` — Variable name
character vector | string

Variable name, specified as a character vector or string.

Example: `'x'`

Data Types: `char` | `string`

Output Arguments

`idx` — Linear indices of problem variables
structure | integer vector

Linear indices of problem variables, returned as a structure or an integer vector. If you convert `prob` to a problem structure by using `prob2struct`, `idx` gives the variable indices in the resulting problem structure that correspond to the variables in `prob`.

- When you call `idx = varindex(prob)`, the returned `idx` is a structure. The field names of the structure are the variable names in `prob`. The value for each field is the integer vector of linear indices to which the variables map in the associated solver-based problem variable.

- When you call `idx = varindex(prob,varname)`, the returned `idx` is the vector of linear indices to which the variable `varname` maps in the associated solver-based problem variable.

See "Obtain Problem Indices" on page 16-475.

See Also

`EquationProblem` | `OptimizationProblem` | `prob2struct`

Topics
"Output Function for Problem-Based Optimization" on page 7-38
"Supply Derivatives in Problem-Based Workflow" on page 7-26

Introduced in R2019a

write

Package: optim.problemdef

Save optimization object description

Syntax

```
write(obj)
write(obj,filename)
```

Description

Use `write` to save the description of an optimization object.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`write(obj)` saves a description of the optimization object `obj` in a file named *obj*`.txt`. Here, *obj* is the workspace variable name of the optimization object. If `write` cannot construct the file name from the expression, it writes the description to `WriteOutput.txt` instead. `write` overwrites any existing file. If the object description is small, consider using `show` instead to display the description at the command line.

`write(obj,filename)` saves a description of `obj` in a file named `filename`.

Examples

Save Expression Description

Create an optimization variable and an expression that uses the variable. Save a description of the expression to a file.

```
x = optimvar('x',3,3);
A = magic(3);
var = sum(sum(A.*x));
write(var)
```

`write` creates a file named `var.txt` in the current folder. The file contains the following text:

```
  8*x(1, 1) + 3*x(2, 1) + 4*x(3, 1) + x(1, 2) + 5*x(2, 2) + 9*x(3, 2) + 6*x(1, 3) + 7*x(2, 3)
+ 2*x(3, 3)
```

Save the expression in a file named `'VarExpression.txt'` in the current folder.

```
write(var,"VarExpression.txt")
```

The `VarExpression.txt` file contains the same text as `var.txt`.

Input Arguments

obj — Optimization object
OptimizationProblem object | EquationProblem object | OptimizationExpression object | OptimizationVariable object | OptimizationConstraint object | OptimizationEquality object | OptimizationInequality object

Optimization object, specified as one of the following:

- `OptimizationProblem` object — `write(obj)` saves a file containing the variables for the solution, objective function, constraints, and variable bounds.
- `EquationProblem` object — `write(obj)` saves a file containing the variables for the solution, equations for the solution, and variable bounds.
- `OptimizationExpression` object — `write(obj)` saves a file containing the optimization expression.
- `OptimizationVariable` object — `write(obj)` saves a file containing the optimization variables. The saved description does not indicate variable types or bounds; it includes only the variable dimensions and index names (if any).
- `OptimizationConstraint` object — `write(obj)` saves a file containing the constraint expression.
- `OptimizationEquality` object — `write(obj)` saves a file containing the equality expression.
- `OptimizationInequality` object — `write(obj)` saves a file containing the inequality expression.

filename — Path to file
string | character vector

Path to the file, specified as a string or character vector. The path is relative to the current folder. The resulting file is a text file, so the file name typically has the extension `.txt`.

Example: `"../Notes/steel_stuff.txt"`

Data Types: `char` | `string`

See Also
show | writebounds

Topics
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2019b

writebounds

Package: `optim.problemdef`

Save description of variable bounds

Syntax

```
writebounds(var)
writebounds(var,filename)
```

Description

Use `writebounds` to save a description of the bounds on optimization variables.

Tip For the full workflow, see "Problem-Based Optimization Workflow" on page 10-2 or "Problem-Based Workflow for Solving Equations" on page 10-4.

`writebounds(var)` saves a description of the variable bounds in a file named *variable*`_bounds.txt`. Here, *variable* is the "Name" on page 16-0 property of `var`. The `writebounds` function overwrites any existing file.

`writebounds(var,filename)` saves a description of the variable bounds in a file named `filename`.

Examples

Save Description of Bounds

Create an optimization variable and save its bounds to a file.

```
x = optimvar('x',10,4,'LowerBound',randi(8,10,4),...
    'UpperBound',10+randi(7,10,4),'Type','integer');
writebounds(x,'BoundFile.txt')
```

The contents of `BoundFile.txt`:

```
        7 <= x(1, 1) <= 14
        8 <= x(2, 1) <= 13
        2 <= x(3, 1) <= 16
        8 <= x(4, 1) <= 16
        6 <= x(5, 1) <= 12
        1 <= x(6, 1) <= 14
        3 <= x(7, 1) <= 14
        5 <= x(8, 1) <= 15
        8 <= x(9, 1) <= 15
        8 <= x(10, 1) <= 16
        2 <= x(1, 2) <= 12
        8 <= x(2, 2) <= 15
        8 <= x(3, 2) <= 15
```

```
 4 <= x(4, 2) <= 12
 7 <= x(5, 2) <= 11
 2 <= x(6, 2) <= 14
 4 <= x(7, 2) <= 17
 8 <= x(8, 2) <= 13
 7 <= x(9, 2) <= 15
 8 <= x(10, 2) <= 12
 6 <= x(1, 3) <= 16
 1 <= x(2, 3) <= 12
 7 <= x(3, 3) <= 14
 8 <= x(4, 3) <= 15
 6 <= x(5, 3) <= 17
 7 <= x(6, 3) <= 17
 6 <= x(7, 3) <= 14
 4 <= x(8, 3) <= 11
 6 <= x(9, 3) <= 12
 2 <= x(10, 3) <= 12
 6 <= x(1, 4) <= 16
 1 <= x(2, 4) <= 12
 3 <= x(3, 4) <= 16
 1 <= x(4, 4) <= 12
 1 <= x(5, 4) <= 17
 7 <= x(6, 4) <= 13
 6 <= x(7, 4) <= 12
 3 <= x(8, 4) <= 12
 8 <= x(9, 4) <= 15
 1 <= x(10, 4) <= 14
```

Input Arguments

var — Optimization variable
`OptimizationVariable` object

Optimization variable, specified as an `OptimizationVariable` object. Create `var` using `optimvar`.

Example: `var = optimvar('var',4,6)`

filename — Path to file
string | character vector

Path to the file, specified as a string or character vector. The path is relative to the current folder. The resulting file is a text file, so the file name typically has the extension `.txt`.

Example: `"../Notes/steel_stuff.txt"`

Data Types: `char` | `string`

Tips

• To obtain the `writebounds` information at the Command Window, use `showbounds`.

See Also
`OptimizationVariable` | `showbounds`

Topics
"Problem-Based Optimization Setup"

Introduced in R2017b

writeconstr

Package: optim.problemdef

(Not recommended) Save optimization constraint description

Syntax

```
writeconstr(constr)
writeconstr(constr,filename)
```

Description

writeconstr is not recommended. Use write instead.

writeconstr(constr) saves a description of the optimization constraint constr in a file named *constr*.txt. Here, *constr* is the workspace variable name of the constraint. If writeconstr cannot construct the file name from the variable name, it writes the description to WriteConstrOutput.txt instead. writeconstr overwrites any existing file.

writeconstr(constr,filename) saves a description of the optimization constraint constr in a file named filename.

Examples

Save Constraint Description

Create an optimization constraint in terms of optimization variables, and save its description in a file.

```
x = optimvar('x',3,2);
cons = sum(x,2) <= [1;3;2];
writeconstr(cons,"TripleConstraint.txt")
```

The TripleConstraint.txt file contains the following text:

```
(1, 1)

  x(1, 1) + x(1, 2) <= 1

(2, 1)

  x(2, 1) + x(2, 2) <= 3

(3, 1)
```

```
x(3, 1) + x(3, 2) <= 2
```

Input Arguments

`constr` — Optimization constraint
OptimizationEquality object | OptimizationInequality object | OptimizationConstraint object

Optimization constraint, specified as an `OptimizationEquality` object, `OptimizationInequality` object, or `OptimizationConstraint` object. `constr` can represent a single constraint or an array of constraints.

Example: `constr = x + y <= 1` is a single constraint when x and y are scalar variables.

Example: `constr = sum(x) == 1` is an array of constraints when x is an array of two or more dimensions.

`filename` — Path to file
string | character vector

Path to the file, specified as a string or character vector. The path is relative to the current folder. The resulting file is a text file, so the file name typically has the extension `.txt`.

Example: `"../Notes/steel_stuff.txt"`

Data Types: `char` | `string`

Tips

- To obtain the `writeconstr` information at the MATLAB Command Window, use `showconstr`.

Compatibility Considerations

`writeconstr` is not recommended
Not recommended starting in R2019b

The `writeconstr` function is not recommended. Instead, use `write`. The `write` function replaces `writeconstr` and many other problem-based functions.

There are no plans to remove `writeconstr` at this time.

See Also
OptimizationConstraint | show | write

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

writeexpr

Package: optim.problemdef

(Not recommended) Save optimization expression description

Syntax

```
writeexpr(expr)
writeexpr(expr,filename)
```

Description

writeexpr is not recommended. Use write instead.

writeexpr(expr) saves a description of the optimization expression expr in a file named *expr*.txt. Here, *expr* is the workspace variable name of the expression. If writeexpr cannot construct the file name from the expression, it writes the description to WriteExprOutput.txt instead. writeexpr overwrites any existing file.

writeexpr(expr,filename) saves a description of the optimization expression expr in a file named filename.

Examples

Save Expression Description

Create an optimization variable and an expression that uses the variable. Save a description of the expression to a file.

```
x = optimvar('x',3,3);
A = magic(3);
var = sum(sum(A.*x));
writeexpr(var,"VarExpression.txt")
```

The VarExpression.txt file contains the following text:

```
  8*x(1, 1) + 3*x(2, 1) + 4*x(3, 1) + x(1, 2) + 5*x(2, 2) + 9*x(3, 2) + 6*x(1, 3) + 7*x(2, 3)
+ 2*x(3, 3)
```

Input Arguments

expr — Optimization expression
OptimizationExpression object

Optimization expression, specified as an OptimizationExpression object.

Example: sum(sum(x))

filename — Path to file
string | character vector

Path to the file, specified as a string or character vector. The path is relative to the current folder. The resulting file is a text file, so the file name typically has the extension `.txt`.

Example: `"../Notes/steel_stuff.txt"`

Data Types: `char` | `string`

Tips

- To obtain the `writeexpr` information at the MATLAB Command Window, use `showexpr`.

Compatibility Considerations

`writeexpr` is not recommended
Not recommended starting in R2019b

The `writeexpr` function is not recommended. Instead, use `write`. The `write` function replaces `writeexpr` and many other problem-based functions.

There are no plans to remove `writeexpr` at this time.

See Also
`OptimizationExpression` | `show` | `write`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

writeproblem

Package: optim.problemdef

(Not recommended) Save optimization problem description

Syntax

```
writeproblem(prob)
writeproblem(prob,filename)
```

Description

writeproblem is not recommended. Use write instead.

writeproblem(prob) saves a description of the optimization problem prob in a file named *prob*.txt. Here, *prob* is the workspace variable name of the problem. If writeproblem cannot construct the file name from the problem name, it writes to WriteProblemOutput.txt. The writeproblem function overwrites any existing file.

writeproblem(prob,filename) saves a description of the optimization problem prob in a file named filename.

Examples

Save Problem Description

Create an optimization problem.

```
x = optimvar('x');
y = optimvar('y');
prob = optimproblem;
prob.Objective = -x - y/3;
prob.Constraints.cons1 = x + y <= 2;
prob.Constraints.cons2 = x + y/4 <= 1;
prob.Constraints.cons3 = x - y <= 2;
prob.Constraints.cons4 = x/4 + y >= -1;
prob.Constraints.cons5 = x + y >= 1;
prob.Constraints.cons6 = -x + y <= 2;
```

Save the problem description in a file in the current directory.

```
writeproblem(prob,'ProblemDescription.txt')
```

The contents of ProblemDescription.txt:

```
    minimize :
       -x - 0.33333*y

    subject to cons1:
       x + y <= 2
```

```
   subject to cons2:
      x + 0.25*y <= 1

   subject to cons3:
      x - y <= 2

   subject to cons4:
      0.25*x + y >= -1

   subject to cons5:
      x + y >= 1

   subject to cons6:
      -x + y <= 2
```

Input Arguments

`prob` — Optimization problem or equation problem
OptimizationProblem object | EquationProblem object

Optimization problem or equation problem, specified as an OptimizationProblem object or an EquationProblem object. Create an optimization problem by using optimproblem; create an equation problem by using eqnproblem.

Warning The problem-based approach does not support complex values in an objective function, nonlinear equalities, or nonlinear inequalities. If a function calculation has a complex value, even as an intermediate value, the final result can be incorrect.

Example: prob = optimproblem; prob.Objective = obj; prob.Constraints.cons1 = cons1;

Example: prob = eqnproblem; prob.Equations = eqs;

`filename` — Path to file
string | character vector

Path to the file, specified as a string or character vector. The path is relative to the current folder. The resulting file is a text file, so the file name typically has the extension .txt.

Example: "../Notes/steel_stuff.txt"

Data Types: char | string

Tips

- writeproblem is equivalent to calling all of the following:

  - writeexpr(prob.Objective,filename)

  - writeconstr on each constraint in prob.Constraints

  - writebounds on all the variables in prob

- To obtain the writeproblem information at the Command Window, use showproblem.

Compatibility Considerations

`writeproblem` is not recommended
Not recommended starting in R2019b

The `writeproblem` function is not recommended. Instead, use `write`. The `write` function replaces `writeproblem` and many other problem-based functions.

There are no plans to remove `writeproblem` at this time.

See Also

`OptimizationProblem` | `show` | `write` | `writebounds`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b

writevar

Package: optim.problemdef

(Not recommended) Save optimization variable description

Syntax

```
writevar(var)
writevar(var,filename)
```

Description

writevar is not recommended. Use write instead.

writevar(var) saves a description of the optimization variable in a file named *variable*.txt. Here, *variable* is the "Name" on page 16-0   property of var. The writevar function overwrites any existing file.

writevar(var,filename) saves a description of the optimization variable in a file named filename.

Examples

Save Optimization Variable Description

Create an optimization variable and save its description in a file.

```
var = optimvar('var',8,3,'Type','integer');
writevar(var,"VariableDescription.txt")
```

The contents of VariableDescription.txt:

```
[ var(1, 1)      var(1, 2)      var(1, 3) ]
[ var(2, 1)      var(2, 2)      var(2, 3) ]
[ var(3, 1)      var(3, 2)      var(3, 3) ]
[ var(4, 1)      var(4, 2)      var(4, 3) ]
[ var(5, 1)      var(5, 2)      var(5, 3) ]
[ var(6, 1)      var(6, 2)      var(6, 3) ]
[ var(7, 1)      var(7, 2)      var(7, 3) ]
[ var(8, 1)      var(8, 2)      var(8, 3) ]
```

Input Arguments

var — Optimization variable
OptimizationVariable object

Optimization variable, specified as an OptimizationVariable object. Create var using optimvar.

Example: var = optimvar('var',4,6)

filename — Path to file
string | character vector

Path to the file, specified as a string or character vector. The path is relative to the current folder. The resulting file is a text file, so the file name typically has the extension `.txt`.

Example: `"../Notes/steel_stuff.txt"`

Data Types: `char` | `string`

Compatibility Considerations

writevar is not recommended
Not recommended starting in R2019b

The `writevar` function is not recommended. Instead, use `write`. The `write` function replaces `writevar` and many other problem-based functions.

There are no plans to remove `writevar` at this time.

See Also
`OptimizationVariable` | `optimvar` | `show` | `write`

Topics
"Problem-Based Optimization Setup"
"Problem-Based Optimization Workflow" on page 10-2

Introduced in R2017b