

# Optimization Toolbox™

## User's Guide



# MATLAB®

R2019a

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

## *Optimization Toolbox™ User's Guide*

© COPYRIGHT 1990–2019 by The MathWorks, Inc.

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

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

### **Trademarks**

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

### **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

November 1990	First printing	
December 1996	Second printing	For MATLAB® 5
January 1999	Third printing	For Version 2 (Release 11)
September 2000	Fourth printing	For Version 2.1 (Release 12)
June 2001	Online only	Revised for Version 2.1.1 (Release 12.1)
September 2003	Online only	Revised for Version 2.3 (Release 13SP1)
June 2004	Fifth printing	Revised for Version 3.0 (Release 14)
October 2004	Online only	Revised for Version 3.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 3.0.2 (Release 14SP2)
September 2005	Online only	Revised for Version 3.0.3 (Release 14SP3)
March 2006	Online only	Revised for Version 3.0.4 (Release 2006a)
September 2006	Sixth printing	Revised for Version 3.1 (Release 2006b)
March 2007	Seventh printing	Revised for Version 3.1.1 (Release 2007a)
September 2007	Eighth printing	Revised for Version 3.1.2 (Release 2007b)
March 2008	Online only	Revised for Version 4.0 (Release 2008a)
October 2008	Online only	Revised for Version 4.1 (Release 2008b)
March 2009	Online only	Revised for Version 4.2 (Release 2009a)
September 2009	Online only	Revised for Version 4.3 (Release 2009b)
March 2010	Online only	Revised for Version 5.0 (Release 2010a)
September 2010	Online only	Revised for Version 5.1 (Release 2010b)
April 2011	Online only	Revised for Version 6.0 (Release 2011a)
September 2011	Online only	Revised for Version 6.1 (Release 2011b)
March 2012	Online only	Revised for Version 6.2 (Release 2012a)
September 2012	Online only	Revised for Version 6.2.1 (Release 2012b)
March 2013	Online only	Revised for Version 6.3 (Release 2013a)
September 2013	Online only	Revised for Version 6.4 (Release 2013b)
March 2014	Online only	Revised for Version 7.0 (Release 2014a)
October 2014	Online only	Revised for Version 7.1 (Release 2014b)
March 2015	Online only	Revised for Version 7.2 (Release 2015a)
September 2015	Online only	Revised for Version 7.3 (Release 2015b)
March 2016	Online only	Revised for Version 7.4 (Release 2016a)
September 2016	Online only	Revised for Version 7.5 (Release 2016b)
March 2017	Online only	Revised for Version 7.6 (Release 2017a)
September 2017	Online only	Revised for Version 8.0 (Release 2017b)
March 2018	Online only	Revised for Version 8.1 (Release 2018a)
September 2018	Online only	Revised for Version 8.2 (Release 2018b)
March 2019	Online only	Revised for Version 8.3 (Release 2019a)



## Acknowledgments

<b>Acknowledgments</b> .....	<b>xxiv</b>
------------------------------	-------------

## Getting Started

### 1

<b>Optimization Toolbox Product Description</b> .....	<b>1-2</b>
Key Features .....	<b>1-2</b>
<b>First Choose Problem-Based or Solver-Based Approach</b> .....	<b>1-3</b>
<b>Solve a Constrained Nonlinear Problem, Problem-Based</b> .....	<b>1-5</b>
<b>Solve a Constrained Nonlinear Problem, Solver-Based</b> .....	<b>1-13</b>
Typical Optimization Problem .....	<b>1-13</b>
Problem Formulation: Rosenbrock's Function .....	<b>1-13</b>
Define the Problem in Toolbox Syntax .....	<b>1-14</b>
Run the Optimization .....	<b>1-16</b>
Interpret the Result .....	<b>1-20</b>
<b>Set Up a Linear Program, Solver-Based</b> .....	<b>1-23</b>
Convert a Problem to Solver Form .....	<b>1-23</b>
Model Description .....	<b>1-23</b>
Solution Method .....	<b>1-25</b>
Bibliography .....	<b>1-31</b>
<b>Set Up a Linear Program, Problem-Based</b> .....	<b>1-32</b>
Convert a Problem to Solver Form .....	<b>1-32</b>
Model Description .....	<b>1-33</b>

First Solution Method: Create Optimization Variable for Each Problem Variable .....	1-34
Create Problem and Objective .....	1-34
Create and Include Linear Constraints .....	1-34
Solve Problem .....	1-35
Examine Solution .....	1-36
Second Solution Method: Create One Optimization Variable and Indices .....	1-37
Set Variable Bounds .....	1-37
Create Problem, Linear Constraints, and Solution .....	1-38
Examine Indexed Solution .....	1-38
Bibliography .....	1-39

## Setting Up an Optimization

# 2

<b>Optimization Theory Overview .....</b>	<b>2-3</b>
<b>Optimization Toolbox Solvers .....</b>	<b>2-4</b>
<b>Optimization Decision Table .....</b>	<b>2-6</b>
<b>Choosing the Algorithm .....</b>	<b>2-8</b>
fmincon Algorithms .....	2-8
fsolve Algorithms .....	2-10
fminunc Algorithms .....	2-10
Least Squares Algorithms .....	2-11
Linear Programming Algorithms .....	2-12
Quadratic Programming Algorithms .....	2-13
Large-Scale vs. Medium-Scale Algorithms .....	2-14
Potential Inaccuracy with Interior-Point Algorithms .....	2-15
<b>Problems Handled by Optimization Toolbox Functions .....</b>	<b>2-17</b>
<b>Complex Numbers in Optimization Toolbox Solvers .....</b>	<b>2-20</b>
<b>Types of Objective Functions .....</b>	<b>2-22</b>
<b>Writing Scalar Objective Functions .....</b>	<b>2-23</b>
Function Files .....	2-23

Anonymous Function Objectives .....	2-25
Including Gradients and Hessians .....	2-25
<b>Writing Vector and Matrix Objective Functions .....</b>	<b>2-34</b>
What Are Vector or Matrix Objective Functions? .....	2-34
Jacobians of Vector Functions .....	2-34
Jacobians of Matrix Functions .....	2-35
Jacobians with Matrix-Valued Independent Variables .....	2-36
<b>Writing Objective Functions for Linear or Quadratic Problems</b> .....	<b>2-38</b>
<b>Maximizing an Objective .....</b>	<b>2-39</b>
<b>Matrix Arguments .....</b>	<b>2-40</b>
<b>Types of Constraints .....</b>	<b>2-41</b>
<b>Iterations Can Violate Constraints .....</b>	<b>2-42</b>
Intermediate Iterations can Violate Constraints .....	2-42
Algorithms That Satisfy Bound Constraints .....	2-42
Solvers and Algorithms That Can Violate Bound Constraints .....	2-42
<b>Bound Constraints .....</b>	<b>2-44</b>
<b>Linear Constraints .....</b>	<b>2-46</b>
Linear Inequality Constraints .....	2-46
Linear Equality Constraints .....	2-47
<b>Nonlinear Constraints .....</b>	<b>2-48</b>
Including Gradients in Constraint Functions .....	2-49
Anonymous Nonlinear Constraint Functions .....	2-50
<b>Or Instead of And Constraints .....</b>	<b>2-53</b>
<b>How to Use All Types of Constraints .....</b>	<b>2-58</b>
<b>Objective and Nonlinear Constraints in the Same Function .</b>	<b>2-60</b>
<b>Objective and Constraints Having a Common Function in Serial or Parallel, Problem-Based .....</b>	<b>2-65</b>

<b>Passing Extra Parameters</b> .....	<b>2-70</b>
Extra Parameters, Fixed Variables, or Data .....	<b>2-70</b>
Anonymous Functions .....	<b>2-70</b>
Nested Functions .....	<b>2-72</b>
Global Variables .....	<b>2-73</b>
<b>What Are Options?</b> .....	<b>2-74</b>
<b>Options in Common Use: Tuning and Troubleshooting</b> .....	<b>2-75</b>
<b>Set and Change Options</b> .....	<b>2-76</b>
<b>Choose Between <code>optimoptions</code> and <code>optimset</code></b> .....	<b>2-78</b>
<b>View Options</b> .....	<b>2-82</b>
<b>Tolerances and Stopping Criteria</b> .....	<b>2-84</b>
<b>Checking Validity of Gradients or Jacobians</b> .....	<b>2-87</b>
Check Gradient or Jacobian in Objective Function .....	<b>2-87</b>
How to Check Derivatives .....	<b>2-87</b>
Example: Checking Derivatives of Objective and Constraint Functions .....	<b>2-88</b>
<b>Bibliography</b> .....	<b>2-95</b>

## Examining Results

# 3

<b>Current Point and Function Value</b> .....	<b>3-2</b>
<b>Exit Flags and Exit Messages</b> .....	<b>3-3</b>
Exit Flags .....	<b>3-3</b>
Exit Messages .....	<b>3-4</b>
Enhanced Exit Messages .....	<b>3-5</b>
Exit Message Options .....	<b>3-8</b>
<b>Iterations and Function Counts</b> .....	<b>3-10</b>



<b>First-Order Optimality Measure</b> .....	<b>3-12</b>
What Is First-Order Optimality Measure? .....	<b>3-12</b>
Stopping Rules Related to First-Order Optimality .....	<b>3-12</b>
Unconstrained Optimality .....	<b>3-13</b>
Constrained Optimality Theory .....	<b>3-13</b>
Constrained Optimality in Solver Form .....	<b>3-15</b>
<b>Iterative Display</b> .....	<b>3-17</b>
Introduction .....	<b>3-17</b>
Common Headings .....	<b>3-18</b>
Function-Specific Headings .....	<b>3-18</b>
<b>Output Structures</b> .....	<b>3-26</b>
<b>Lagrange Multiplier Structures</b> .....	<b>3-27</b>
<b>Hessian</b> .....	<b>3-30</b>
fminunc Hessian .....	<b>3-30</b>
fmincon Hessian .....	<b>3-31</b>
<b>Plot Functions</b> .....	<b>3-33</b>
Plot an Optimization During Execution .....	<b>3-33</b>
Using a Plot Function .....	<b>3-33</b>
<b>Output Functions</b> .....	<b>3-39</b>
What Is an Output Function? .....	<b>3-39</b>
Example: Using Output Functions .....	<b>3-39</b>

## Steps to Take After Running a Solver

# 4

<b>Overview of Next Steps</b> .....	<b>4-2</b>
<b>When the Solver Fails</b> .....	<b>4-3</b>
Too Many Iterations or Function Evaluations .....	<b>4-3</b>
Converged to an Infeasible Point .....	<b>4-7</b>
Problem Unbounded .....	<b>4-9</b>
fsolve Could Not Solve Equation .....	<b>4-10</b>

<b>Solver Takes Too Long</b> .....	<b>4-11</b>
Enable Iterative Display .....	<b>4-11</b>
Use Appropriate Tolerances .....	<b>4-11</b>
Use a Plot Function .....	<b>4-12</b>
Use 'lbfgs' HessianApproximation Option .....	<b>4-12</b>
Enable CheckGradients .....	<b>4-12</b>
Use Inf Instead of a Large, Arbitrary Bound .....	<b>4-13</b>
Use an Output Function .....	<b>4-13</b>
Use a Sparse Solver or a Multiply Function .....	<b>4-13</b>
Use Parallel Computing .....	<b>4-14</b>
<b>When the Solver Might Have Succeeded</b> .....	<b>4-15</b>
Final Point Equals Initial Point .....	<b>4-15</b>
Local Minimum Possible .....	<b>4-15</b>
<b>When the Solver Succeeds</b> .....	<b>4-22</b>
What Can Be Wrong If The Solver Succeeds? .....	<b>4-22</b>
1. Change the Initial Point .....	<b>4-23</b>
2. Check Nearby Points .....	<b>4-24</b>
3. Check your Objective and Constraint Functions .....	<b>4-25</b>
<b>Local vs. Global Optima</b> .....	<b>4-27</b>
Why Didn't the Solver Find the Smallest Minimum? .....	<b>4-27</b>
Searching for a Smaller Minimum .....	<b>4-28</b>
Basins of Attraction .....	<b>4-28</b>
<b>Optimizing a Simulation or Ordinary Differential Equation</b> .	<b>4-32</b>
What Is Optimizing a Simulation or ODE? .....	<b>4-32</b>
Potential Problems and Solutions .....	<b>4-32</b>
Bibliography .....	<b>4-37</b>

## Optimization App

# 5

<b>Optimization App</b> .....	<b>5-2</b>
Optimization App Basics .....	<b>5-2</b>
Specifying Certain Options .....	<b>5-8</b>
Importing and Exporting Your Work .....	<b>5-11</b>

<b>Optimization App Alternatives</b> .....	<b>5-15</b>
Optimize Without Using the App .....	<b>5-15</b>
Set Options Using Live Scripts .....	<b>5-15</b>
Set Options: Command Line or Standard Scripts .....	<b>5-17</b>
Choose Plot Functions .....	<b>5-19</b>
Pass Solver Arguments .....	<b>5-20</b>

## Nonlinear algorithms and examples

# 6

<b>Unconstrained Nonlinear Optimization Algorithms</b> .....	<b>6-2</b>
Unconstrained Optimization Definition .....	<b>6-2</b>
fminunc trust-region Algorithm .....	<b>6-2</b>
fminunc quasi-newton Algorithm .....	<b>6-5</b>
 <b>fminsearch Algorithm</b> .....	 <b>6-11</b>
 <b>fminunc Unconstrained Minimization</b> .....	 <b>6-14</b>
Step 1: Write a file objfun.m. ....	<b>6-14</b>
Step 2: Set options. ....	<b>6-14</b>
Step 3: Invoke fminunc using the options. ....	<b>6-14</b>
 <b>Minimization with Gradient and Hessian</b> .....	 <b>6-16</b>
Step 1: Write a file brownfgh.m that computes the objective function, the gradient of the objective, and the sparse tridiagonal Hessian matrix. ....	<b>6-16</b>
Step 2: Call a nonlinear minimization routine with a starting point xstart. ....	<b>6-16</b>
 <b>Minimization with Gradient and Hessian Sparsity Pattern</b> ..	 <b>6-18</b>
Step 1: Write a file brownfg.m that computes the objective function and the gradient of the objective. ....	<b>6-18</b>
Step 2: Call a nonlinear minimization routine with a starting point xstart. ....	<b>6-19</b>
 <b>Constrained Nonlinear Optimization Algorithms</b> .....	 <b>6-22</b>
Constrained Optimization Definition .....	<b>6-22</b>
fmincon Trust Region Reflective Algorithm .....	<b>6-22</b>
fmincon Active Set Algorithm .....	<b>6-27</b>
fmincon SQP Algorithm .....	<b>6-36</b>

fmincon Interior Point Algorithm .....	6-37
fminbnd Algorithm .....	6-40
fseminf Problem Formulation and Algorithm .....	6-41
<b>Tutorial for the Optimization Toolbox™ .....</b>	<b>6-44</b>
<b>Banana Function Minimization .....</b>	<b>6-61</b>
<b>Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™ .....</b>	<b>6-72</b>
<b>Nonlinear Inequality Constraints .....</b>	<b>6-79</b>
Step 1: Write a file objfun.m for the objective function. ....	6-79
Step 2: Write a file confun.m for the constraints. ....	6-79
Step 3: Invoke constrained optimization routine. ....	6-79
<b>Nonlinear Constraints with Gradients .....</b>	<b>6-81</b>
Step 1: Write a file for the objective function and gradient. ...	6-81
Step 2: Write a file for the nonlinear constraints and the gradients of the nonlinear constraints. ....	6-81
Step 3: Invoke the constrained optimization routine. ....	6-82
<b>fmincon Interior-Point Algorithm with Analytic Hessian ....</b>	<b>6-84</b>
<b>Linear or Quadratic Objective with Quadratic Constraints ..</b>	<b>6-90</b>
<b>Nonlinear Equality and Inequality Constraints .....</b>	<b>6-95</b>
Step 1: Write a file objfun.m. ....	6-95
Step 2: Write a file confuneq.m for the nonlinear constraints. .....	6-95
Step 3: Invoke constrained optimization routine. ....	6-96
<b>Optimization App with the fmincon Solver .....</b>	<b>6-97</b>
Step 1: Write a file objecfun.m for the objective function. ...	6-97
Step 2: Write a file nonlconstr.m for the nonlinear constraints. .....	6-97
Step 3: Set up and run the problem with the Optimization app. .....	6-98
<b>Minimization with Bound Constraints and Banded Preconditioner .....</b>	<b>6-102</b>
Step 1: Write a file tbroyfg.m that computes the objective function and the gradient of the objective .....	6-102

Step 2: Call a nonlinear minimization routine with a starting point xstart. . . . .	6-105
<b>Minimization with Linear Equality Constraints . . . . .</b>	<b>6-107</b>
Step 1: Write a file brownfgh.m that computes the objective function, the gradient of the objective, and the sparse tridiagonal Hessian matrix. . . . .	6-107
Step 2: Call a nonlinear minimization routine with a starting point xstart. . . . .	6-107
<b>Minimization with Dense Structured Hessian, Linear Equalities . . . . .</b>	<b>6-110</b>
Hessian Multiply Function for Lower Memory . . . . .	6-110
Step 1: Write a file brownvv.m that computes the objective function, the gradient, and the sparse part of the Hessian. . . . .	6-111
Step 2: Write a function to compute Hessian-matrix products for H given a matrix Y. . . . .	6-111
Step 3: Call a nonlinear minimization routine with a starting point and linear equality constraints. . . . .	6-112
Preconditioning . . . . .	6-114
<b>Symbolic Math Toolbox Calculates Gradients and Hessians . . . . .</b>	<b>6-115</b>
Create the Variables . . . . .	6-117
Include the Linear Constraints . . . . .	6-118
Create the Nonlinear Constraints, Their Gradients and Hessians . . . . .	6-120
Create the Objective Function, Its Gradient and Hessian . . . . .	6-121
Create the Objective Function File . . . . .	6-121
Create the Constraint Function File . . . . .	6-122
Generate the Hessian Files . . . . .	6-123
Run the Optimization . . . . .	6-123
Compare to Optimization Without Gradients and Hessians . . . . .	6-126
Clear the Symbolic Variable Assumptions . . . . .	6-128
<b>Using Symbolic Mathematics with Optimization Toolbox™ Solvers . . . . .</b>	<b>6-129</b>
<b>One-Dimensional Semi-Infinite Constraints . . . . .</b>	<b>6-141</b>
<b>Two-Dimensional Semi-Infinite Constraint . . . . .</b>	<b>6-145</b>

<b>Analyzing the Effect of Uncertainty Using Semi-Infinite Programming</b> .....	<b>6-148</b>
--	--------------

## **Nonlinear Problem-Based**

# 7

<b>Rational Objective Function, Problem-Based</b> .....	<b>7-2</b>
<b>Solve Constrained Nonlinear Optimization, Problem-Based</b> .....	<b>7-4</b>
<b>Convert Nonlinear Function to Optimization Expression</b> .....	<b>7-8</b>
<b>Constrained Electrostatic Nonlinear Optimization, Problem-Based</b> .....	<b>7-13</b>
<b>Problem-Based Nonlinear Minimization with Linear Constraints</b> .....	<b>7-20</b>
<b>Include Derivatives in Problem-Based Workflow</b> .....	<b>7-24</b>
Why Include Derivatives? .....	<b>7-24</b>
Create Optimization Problem .....	<b>7-24</b>
Convert Problem to Solver-Based Form .....	<b>7-25</b>
Calculate Derivatives and Keep Track of Variables .....	<b>7-25</b>
Edit the Objective and Constraint Files .....	<b>7-26</b>
Run Problem Using Two Methods .....	<b>7-27</b>
<b>Output Function for Problem-Based Optimization</b> .....	<b>7-31</b>

## **Multiobjective Algorithms and Examples**

# 8

<b>Multiobjective Optimization Algorithms</b> .....	<b>8-2</b>
Multiobjective Optimization Definition .....	<b>8-2</b>
Algorithms .....	<b>8-3</b>
<b>Compare fminimax and fminunc</b> .....	<b>8-8</b>

<b>Using fminimax with a Simulink Model</b> .....	<b>8-11</b>
<b>Signal Processing Using fgoalattain</b> .....	<b>8-15</b>
Step 1: Write a file filtmin.m .....	<b>8-16</b>
Step 2: Invoke optimization routine .....	<b>8-16</b>
<b>Generate and Plot a Pareto Front</b> .....	<b>8-19</b>
<b>Multi-Objective Goal Attainment Optimization</b> .....	<b>8-23</b>
<b>Minimax Optimization</b> .....	<b>8-31</b>

## **Linear Programming and Mixed-Integer Linear Programming**

# 9

<b>Linear Programming Algorithms</b> .....	<b>9-2</b>
Linear Programming Definition .....	<b>9-2</b>
Interior-Point linprog Algorithm .....	<b>9-2</b>
Interior-Point-Legacy Linear Programming .....	<b>9-8</b>
Dual-Simplex Algorithm .....	<b>9-11</b>
<b>Typical Linear Programming Problem</b> .....	<b>9-16</b>
<b>Maximize Long-Term Investments Using Linear Programming:     Solver-Based</b> .....	<b>9-18</b>
<b>Mixed-Integer Linear Programming Algorithms</b> .....	<b>9-33</b>
Mixed-Integer Linear Programming Definition .....	<b>9-33</b>
intlinprog Algorithm .....	<b>9-33</b>
<b>Tuning Integer Linear Programming</b> .....	<b>9-45</b>
Change Options to Improve the Solution Process .....	<b>9-45</b>
Some "Integer" Solutions Are Not Integers .....	<b>9-46</b>
Large Components Not Integer Valued .....	<b>9-46</b>
Large Coefficients Disallowed .....	<b>9-47</b>
<b>Mixed-Integer Linear Programming Basics: Solver-Based</b> ..	<b>9-48</b>
<b>Factory, Warehouse, Sales Allocation Model: Solver-Based</b> ..	<b>9-52</b>

<b>Traveling Salesman Problem: Solver-Based</b> .....	<b>9-64</b>
<b>Optimal Dispatch of Power Generators: Solver-Based</b> .....	<b>9-73</b>
<b>Mixed-Integer Quadratic Programming Portfolio Optimization: Solver-Based</b> .....	<b>9-86</b>
<b>Solve Sudoku Puzzles Via Integer Programming: Solver-Based</b> .....	<b>9-95</b>
<b>Office Assignments by Binary Integer Programming: Solver- Based</b> .....	<b>9-104</b>
<b>Cutting Stock Problem: Solver-Based</b> .....	<b>9-113</b>
<b>Factory, Warehouse, Sales Allocation Model: Problem-Based</b> .....	<b>9-119</b>
<b>Traveling Salesman Problem: Problem-Based</b> .....	<b>9-129</b>
<b>Optimal Dispatch of Power Generators: Problem-Based</b> ...	<b>9-138</b>
<b>Office Assignments by Binary Integer Programming: Problem- Based</b> .....	<b>9-149</b>
<b>Mixed-Integer Quadratic Programming Portfolio Optimization: Problem-Based</b> .....	<b>9-156</b>
<b>Cutting Stock Problem: Problem-Based</b> .....	<b>9-165</b>
<b>Solve Sudoku Puzzles Via Integer Programming: Problem- Based</b> .....	<b>9-171</b>

## Problem-Based Optimization

# 10

<b>Problem-Based Workflow</b> .....	<b>10-2</b>
<b>Optimization Expressions</b> .....	<b>10-4</b>
What Are Optimization Expressions? .....	<b>10-4</b>



Expressions for Objective Functions .....	10-4
Expressions for Constraints .....	10-6
Optimization Variables Have Handle Behavior .....	10-8
<b>Pass Extra Parameters in Problem-Based Approach .....</b>	<b>10-10</b>
<b>Review or Modify Optimization Problems .....</b>	<b>10-14</b>
Review Problem Using showproblem or writeproblem .....	10-14
Change Default Solver or Options .....	10-15
Correct a Misspecified Problem .....	10-17
Duplicate Variable Name .....	10-20
<b>Named Index for Optimization Variables .....</b>	<b>10-22</b>
Create Named Indices .....	10-22
Use Named Indices .....	10-23
View Solution with Index Variables .....	10-25
<b>Examine Optimization Solution .....</b>	<b>10-28</b>
Obtain Numeric Solution .....	10-28
Examine Solution Quality .....	10-29
Infeasible Solution .....	10-29
Solution Takes Too Long .....	10-30
<b>Create Efficient Optimization Problems .....</b>	<b>10-32</b>
<b>Separate Optimization Model from Data .....</b>	<b>10-34</b>
<b>Problem-Based Optimization Algorithms .....</b>	<b>10-36</b>
<b>Variables with Duplicate Names Disallowed .....</b>	<b>10-38</b>
<b>Expression Contains Inf or NaN .....</b>	<b>10-40</b>
<b>Supported Operations on Optimization Variables and</b>	
<b>Expressions .....</b>	<b>10-42</b>
Notation for Supported Operations .....	10-42
Operations Returning Optimization Expressions .....	10-42
Operations Returning Optimization Variables .....	10-43
Operations on Optimization Expressions .....	10-43
Operations Returning Constraint Expressions .....	10-44
Some Undocumented Operations Work on Optimization	
Variables and Expressions .....	10-44

<b>Mixed-Integer Linear Programming Basics: Problem-Based</b> .....	<b>10-45</b>
<b>Create Initial Point for Optimization with Named Index Variables</b> .....	<b>10-49</b>

## Quadratic Programming

# 11

<b>Quadratic Programming Algorithms</b> .....	<b>11-2</b>
Quadratic Programming Definition .....	<b>11-2</b>
interior-point-convex quadprog Algorithm .....	<b>11-2</b>
trust-region-reflective quadprog Algorithm .....	<b>11-9</b>
<b>Quadratic Minimization with Bound Constraints</b> .....	<b>11-15</b>
Step 1: Load the Hessian and define f, lb, and ub. ....	<b>11-15</b>
Step 2: Call a quadratic minimization routine with a starting point xstart. ....	<b>11-15</b>
<b>Quadratic Minimization with Dense, Structured Hessian</b> ..	<b>11-18</b>
Take advantage of a structured Hessian .....	<b>11-18</b>
Step 1: Decide what part of H to pass to quadprog as the first argument. ....	<b>11-19</b>
Step 2: Write a function to compute Hessian-matrix products for H. ....	<b>11-19</b>
Step 3: Call a quadratic minimization routine with a starting point. ....	<b>11-20</b>
Preconditioning .....	<b>11-21</b>
<b>Large Sparse Quadratic Program with Interior Point Algorithm</b> .....	<b>11-24</b>
<b>Bound-Constrained Quadratic Programming, Solver-Based</b> .....	<b>11-29</b>
<b>Quadratic Programming for Portfolio Optimization Problems, Solver-Based</b> .....	<b>11-34</b>
<b>Quadratic Programming with Bound Constraints: Problem- Based</b> .....	<b>11-42</b>

<b>Large Sparse Quadratic Program, Problem-Based</b> . . . . .	<b>11-46</b>
<b>Bound-Constrained Quadratic Programming, Problem-Based</b> . . . . .	<b>11-50</b>
<b>Quadratic Programming for Portfolio Optimization, Problem-  Based</b> . . . . .	<b>11-55</b>

## Least Squares

# 12

<b>Least-Squares (Model Fitting) Algorithms</b> . . . . .	<b>12-2</b>
Least Squares Definition . . . . .	<b>12-2</b>
Trust-Region-Reflective Least Squares . . . . .	<b>12-3</b>
Interior-Point Linear Least Squares . . . . .	<b>12-6</b>
Levenberg-Marquardt Method . . . . .	<b>12-7</b>
<b>Nonlinear Data-Fitting</b> . . . . .	<b>12-11</b>
<b>lsqnonlin with a Simulink Model</b> . . . . .	<b>12-21</b>
<b>Nonlinear Least Squares With and Without Jacobian</b> . . . . .	<b>12-27</b>
Problem definition and solution technique . . . . .	<b>12-27</b>
Step 1: Write a file myfun.m that computes the objective function values. . . . .	<b>12-27</b>
Step 2: Call the nonlinear least-squares routine. . . . .	<b>12-28</b>
Step 3: Include a Jacobian. . . . .	<b>12-28</b>
<b>Linear Least Squares with Bound Constraints</b> . . . . .	<b>12-31</b>
<b>Optimization App with the lsqin Solver</b> . . . . .	<b>12-33</b>
The Problem . . . . .	<b>12-33</b>
Setting Up the Problem . . . . .	<b>12-33</b>
<b>Jacobian Multiply Function with Linear Least Squares</b> . . . . .	<b>12-37</b>
<b>Large-Scale Constrained Linear Least-Squares, Solver-Based</b> . . . . .	<b>12-42</b>
<b>Shortest Distance to a Plane</b> . . . . .	<b>12-47</b>

<b>Nonnegative Least-Squares, Problem-Based</b> .....	<b>12-50</b>
<b>Large-Scale Constrained Linear Least-Squares, Problem-Based</b> .....	<b>12-54</b>
<b>Nonlinear Curve Fitting with lsqcurvefit</b> .....	<b>12-60</b>
<b>Fit a Model to Complex-Valued Data</b> .....	<b>12-62</b>
<b>Fit an Ordinary Differential Equation (ODE)</b> .....	<b>12-67</b>

## Systems of Equations

# 13

<b>Equation Solving Algorithms</b> .....	<b>13-2</b>
Equation Solving Definition .....	<b>13-2</b>
Trust-Region fsolve Algorithm .....	<b>13-2</b>
Trust-Region Dogleg Method .....	<b>13-5</b>
Levenberg-Marquardt Method .....	<b>13-7</b>
\ Algorithm .....	<b>13-7</b>
fzero Algorithm .....	<b>13-7</b>
<b>Nonlinear Equations with Analytic Jacobian</b> .....	<b>13-9</b>
Step 1: Write a file bananaobj.m to compute the objective function values and the Jacobian. ....	<b>13-10</b>
Step 2: Call the solve routine for the system of equations. .	<b>13-10</b>
<b>Nonlinear Equations with Finite-Difference Jacobian</b> .....	<b>13-12</b>
<b>Nonlinear Equations with Jacobian</b> .....	<b>13-14</b>
Step 1: Write a file nlsf1.m that computes the objective function values and the Jacobian. ....	<b>13-14</b>
Step 2: Call the solve routine for the system of equations. .	<b>13-14</b>
<b>Nonlinear Equations with Jacobian Sparsity Pattern</b> .....	<b>13-17</b>
Step 1: Write a file nlsf1a.m that computes the objective function values. ....	<b>13-17</b>
Step 2: Call the system of equations solve routine. ....	<b>13-18</b>

<b>Nonlinear Systems with Constraints</b> .....	<b>13-20</b>
Solve Equations with Inequality Constraints .....	<b>13-20</b>
Use Different Start Points .....	<b>13-21</b>
Use Different Algorithms .....	<b>13-21</b>
Use lsqnonlin with Bounds .....	<b>13-22</b>
Set Equations and Inequalities as fmincon Constraints ....	<b>13-23</b>

## Parallel Computing for Optimization

# 14

<b>What Is Parallel Computing in Optimization Toolbox?</b> .....	<b>14-2</b>
Parallel Optimization Functionality .....	<b>14-2</b>
Parallel Estimation of Gradients .....	<b>14-3</b>
Nested Parallel Functions .....	<b>14-4</b>
 <b>Using Parallel Computing in Optimization Toolbox</b> .....	<b>14-6</b>
Using Parallel Computing with Multicore Processors .....	<b>14-6</b>
Using Parallel Computing with a Multiprocessor Network ...	<b>14-7</b>
Deploy Parallel Optimization .....	<b>14-8</b>
Testing Parallel Computations .....	<b>14-8</b>
 <b>Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™</b> .....	<b>14-10</b>
 <b>Improving Performance with Parallel Computing</b> .....	<b>14-18</b>
Factors That Affect Speed .....	<b>14-18</b>
Factors That Affect Results .....	<b>14-18</b>
Searching for Global Optima .....	<b>14-19</b>

## Argument and Options Reference

# 15

<b>Function Input Arguments</b> .....	<b>15-2</b>
<b>Function Output Arguments</b> .....	<b>15-5</b>

<b>Optimization Options Reference</b> .....	<b>15-8</b>
Optimization Options .....	<b>15-8</b>
Hidden Options .....	<b>15-23</b>
<b>Current and Legacy Option Name Tables</b> .....	<b>15-31</b>
<b>Output Function Syntax</b> .....	<b>15-37</b>
What Are Output Functions? .....	<b>15-37</b>
Structure of the Output Function .....	<b>15-38</b>
Fields in optimValues .....	<b>15-38</b>
States of the Algorithm .....	<b>15-44</b>
Stop Flag .....	<b>15-45</b>
<b>Plot Function Syntax</b> .....	<b>15-47</b>
<b>intlinprog Output Function and Plot Function Syntax</b> .....	<b>15-48</b>
What Are Output Functions and Plot Functions? .....	<b>15-48</b>
Custom Function Syntax .....	<b>15-49</b>
optimValues Structure .....	<b>15-50</b>

## Functions — Alphabetical List

# 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” on page 1-2
- “First Choose Problem-Based or Solver-Based Approach” on page 1-3
- “Solve a Constrained Nonlinear Problem, Problem-Based” on page 1-5
- “Solve a Constrained Nonlinear Problem, Solver-Based” on page 1-13
- “Set Up a Linear Program, Solver-Based” on page 1-23
- “Set Up a Linear Program, Problem-Based” on page 1-32

## Optimization Toolbox Product Description

### Solve linear, quadratic, integer, and nonlinear optimization problems

Optimization Toolbox provides functions for finding parameters that minimize or maximize objectives while satisfying constraints. The toolbox includes solvers for linear programming (LP), mixed-integer linear programming (MILP), quadratic programming (QP), 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 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 can be used to find optimal solutions in applications such as portfolio optimization, resource allocation, and production planning and scheduling.

### Key Features

- Nonlinear and multiobjective optimization of smooth constrained and unconstrained problems
- Solvers for nonlinear least squares, constrained linear least squares, data fitting, and nonlinear equations
- Quadratic programming (QP) and linear programming (LP)
- Mixed-integer linear programming (MILP)
- Optimization modeling tools
- Graphical monitoring of optimization progress
- Gradient estimation acceleration (with Parallel Computing Toolbox™)

# First Choose Problem-Based or Solver-Based Approach

There are two approaches to solving optimization problems using Optimization Toolbox: problem-based and solver-based. Before you start to solve an optimization problem, you must first choose an approach.

---

**Note** The problem-based approach currently does not apply to:

- Equation-solving
- Nonlinear least-squares
- Multiobjective or semi-infinite programming problems

If you have a problem of these types, use the solver-based approach “Solver-Based Optimization Problem Setup”.

---

Here is a summary of the main differences between the two approaches.

Approaches	Characteristics
“Problem-Based Optimization Setup”	<b>Easier to create and debug</b>
	<b>Not for equation-solving or nonlinear least-squares</b>
	Represent the objective and constraints symbolically
	Solution time is longer because of translation time from problem to solver
	Does not directly allow inclusion of gradient or Hessian; see “Problem-Based Workflow” on page 7-24
	See the steps in “Problem-Based Workflow” on page 10-2  Basic linear example: “Mixed-Integer Linear Programming” on page 10-45 or the video Solve a Mixed-Integer Linear Program Optimization Modeling. Basic nonlinear example: “Solve a Nonlinear Optimization Problem, Problem-Based” on page 1-5.
“Solver-Based Optimization Problem Setup”	<b>Harder to create and debug</b>
	Represent the objective and constraints as functions or matrices
	Solution time is shorter because there is no translation time from problem to solver
Allows inclusion of gradient or Hessian	

Approaches	Characteristics
	To save memory in large problems, allows use of Hessian and Jacobian multiply function. See “Quadratic Minimization with Hessian” on page 11-18 or “Jacobian Multiply Function with Hessian” on page 12-37.
	See the steps in “Solver-Based Optimization Problem Setup” on page 1-13.
	Basic linear example: “Mixed-Integer Linear Programming” on page 9-48. Basic nonlinear example: “Solve a Constrained Problem with Solver-Based” on page 1-13.

## 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:

For all but polynomial or rational expressions, you must convert your nonlinear functions using `fcn2optimexpr`. See the last part of this example, "Alternative Formulation Using `fcn2optimexpr`," 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 1-13.

## Problem Formulation: Rosenbrock's Function

Consider the problem of minimizing Rosenbrock's function

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

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');

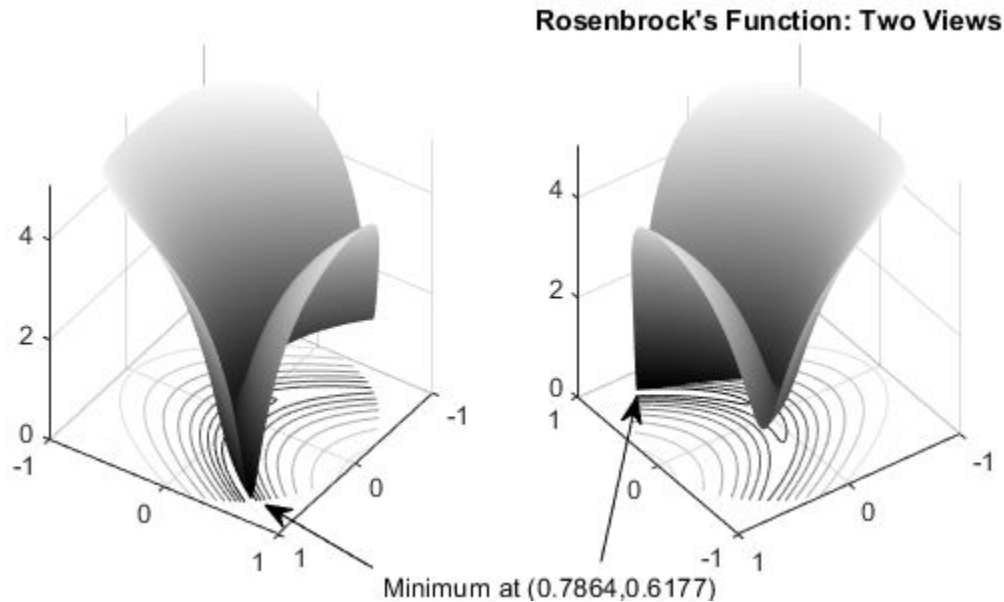
% Create contour
contour(X,Y,Z,'Parent',subplot2);

% 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" (MATLAB) 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  $x_1^2 + x_2^2 \leq 1$  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 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.

```
showproblem(prob)
```

```
OptimizationProblem :  
  
  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)
```

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: 6.9164e-06  
    algorithm: 'interior-point'  
    firstorderopt: 2.0934e-08  
    cgiterations: 4  
    message: '...'  
    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 4-22.

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 2-84.

### 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.

```
showproblem(convprob)
```

```
OptimizationProblem :  
  
  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 the same as before.

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

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: 6.9164e-06  
  algorithm: 'interior-point'  
  firstorderopt: 2.0934e-08  
  cgiterations: 4  
  message: '...'
```

```
solver: 'fmincon'
```

## See Also

### More About

- “Solve a Constrained Nonlinear Problem, Solver-Based” on page 1-13
- “First Choose Problem-Based or Solver-Based Approach” on page 1-3

## Solve a Constrained Nonlinear Problem, Solver-Based

### In this section...

“Typical Optimization Problem” on page 1-13

“Problem Formulation: Rosenbrock's Function” on page 1-13

“Define the Problem in Toolbox Syntax” on page 1-14

“Run the Optimization” on page 1-16

“Interpret the Result” on page 1-20

### Typical Optimization Problem

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

For the problem-based approach to this problem, see “Solve a Constrained Nonlinear Problem, Problem-Based” on page 1-5.

### Problem Formulation: Rosenbrock's Function

Consider the problem of minimizing Rosenbrock's function

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

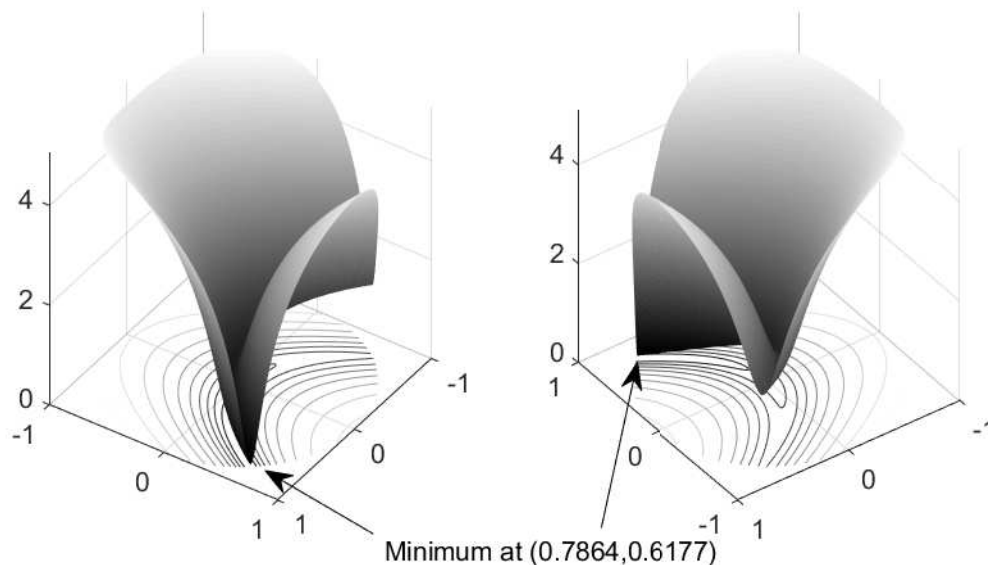
over the unit disk, that is, the disk of radius 1 centered at the origin. In other words, find  $x$  that minimizes the function  $f(x)$  over the set  $x_1^2 + x_2^2 \leq 1$ . This problem is a minimization of a nonlinear function with a nonlinear constraint.

---

**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.



### 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 equations.

All Optimization Toolbox optimization functions minimize an objective function. To maximize a function  $f$ , apply an optimization routine to minimize  $-f$ . For more details about maximizing, see “Maximizing an Objective” on page 2-39.

## Define the Problem in Toolbox Syntax

To use Optimization Toolbox software, 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.

### Function File for Objective 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” (MATLAB). 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 name `rosenbrock.m`.

### Function File for Constraint

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  $x_1^2 + x_2^2 - 1 \leq 0$ .

Furthermore, 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`.

### Run the Optimization

There are two ways to run the optimization:

- Use the Optimization app; see “Minimize Rosenbrock's Function Using the Optimization App” on page 1-16.
- Use command-line functions; see “Minimize Rosenbrock's Function at the Command Line” on page 1-20.

#### Minimize Rosenbrock's Function Using the Optimization App

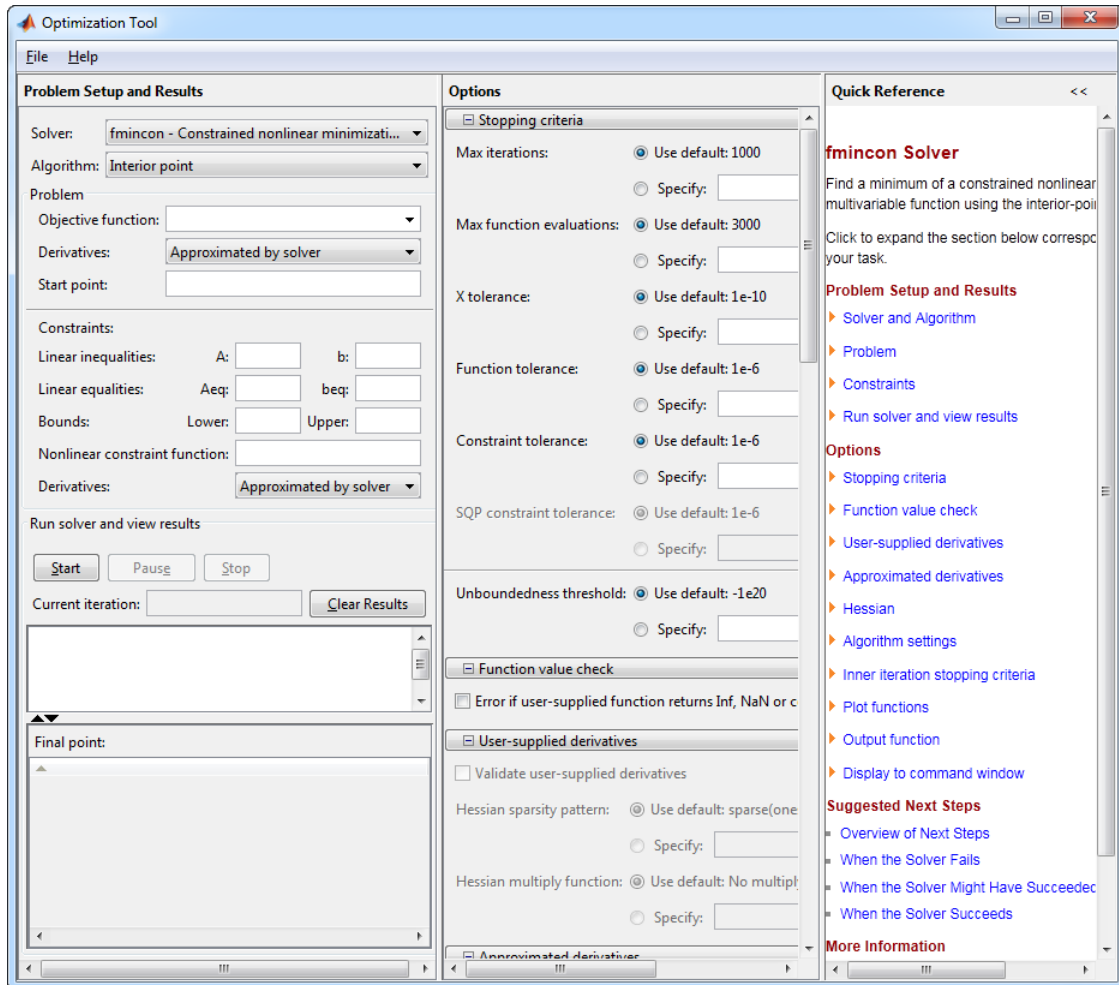
---

**Note** The Optimization app warns that it will be removed in a future release. For alternatives, see “Optimization App Alternatives” on page 5-15.

---

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





The default **Solver**, `fmincon - Constrained nonlinear minimization`, is selected. This solver is appropriate for this problem because Rosenbrock's function is nonlinear, and the problem has a constraint. For more information about choosing a solver, see “Optimization Decision Table” on page 2-6.

The default **Algorithm**, `Interior point`, is also selected.

- 2 In the **Objective function** box, enter `@rosenbrock`. The `@` character indicates the function handle (MATLAB) of the file `rosenbrock.m`.

- 3 In the **Start point** box, enter `[0 0]` to specify the initial point where `fmincon` begins its search for a minimum.
- 4 In the **Nonlinear constraint function** box, enter `@unitdisk`, the function handle of `unitdisk.m`.

Ensure that your **Problem Setup and Results** pane matches this figure.

**Problem Setup and Results**

Solver: `fmincon - Constrained nonlinear minimization` ▼

Algorithm: `Interior point` ▼

**Problem**

Objective function: `@rosenbrock` ▼

Derivatives: `Approximated by solver` ▼

Start point: `[0 0]`

**Constraints:**


Linear inequalities: A:  b:

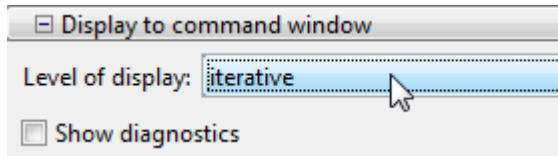
Linear equalities: Aeq:  beq:

Bounds: Lower:  Upper:

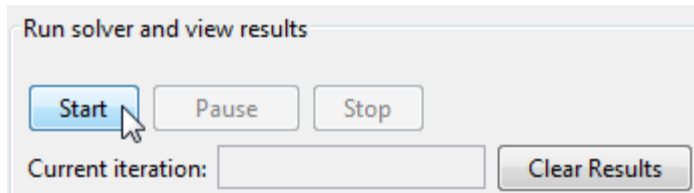
Nonlinear constraint function: `@unitdisk`

Derivatives: `Approximated by solver` ▼

- 5 In the **Options** pane, under **Display to command window** (at the bottom of the pane), select `iterative` from the **Level of display** list. (If you do not see the option, click  **Display to command window**.) This setting shows the progress of `fmincon` in the command window.



- 6 In the **Problem Setup and Results** pane, under **Run solver and view results**, click **Start**.



The following message appears in the **Run solver and view results** box:

```
Optimization running.
Objective function value: 0.04567482475812774
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.
```

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

The message tells you that:

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

At the bottom of the **Problem Setup and Results** pane, the minimizer  $x$  appears under **Final point**. For more information about exit messages, see “Exit Flags and Exit Messages” on page 3-3.

Final point:	
1 ▲	2
0.786	0.618

## Minimize Rosenbrock's Function at the Command Line

You can run the same optimization from the command line.

- 1 Create options that choose iterative display and the interior-point algorithm.

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

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

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

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

MATLAB outputs a table of iterations and the results of the optimization.

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the 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 message tells you that the search for a constrained optimum ended because the derivative of the objective function is nearly 0 in directions allowed by the constraint, and that the constraint is satisfied to the requisite accuracy. Several phrases in the message contain links to more information about the terms used in the message. For more details about these links, see “Enhanced Exit Messages” on page 3-5.

## Interpret the Result

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

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of 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	6.437e-05	3.974e-05
22	78	4.568281e-02	0.000e+00	8.000e-06	1.083e-07
23	81	4.567641e-02	0.000e+00	1.601e-06	2.793e-05
24	84	4.567482e-02	0.000e+00	1.996e-08	6.916e-06

Your table can differ, depending on toolbox version and computing platform. The following description applies to the table as displayed.

- The first column, labeled `Iter`, is the iteration number from 0 to 24. `fmincon` took 24 iterations to converge.
- The second column, labeled `F-count`, reports the cumulative number of times Rosenbrock's function was evaluated. The final row shows an `F-count` of 84, indicating that `fmincon` evaluated Rosenbrock's function 84 times in the process of finding a minimum.
- The third column, labeled `f(x)`, displays the value of the objective function. The final value, 0.04567482, is the minimum reported in the Optimization app **Run solver and view results** box, and at the end of the exit message in the command window.
- The fourth column, `Feasibility`, is 0 for all iterations. This column shows the value of the constraint function `unitdisk` at each iteration where the constraint is positive. 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 3-17.

## See Also

`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
- “Getting Started with Optimization Toolbox”
- “Solver-Based Optimization Problem Setup”

## Set Up a Linear Program, Solver-Based

### In this section...

“Convert a Problem to Solver Form” on page 1-23

“Model Description” on page 1-23

“Solution Method” on page 1-25

“Bibliography” on page 1-31

### 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 1-23 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 1-26.

### 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:

$$0.002614 \text{ HPS} + 0.0239 \text{ PP} + 0.009825 \text{ EP}.$$

The constraints are:

$$\begin{array}{rcll}
 2500 & & \leq & P1 & \leq & 6250 \\
 I1 & & & & \leq & 192,000 \\
 C & & & & \leq & 62,000 \\
 I1 & & - & HE1 & \leq & 132,000 \\
 I1 & = & LE1 & + & HE1 & + & C \\
 1359.8 & I1 & = & 1267.8 & HE1 & + & 1251.4 & LE1 & + & 192 & C & + & 3413 & P1 \\
 3000 & & & & \leq & P2 & \leq & 9000 \\
 I2 & & & & \leq & & & & & & & & & 244,000 \\
 LE2 & & & & \leq & & & & & & & & & 142,000 \\
 I2 & = & & & LE2 & + & & & & & & & & HE2 \\
 1359.8 & I2 & = & 1267.8 & HE2 & + & 1251.4 & LE2 & + & 3413 & P2 \\
 HPS & = & & I1 & + & & I2 & + & & & & & & BF1 \\
 HPS & = & & C & + & & MPS & + & & & & & & LPS \\
 LPS & = & & LE1 & + & & LE2 & + & & & & & & BF2 \\
 MPS & = & HE1 & + & HE2 & + & BF1 & - & & & & & & BF2 \\
 P1 & + & & P2 & + & & PP & \geq & & & & & & 24,550 \\
 EP & + & & & PP & \geq & & & & & & & & 12,000 \\
 MPS & & & & \geq & & & & & & & & & 271,536 \\
 LPS & & & & \geq & & & & & & & & & 100,623 \\
 \text{All} & & & \text{variables} & & & \text{are} & & & & & & & \text{positive.}
 \end{array}$$



## Solution Method

To solve the optimization problem, take the following steps.

1. “Choose a Solver” on page 1-25
2. “Combine Variables Into One Vector” on page 1-26
3. “Write Bound Constraints” on page 1-27
4. “Write Linear Inequality Constraints” on page 1-28
5. “Write Linear Equality Constraints” on page 1-29
6. “Write the Objective” on page 1-30
7. “Solve the Problem with linprog” on page 1-30
8. “Examine the Solution” on page 1-31

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 2-6. 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 2-17 or the `linprog` function reference page, the `linprog` solver solves problems of the form

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub. \end{cases} \quad (1-1)$$

- $f^T x$  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  $f$ .

- $A x \leq b$  represents linear inequalities.  $A$  is a  $k$ -by- $n$  matrix, where  $k$  is the number of inequalities and  $n$  is the number of variables (size of  $x$ ).  $b$  is a vector of length  $k$ . For more information, see “Linear Inequality Constraints” on page 2-46.

- $Aeq\ x = beq$  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 2-47.
- $lb \leq x \leq ub$  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 2-44.

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 1-23. 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 1-23. The lower bounds:

P1	$\geq$	2500
P2	$\geq$	3000
MPS	$\geq$	271,536
LPS	$\geq$	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 1-23:

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

In order to have the equations in the form  $Ax \leq b$ , put all the variables on the left side of the inequality. All these equations already have that form. Ensure that each inequality is in “less than” form by multiplying through by -1 wherever appropriate:

I1	-	HE1	≤	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 1-23:

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 1-32
- “Solver-Based Optimization Problem Setup”

## Set Up a Linear Program, Problem-Based

In this section...
“Convert a Problem to Solver Form” on page 1-32
“Model Description” on page 1-33
“First Solution Method: Create Optimization Variable for Each Problem Variable” on page 1-34
“Create Problem and Objective” on page 1-34
“Create and Include Linear Constraints” on page 1-34
“Solve Problem” on page 1-35
“Examine Solution” on page 1-36
“Second Solution Method: Create One Optimization Variable and Indices” on page 1-37
“Set Variable Bounds” on page 1-37
“Create Problem, Linear Constraints, and Solution” on page 1-38
“Examine Indexed Solution” on page 1-38
“Bibliography” on page 1-39

### Convert a 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]. 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 1-23 from the picture.
- 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.



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: Problem-Based Solution of a Mathematical Model.

## 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:

$$0.002614 \quad \text{HPS} \quad + \quad 0.0239 \quad \text{PP} \quad + \quad 0.009825 \quad \text{EP}.$$

The constraints are:

$$\begin{array}{rcll}
 2500 & & \leq & \text{P1} & \leq & 6250 \\
 \text{I1} & & & & \leq & 192,000 \\
 \text{C} & & & & \leq & 62,000 \\
 \text{I1} & & - & \text{HE1} & \leq & 132,000 \\
 \text{I1} & = & \text{LE1} & + & \text{HE1} & + & \text{C} \\
 1359.8 \quad \text{I1} & = & 1267.8 \quad \text{HE1} & + & 1251.4 \quad \text{LE1} & + & 192 \quad \text{C} & + & 3413 \quad \text{P1} \\
 3000 & & \leq & \text{P2} & \leq & 9000 \\
 \text{I2} & & & & \leq & 244,000 \\
 \text{LE2} & & & & \leq & 142,000 \\
 \text{I2} & = & \text{LE2} & + & & & \text{HE2} \\
 1359.8 \quad \text{I2} & = & 1267.8 \quad \text{HE2} & + & 1251.4 \quad \text{LE2} & + & 3413 \quad \text{P2} \\
 \text{HPS} & = & \text{I1} & + & \text{I2} & + & \text{BF1}
 \end{array}$$

HPS	=		C	+		+		LPS
LPS	=		LE1	+		+		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.

```
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.

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

## Create and Include Linear Constraints

There are three linear inequalities in the problem expressions:

$$\begin{array}{rcccccccl}
 \text{I1} & & - & & \text{HE1} & & \leq & & 132,000 \\
 \text{EP} & & + & & \text{PP} & & \geq & & 12,000 \\
 \text{P1} & + & & \text{P2} & + & \text{PP} & \geq & & 24,550.
 \end{array}$$

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;

```

There are eight linear equalities:

$$\begin{array}{rcccccccl}
 \text{I2} & & = & & \text{LE2} & & + & & \text{HE2} \\
 \text{LPS} & = & & \text{LE1} & + & \text{LE2} & + & & \text{BF2} \\
 \text{HPS} & = & & \text{I1} & + & \text{I2} & + & & \text{BF1} \\
 \text{HPS} & = & & \text{C} & + & \text{MPS} & + & & \text{LPS} \\
 \text{I1} & = & & \text{LE1} & + & \text{HE1} & + & & \text{C} \\
 \text{MPS} & = & \text{HE1} & + & \text{HE2} & + & \text{BF1} & - & \text{BF2} \\
 1359.8 \text{ I1} & = & 1267.8 \text{ HE1} & + & 1251.4 \text{ LE1} & + & 192 \text{ C} & + & 3413 \text{ P1} \\
 1359.8 \text{ I2} & = & 1267.8 \text{ HE2} & + & 1251.4 \text{ LE2} & + & 3413 \text{ P2} & & 
 \end{array}$$

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 could have asked for this value when you called 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

This table is too wide to see easily. Stack the variables to get them to a vertical orientation.

```
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 video Optimization Modeling, Part 2: Problem-Based Solution of a Mathematical Model gives interpretations of 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('I'  
linprob.Constraints.econs8 = 1267.8*x('HE2') + 1251.4*x('LE2') + 3413*x('P2') == 1359.8  
  
[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*. McGraw-Hill, New York, 1988.

## See Also

### More About

- “Set Up a Linear Program, Solver-Based” on page 1-23
- “Problem-Based Optimization Setup”





# Setting Up an Optimization

---

- “Optimization Theory Overview” on page 2-3
- “Optimization Toolbox Solvers” on page 2-4
- “Optimization Decision Table” on page 2-6
- “Choosing the Algorithm” on page 2-8
- “Problems Handled by Optimization Toolbox Functions” on page 2-17
- “Complex Numbers in Optimization Toolbox Solvers” on page 2-20
- “Types of Objective Functions” on page 2-22
- “Writing Scalar Objective Functions” on page 2-23
- “Writing Vector and Matrix Objective Functions” on page 2-34
- “Writing Objective Functions for Linear or Quadratic Problems” on page 2-38
- “Maximizing an Objective” on page 2-39
- “Matrix Arguments” on page 2-40
- “Types of Constraints” on page 2-41
- “Iterations Can Violate Constraints” on page 2-42
- “Bound Constraints” on page 2-44
- “Linear Constraints” on page 2-46
- “Nonlinear Constraints” on page 2-48
- “Or Instead of And Constraints” on page 2-53
- “How to Use All Types of Constraints” on page 2-58
- “Objective and Nonlinear Constraints in the Same Function” on page 2-60
- “Objective and Constraints Having a Common Function in Serial or Parallel, Problem-Based” on page 2-65
- “Passing Extra Parameters” on page 2-70
- “What Are Options?” on page 2-74
- “Options in Common Use: Tuning and Troubleshooting” on page 2-75
- “Set and Change Options” on page 2-76

- “Choose Between optimoptions and optimset” on page 2-78
- “View Options” on page 2-82
- “Tolerances and Stopping Criteria” on page 2-84
- “Checking Validity of Gradients or Jacobians” on page 2-87
- “Bibliography” on page 2-95

## Optimization Theory Overview

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

A General Problem (GP) description is stated as

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

subject to

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

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

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

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 2-20.

# Optimization Toolbox Solvers

There are four general categories of Optimization Toolbox solvers:

- Minimizers on page 2-17

This group of solvers attempts to find a local minimum of the objective function near a starting point  $x_0$ . They address problems of unconstrained optimization, linear programming, quadratic programming, and general nonlinear programming.

- Multiobjective minimizers on page 2-18

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

- Equation solvers on page 2-18

This group of solvers attempts to find a solution to a scalar- or vector-valued nonlinear equation  $f(x) = 0$  near a starting point  $x_0$ . Equation-solving can be considered a form of optimization because it is equivalent to finding the minimum norm of  $f(x)$  near  $x_0$ .

- Least-Squares (curve-fitting) solvers on page 2-19

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

For more information see “Problems Handled by Optimization Toolbox Functions” on page 2-17. See “Optimization Decision Table” on page 2-6 for aid in choosing among solvers for minimization.

Minimizers formulate optimization problems in the form

$$\min_x f(x),$$

possibly subject to constraints.  $f(x)$  is called an objective function. In general,  $f(x)$  is a scalar function of type `double`, and  $x$  is a vector or scalar of type `double`. However, multiobjective optimization, equation solving, and some sum-of-squares minimizers, can have vector or matrix objective functions  $F(x)$  of type `double`. To use Optimization Toolbox solvers for maximization instead of minimization, see “Maximizing an Objective” on page 2-39.

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 2-17.

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` function can address mixed-integer 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 ( $f = \text{const}$ , 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	*
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 2-17 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 2-8.

---

# Choosing the Algorithm

In this section...
"fmincon Algorithms" on page 2-8
"fsolve Algorithms" on page 2-10
"fminunc Algorithms" on page 2-10
"Least Squares Algorithms" on page 2-11
"Linear Programming Algorithms" on page 2-12
"Quadratic Programming Algorithms" on page 2-13
"Large-Scale vs. Medium-Scale Algorithms" on page 2-14
"Potential Inaccuracy with Interior-Point Algorithms" on page 2-15

## 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 4-15.

- To run an optimization again to obtain more speed on small- to medium-sized problems, try 'sqp' next, and 'active-set' last.
- Use 'trust-region-reflective' when applicable. Your problem must have: objective function includes gradient, only bounds, or only linear equality constraints (but not both).

See “Potential Inaccuracy with Interior-Point Algorithms” on page 2-15.

### 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 2-14. 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 2-14.
- '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 2-14.
- '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 2-14. 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 6-22.

### **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 4-15.
- 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 4-15.

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

## Least Squares Algorithms

### lsqin

`lsqin` has two algorithms:

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

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” (MATLAB). 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 12-37, try `'trust-region-reflective'`.

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

See “Potential Inaccuracy with Interior-Point Algorithms” on page 2-15.

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 has bounds, you must use 'trust-region-reflective'.
- If your problem has no bounds and is underdetermined (fewer equations than dimensions), use 'levenberg-marquardt'.

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

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 4-15.

See “Potential Inaccuracy with Interior-Point Algorithms” on page 2-15.

**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 two algorithms:

- 'interior-point-convex' (default)
- 'trust-region-reflective'

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” (MATLAB). 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'`.

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

See “Potential Inaccuracy with Interior-Point Algorithms” on page 2-15.

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 iterating 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 2-84).
- 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, equation solving, multiobjective optimization, and solving least-squares or data-fitting problems.

### Minimization Problems

Type	Formulation	Solver
Scalar minimization	$\min_x f(x)$ such that $lb < x < ub$ ( $x$ is scalar)	fminbnd
Unconstrained minimization	$\min_x f(x)$	fminunc, fminsearch
Linear programming	$\min_x f^T x$ such that $A \cdot x \leq b$ , $Aeq \cdot x = beq$ , $lb \leq x \leq ub$	linprog
Mixed-integer linear programming	$\min_x f^T x$ such that $A \cdot x \leq b$ , $Aeq \cdot x = beq$ , $lb \leq x \leq ub$ , $x(\text{intcon})$ is integer-valued.	intlinprog
Quadratic programming	$\min_x \frac{1}{2} x^T H x + c^T x$ such that $A \cdot x \leq b$ , $Aeq \cdot x = beq$ , $lb \leq x \leq ub$	quadprog
Constrained minimization	$\min_x f(x)$ 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	$\min_x f(x)$ 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 Problems**

Type	Formulation	Solver
Goal attainment	$\min_{x, \gamma}$ <p>such that <math>F(x) - w \cdot \gamma \leq \text{goal}</math>, <math>c(x) \leq 0</math>, <math>\text{ceq}(x) = 0</math>,  <math>A \cdot x \leq b</math>, <math>A_{\text{eq}} \cdot x = \text{beq}</math>, <math>lb \leq x \leq ub</math></p>	fgoalattain
Minimax	$\min_x \max_i F_i(x)$ <p>such that <math>c(x) \leq 0</math>, <math>\text{ceq}(x) = 0</math>, <math>A \cdot x \leq b</math>,  <math>A_{\text{eq}} \cdot x = \text{beq}</math>, <math>lb \leq x \leq ub</math></p>	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	$\min_x \frac{1}{2} \ C \cdot x - d\ _2^2$ <p><i>m</i> equations, <i>n</i> variables</p>	<code>mldivide</code> (matrix left division)
Nonnegative linear-least-squares	$\min_x \frac{1}{2} \ C \cdot x - d\ _2^2$ <p>such that <math>x \geq 0</math></p>	<code>lsqnonneg</code>
Constrained linear-least-squares	$\min_x \frac{1}{2} \ C \cdot x - d\ _2^2$ <p>such that <math>A \cdot x \leq b, Aeq \cdot x = beq, lb \leq x \leq ub</math></p>	<code>lsqlin</code>
Nonlinear least-squares	$\min_x \ F(x)\ _2^2 = \min_x \sum_i F_i^2(x)$ <p>such that <math>lb \leq x \leq ub</math></p>	<code>lsqnonlin</code>
Nonlinear curve fitting	$\min_x \ F(x, xdata) - ydata\ _2^2$ <p>such that <math>lb \leq x \leq ub</math></p>	<code>lsqcurvefit</code>

# 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.

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 12-62.

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 12-62

## 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 2-23
Nonlinear least squares	lsqcurvefit lsqnonlin	“Writing Vector and Matrix Objective Functions” on page 2-34
Multivariable equation solving	fsolve	
Multiobjective	fgoalattain fminimax	
Linear programming	linprog	“Writing Objective Functions for Linear or Quadratic Problems” on page 2-38
Mixed-integer linear programming	intlinprog	
Linear least squares	lsqlin	
	lsqnonneg	
Quadratic programming	quadprog	

## Writing Scalar Objective Functions

### In this section...

“Function Files” on page 2-23

“Anonymous Function Objectives” on page 2-25

“Including Gradients and Hessians” on page 2-25

### 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 2-40. A function file can return more outputs (see “Including Gradients and Hessians” on page 2-25).

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 2-70. For more complex examples of function files, see “Minimization with Gradient and Hessian Sparsity Pattern” on page 6-18 or “Minimization with Bound Constraints and Banded Preconditioner” on page 6-102.

### Local Functions and Nested Functions

Functions can exist inside other files as local functions (MATLAB) or nested functions (MATLAB). 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 2-72.

For example, suppose you want to minimize the `myObjective.m` objective function, described in “Function Files” on page 2-23, subject to the `ellipseparabola.m` constraint, described in “Nonlinear Constraints” on page 2-48. 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?” (MATLAB). 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 2-26
- “How to Include Gradients” on page 2-26
- “Including Hessians” on page 2-28
- “Benefits of Including Derivatives” on page 2-32

- “Choose Input Hessian Approximation for interior-point fmincon” on page 2-32

### 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 2-32. To also include second derivatives (Hessians), see “Including Hessians” on page 2-28.

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 2-29)
	sqp	Optional	No
	trust-region-reflective	Required	Optional (see “Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms” on page 2-28)
fminunc	quasi-newton	Optional	No
	trust-region	Required	Optional (see “Hessian for fminunc trust-region or fmincon trust-region-reflective algorithms” on page 2-28)

### 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 2-87.

---

**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

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

which is described and plotted in “Solve a Constrained Nonlinear Problem, Solver-Based” on page 1-13. The gradient of  $f(x)$  is

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

`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 nargin > 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” (MATLAB).

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 “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

### 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 2-28
- “Hessian for `fmincon` interior-point algorithm” on page 2-29
- “Hessian Multiply Function” on page 2-31

### 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 2-26)

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

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

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

For details, see “Constrained Optimality Theory” on page 3-13. The Hessian of the Lagrangian is

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

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  $x_1^2 + x_2^2 \leq 1$ , notice that the constraint function  $g(x) = x_1^2 + x_2^2 - 1 \leq 0$  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 6-84 and “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

### 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',...
    'SpecifyConstraintGradient',true,'SubproblemAlgorithm','cg','HessianMultiplyFcn',@HessMultFcn);
```

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 2-28). `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 6-110.

### 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 “Compare to Optimization Without Gradients and Hessians” on page 6-126.

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 2-48.

### 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



Hessian	Relative Memory Usage	Relative Efficiency
'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 2-28, “fmincon Interior-Point Algorithm with Analytic Hessian” on page 6-84, and “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

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 6-37, and its discussion of the LDL approach and the conjugate gradient approach to solving “Equation 6-52”.

## See Also

### More About

- “Checking Validity of Gradients or Jacobians” on page 2-87

## Writing Vector and Matrix Objective Functions

### In this section...

“What Are Vector or Matrix Objective Functions?” on page 2-34

“Jacobians of Vector Functions” on page 2-34

“Jacobians of Matrix Functions” on page 2-35

“Jacobians with Matrix-Valued Independent Variables” on page 2-36

### What Are Vector or 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 2-23 is the way to write their derivatives. The first-order partial derivatives of a vector-valued or matrix-valued function is called a Jacobian; the first-order partial derivatives of a scalar function is called a gradient.

For information on complex-valued objective functions, see “Complex Numbers in Optimization Toolbox Solvers” on page 2-20.

### 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 nargin > 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 your solver that your objective function includes a Jacobian, set the `SpecifyObjectiveGradient` option to `true`. For example,

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

## Jacobians of Matrix Functions

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

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

as a vector  $f$ :

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

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

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

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

For example, if

$$F(x) = \begin{bmatrix} x_1x_2 & x_1^3 + 3x_2^2 \\ 5x_2 - x_1^4 & x_2/x_1 \\ 4 - x_2^2 & x_1^3 - x_2^4 \end{bmatrix},$$

then the Jacobian of  $F$  is

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

## Jacobians with Matrix-Valued Independent Variables

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

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

then the gradient is defined in terms of the vector

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

With

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

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

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

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 2-87

## Writing Objective Functions for Linear or Quadratic Problems

The following solvers handle linear or quadratic objective functions:

- `linprog` and `intlinprog`: minimize

$$f'x = f(1)*x(1) + f(2)*x(2) + \dots + f(n)*x(n).$$

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

- `lsqlin` and `lsqnonneg`: minimize

$$\|Cx - d\|.$$

Input the matrix `C` and the vector `d` for the objective. See “Linear Least Squares with Bound Constraints” on page 12-31.

- `quadprog`: minimize

$$\begin{aligned} & \frac{1}{2} x' H x + f' x \\ = & \frac{1}{2} (x(1)*H(1,1)*x(1) + 2*x(1)*H(1,2)*x(2) + \dots \\ & + x(n)*H(n,n)*x(n)) + f(1)*x(1) + f(2)*x(2) + \dots + f(n)*x(n). \end{aligned}$$

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

## Maximizing an Objective

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

$$\max_x f(x),$$

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

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

```
[x fval] = fminunc(@(x)-tan(cos(x)),5)
Local minimum found.
```

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

```
x =
    6.2832
```

```
fval =
   -1.5574
```

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

# Matrix Arguments

Optimization Toolbox solvers accept vectors for many arguments, such as the initial point  $x_0$ , 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,  $x_0$  becomes  $x_0(:)$ . For an explanation of this syntax, see the  $A(:)$  entry in `colon`, or the "Indexing with a Single Index" section of "Array Indexing" (MATLAB).
- For output, solvers reshape the solution  $x$  to the same size as the input  $x_0$ .
- When  $x_0$  is a matrix, solvers pass  $x$  as a matrix of the same size as  $x_0$  to both the objective function and to any nonlinear constraint function.
- "Linear Constraints" on page 2-46, though, take  $x$  in vector form,  $x(:)$ . In other words, a linear constraint of the form

$$A*x \leq b \quad \text{or} \quad Aeq*x = beq$$

takes  $x$  as a vector, not a matrix. Ensure that your matrix  $A$  or  $Aeq$  has the same number of columns as  $x_0$  has elements, or the solver will error.

## See Also

`colon`

## More About

- "Writing Scalar Objective Functions" on page 2-23
- "Bound Constraints" on page 2-44
- "Linear Constraints" on page 2-46
- "Nonlinear Constraints" on page 2-48
- "Array Indexing" (MATLAB)



## Types of Constraints

Optimization Toolbox solvers have special forms for constraints:

- “Bound Constraints” on page 2-44 — Lower and upper bounds on individual components:  $x \geq l$  and  $x \leq u$ .
- “Linear Inequality Constraints” on page 2-46 —  $A \cdot x \leq b$ .  $A$  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 2-47 —  $Aeq \cdot x = beq$ . Equality constraints have the same form as inequality constraints.
- “Nonlinear Constraints” on page 2-48 —  $c(x) \leq 0$  and  $ceq(x) = 0$ . Both  $c$  and  $ceq$  are scalars or vectors representing several constraints.

Optimization Toolbox functions assume that inequality constraints are of the form  $c_i(x) \leq 0$  or  $A \cdot x \leq b$ . 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 2-46 and “Nonlinear Constraints” on page 2-48.

You can sometimes write constraints in several ways. For best results, use the lowest numbered constraints possible:

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

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

For information on how to pass extra parameters to constraint functions, see “Passing Extra Parameters” on page 2-70.

# Iterations Can Violate Constraints

In this section...
“Intermediate Iterations can Violate Constraints” on page 2-42
“Algorithms That Satisfy Bound Constraints” on page 2-42
“Solvers and Algorithms That Can Violate Bound Constraints” on page 2-42

## 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 2-44

## Bound Constraints

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

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

Give bounds as vectors with the same length as  $x$ , or as matrices on page 2-40 with the same number of elements as  $x$ .

- If a particular component has no lower bound, use `-Inf` as the bound; similarly, use `Inf` if a component has no upper bound.
- If you have only bounds of one type (upper or lower), you do not need to write the other type. For example, if you have no upper bounds, you do not need to supply a vector of `Infs`.
- If only the first  $m$  out of  $n$  components have bounds, then you need only supply a vector of length  $m$  containing bounds. However, this shortcut causes solvers to throw a warning.

For example, suppose your bounds are:

$$\begin{array}{rcl} x_3 & \geq & 8 \\ x_2 & \leq & 3. \end{array}$$

Write the constraint vectors as

$$\begin{array}{l} l \\ u \end{array} = \begin{array}{l} [-Inf; \\ [Inf; \quad 3] \end{array} \quad (\text{throws a warning}) \quad \text{or} \quad \begin{array}{l} -Inf; \quad 8 \\ [Inf; \quad 3; \quad Inf]. \end{array}$$

---

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

---

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

For a more complex example of bounds, see “Set Up a Linear Program, Solver-Based” on page 1-23.

## **See Also**

### **More About**

- “Iterations Can Violate Constraints” on page 2-42

## Linear Constraints

### In this section...

“Linear Inequality Constraints” on page 2-46

“Linear Equality Constraints” on page 2-47

### Linear Inequality Constraints

Several optimization solvers accept linear constraints, meaning 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 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, and linear equality constraints in the `Aeq` and `beq` arguments.

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

$$\begin{array}{rcccccccc}
 x_1 & & & + & & & x_3 & & \leq & & 4, \\
 2x_2 & & & - & & & x_3 & & \geq & & -2, \\
 x_1 & - & & & x_2 & + & & x_3 & - & x_4 & \geq & 9.
 \end{array}$$

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

Write these using the following matrix  $A$  and vector  $b$ :

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

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

Notice that the “greater than” inequalities were first multiplied by  $-1$  in order to get them into “less than” inequality form. In MATLAB syntax:

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

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

Even if you pass an initial point  $x_0$  as a matrix, solvers pass the current point  $x$  as a column vector to linear constraints. See “Matrix Arguments” on page 2-40.

For a more complex example of linear constraints, see “Set Up a Linear Program, Solver-Based” on page 1-23.

Intermediate iterations can violate linear constraints. See “Iterations Can Violate Constraints” on page 2-42.

## 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  $beq$ .

You do not need to give gradients for linear constraints; solvers calculate them automatically. Linear constraints do not affect Hessians. The form of this type of constraint is the same as for “Linear Inequality Constraints” on page 2-46.

## 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 are of the form  $ceq(x) = 0$ .

---

**Note** Nonlinear constraint functions must return both  $c$  and  $ceq$ , the inequality and equality constraint functions, even if they do not both exist. Return 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 function. Also, both inequalities were put into  $\leq 0$  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  $ceq$ , your solver can run faster and give more reliable results.

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

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

```

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

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

```

See “Writing Scalar Objective Functions” on page 2-23 for information on conditionalized functions. The gradient matrix has the form

$$\text{gradc}_i \quad j \quad = \quad [\partial c(j)/\partial x_i].$$

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

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

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

Make sure to pass the options structure to your solver:

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

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians automatically, as described in “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

### Anonymous Nonlinear Constraint Functions

For information on anonymous objective functions, see “Anonymous Function Objectives” on page 2-25.

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

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

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

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

Suppose that your nonlinear equality is

$$x_2 = \tanh(x_1).$$

Write a nonlinear constraint function as follows:

```

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

```

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

```

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

```

Local minimum found that satisfies the constraints.

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

```

z =
    -0.6530
    -0.5737

```

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

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

```

cout =
    -0.8704
         0

```

```

ceqout =
         0

```

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

## See Also

`GlobalSearch` | `MultiStart` | `fgoalattain` | `fmincon` | `ga` | `patternsearch`

### **More About**

- “Tutorial for the Optimization Toolbox™” on page 6-44
- “Nonlinear Constraints with Gradients” on page 6-81
- “Or Instead of And Constraints” on page 2-53
- “How to Use All Types of Constraints” on page 2-58

## 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 2-48: you give them as a set of functions that must be negative at a feasible point. If your constraints are

$$F_1(x) \leq 0 \quad \text{OR} \quad F_2(x) \leq 0 \quad \text{OR} \quad F_3(x) \leq 0,$$

then set the nonlinear inequality constraint function  $c(x)$  as:

$$c(x) = \min(F_1(x), F_2(x), F_3(x)).$$

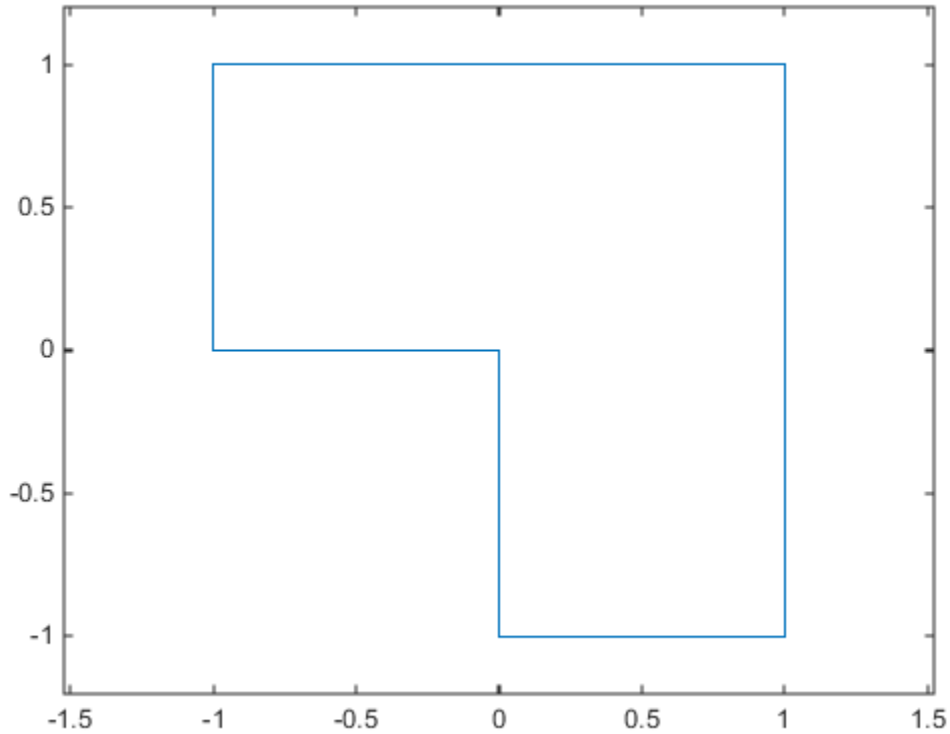
$c(x)$  is not smooth, which is a general requirement for constraint functions, due to the minimum. Nevertheless, the method often works.

---

**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  $a \leq x(1) \leq b$ ,  $c \leq x(2) \leq d$ :

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

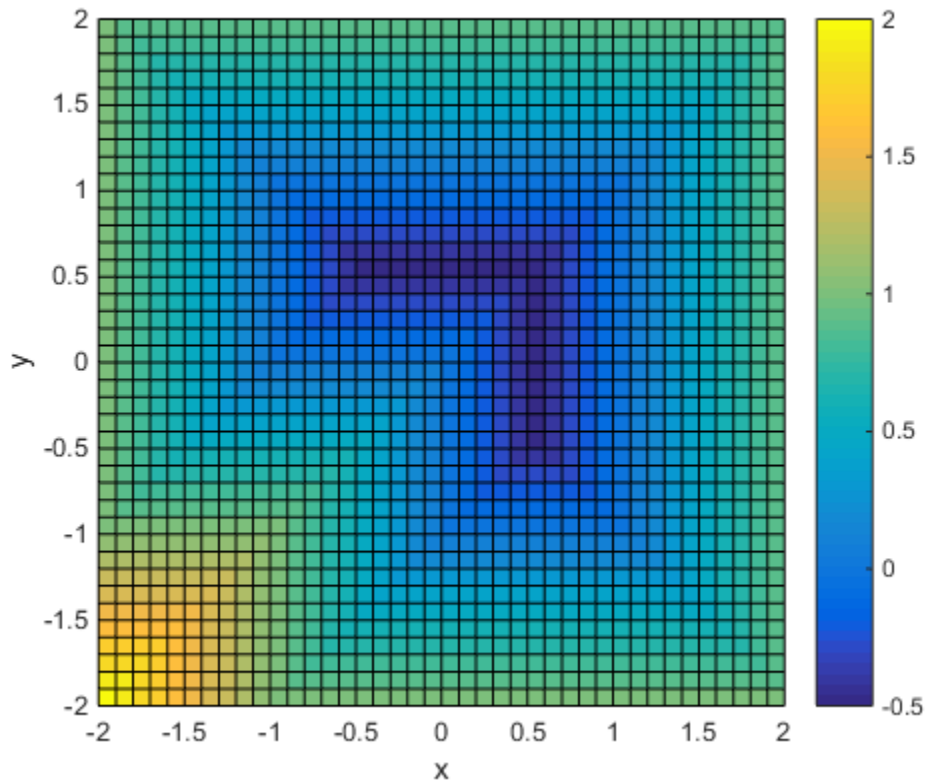
if (b <= a) || (d <= c)
    error('Give a rectangle a < b, c < d')
end

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

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

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

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



### Code for creating the figure

Plot `rectconstrfcn` over the region  $\max|x| \leq 2$  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 2-48

## How to Use All Types of Constraints

This section contains an example of a nonlinear minimization problem with all possible types of constraints. The objective function is in the local function `myobj(x)`. The nonlinear constraints are in the local function `myconstr(x)`. This example does not use gradients.

```
function [x fval exitflag] = fullexample
x0 = [1; 4; 5; 2; 5];
lb = [-Inf; -Inf; 0; -Inf; 1];
ub = [ Inf;  Inf; 20; Inf; Inf];
Aeq = [1 -0.3 0 0 0];
beq = 0;
A = [0 0 0 -1 0.1
     0 0 0 1 -0.5
     0 0 -1 0 0.9];
b = [0; 0; 0];
opts = optimoptions(@fmincon,'Algorithm','sqp');

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

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

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

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

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

Calling `fullexample` produces the following display in the Command Window:

```
[x fval exitflag] = fullexample;

Local minimum found that satisfies the constraints.

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

x =
    0.6114
    2.0380
```

```
1.3948
0.1572
1.5498

fval =
 37.3806

exitflag =
 1
```

## 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 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 2-65.

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

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

---

**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. Function that computes objective and constraints.

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

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

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

which has a unique minimum value of 0 at (1,1). See “Solve a Constrained Nonlinear Problem, Solver-Based” on page 1-13.

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

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

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

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

Suppose the objective function is

$$y = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 + 20*(x_3 - x_4^2)^2 + 5*(1 - x_4)^2.$$

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

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

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

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

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

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

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

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

end

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

### Step 3. Time to run with nested function.

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

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

Elapsed time is 203.797275 seconds.

### Step 4. Time to run without nested function.

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

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

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

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

```
tic
[x,fval,exitflag,output] = fmincon(@myrosen2,x0,...
```

```

                                [],[],[],[],[],[],@constr,opts);
toc

```

Elapsed time is 406.771978 seconds.

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

### Step 5. Save computing time with parallel computing.

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

```
parpool
```

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

```
ans =
```

```
Pool with properties:
```

```

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

```

```

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

```

Elapsed time is 97.528110 seconds.

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

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

```

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

```

Elapsed time is 188.985178 seconds.

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

## See Also

### More About

- “Objective and Constraints Having a Common Function in Serial or Parallel, Problem-Based” on page 2-65
- “Solve a Constrained Nonlinear Problem, Solver-Based” on page 1-13
- “Optimizing a Simulation or Ordinary Differential Equation” on page 4-32
- “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 2-60.

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;  
showproblem(prob)
```

```
OptimizationProblem :  
  
    minimize :  
    ((arg3 + (20 .* (x(3) - x(4).^2).^2)) + (5 .* (1 - x(4)).^2))  
  
where:  
  
    [arg3,~] = generatedFunction_computeall_withReuse(x);  
  
    subject to cons:  
    arg_LHS <= 0  
  
where:  
  
    [~,arg_LHS] = generatedFunction_computeall_withReuse(x);
```

### Solve Problem

Monitor the time it takes to solve the problem starting from the initial point  $x_0$ .  $x = [-1;1;1;2]$ .

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

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: [4x1 double]
```

```
fval = 0.7107
```

```
exitflag =
    OptimalSolution
```

```
output = struct with fields:
    iterations: 25
    funcCount: 149
    constrviolation: 0
    stepsize: 1.3715e-07
    algorithm: 'interior-point'
    firstorderopt: 4.0000e-07
    cgiterations: 7
    message: '↵Local minimum found that satisfies the constraints.↵Optimization terminated 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.'
    solver: 'fmincon'
```

```
time1 = toc
```

```
time1 = 149.8888
```

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", time1, funcCount);
```

The number of seconds to solve was 149.889, 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);
```

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 = 298.4074
```

### Parallel Processing

If you have a Parallel Processing 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);
```

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 = 74.0615
```

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);
```

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 = 144.6696
```

### 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=4x1 table
```

	Var1
Reuse Serial	149.89
No Reuse Serial	298.41
Reuse Parallel	74.062
No Reuse Parallel	144.67

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 2-60
- “Using Parallel Computing in Optimization Toolbox” on page 14-6

## 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 2-70
- “Nested Functions” on page 2-72
- “Global Variables” on page 2-73

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 10-10.

For example, suppose you want to minimize the function

$$f(x) = (a - bx_1^2 + x_1^4/3)x_1^2 + x_1x_2 + (-c + cx_2^2)x_2^2 \quad (2-3)$$

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 2-70.



## 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 14-18.

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 2-70.

## See Also

### More About

- “Solver-Based Optimization Problem Setup”
- “Pass Extra Parameters in Problem-Based Approach” on page 10-10

### 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 2-76.

For an overview of all options, including which solvers use each option, see “Optimization Options Reference” on page 15-8.

## 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 2-8 — `Algorithm`
- “Tolerances and Stopping Criteria” on page 2-84 — `OptimalityTolerance`, `StepTolerance`, `MaxFunctionEvaluations`, and `MaxIterations`
- Finite differences — `FiniteDifferenceType` and `FiniteDifferenceStepSize`

To diagnose and troubleshoot, try setting these options first:

- “Iterative Display” on page 3-17 — `Display`
- Function evaluation errors — `FunValCheck`
- “Plot Functions” on page 3-33 and “Output Functions” on page 3-39 — `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, to set the `fmincon` algorithm to `sqp`, set iterative display, and set a small value of 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 are the solvers that do not require an Optimization Toolbox license.

---

Change options as follows:

- Dot notation. For example,  
`options.StepTolerance = 1e-10;`
- `optimoptions`. For example,  
`options = optimoptions(options, 'StepTolerance', 1e-10);`
- Reset an option to default using `resetoptions`. For example,  
`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. For example,

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

---

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

## **See Also**

### **More About**

- “Optimization Options Reference” on page 15-8

### 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 2-76 and “View Options” on page 2-82.

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 15-23.

## See Also

### More About

- “Optimization Options Reference” on page 15-8

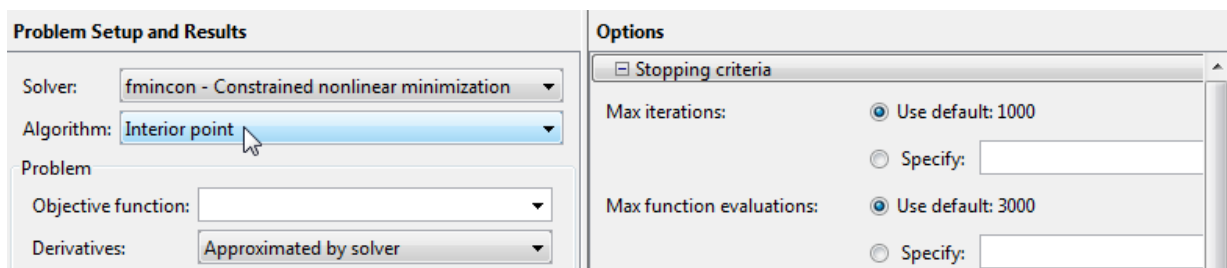
## Tolerances and Stopping Criteria

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 2-76.

**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,

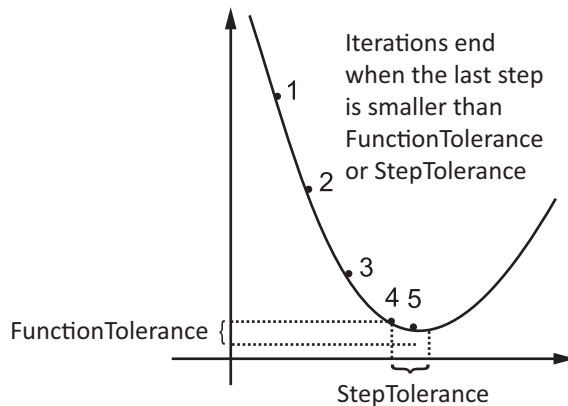
```
options = optimoptions('fmincon');
[options.OptimalityTolerance,options.FunctionTolerance,options.StepTolerance]
```

```
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  $|x_i - x_{i+1}| < \text{StepTolerance} * (1 + |x_i|)$ , or a similar relative measure.



- 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  $|f(x_i) - f(x_{i+1})| < \text{FunctionTolerance}$ , the iterations end. **FunctionTolerance** is generally used as a *relative* bound, meaning iterations end when  $|f(x_i) - f(x_{i+1})| < \text{FunctionTolerance} * (1 + |f(x_i)|)$ , or a similar relative measure.

---

**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. First-order optimality measure is defined in “First-Order Optimality Measure” on page 3-12.
- **ConstraintTolerance** is an upper bound on the magnitude of any constraint functions. If a solver returns a point  $x$  with  $c(x) > \text{ConstraintTolerance}$  or  $|ceq(x)| > \text{ConstraintTolerance}$ , the solver reports that the constraints are violated at  $x$ . **ConstraintTolerance** can also be a relative bound.

---

**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 3-10.

There are two other tolerances that apply to particular solvers: `TolPCG` and `MaxPCGIter`. These relate to preconditioned conjugate gradient steps. For more information, see “Preconditioned Conjugate Gradient Method” on page 6-24.

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 9-46 and “Branch and Bound” on page 9-40.

## See Also

### More About

- “Optimization Options Reference” on page 15-8

## Checking Validity of Gradients or Jacobians

### In this section...

“Check Gradient or Jacobian in Objective Function” on page 2-87

“How to Check Derivatives” on page 2-87

“Example: Checking Derivatives of Objective and Constraint Functions” on page 2-88

### 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  $x_0$ , 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 3-10.

### 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`.
- Using the Optimization app:

- 1 In the **Problem Setup and Results** pane, choose **Derivatives: Objective function: Gradient supplied** or **Nonlinear constraint function: Derivatives: Gradient supplied**. Make sure your objective or constraint functions supply the appropriate derivatives.
- 2 In the **Options** pane, check **User-supplied derivatives > Validate user-supplied derivatives**

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

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

### Example: Checking Derivatives of Objective and Constraint Functions

- “Objective and Constraint Functions” on page 2-88
- “Checking Derivatives at the Command Line” on page 2-89
- “Checking Derivatives with the Optimization App” on page 2-90

#### 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 1-13. 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 nargin > 1
    g = [-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
        200*(x(2)-x(1)^2)];

    if nargin > 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 nargin > 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 nargin > 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...
```

### Checking Derivatives with the Optimization App

---

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

---

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

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

**Problem Setup and Results**

Solver:

Algorithm:

Problem

Objective function:

Derivatives:

Start point:

Constraints:

Linear inequalities: A:  b:

Linear equalities: Aeq:  beq:

Bounds: Lower:  Upper:

Nonlinear constraint function:

Derivatives:

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

User-supplied derivatives

Validate user-supplied derivatives

Hessian sparsity pattern:  Use default: sparse(ones(numberOfVariables))  
 Specify:

Hessian multiply function:  Use default: No multiply function  
 Specify:

Approximated derivatives

Finite differences  $f(x + r^*x) - f(x)$

Type:

Relative perturbation vector r:  Use default: sqrt(eps)\*ones(numberOfVariables,1)  
 Specify:

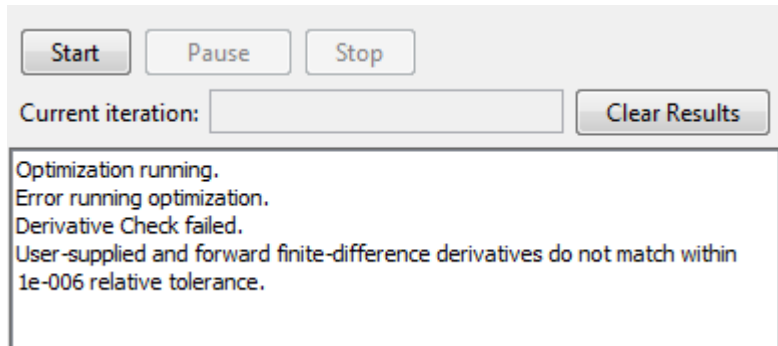
Minimum perturbation |r\*x|:  Use default: 0  
 Specify:

Maximum perturbation |r\*x|:  Use default: Inf  
 Specify:

Evaluate in parallel

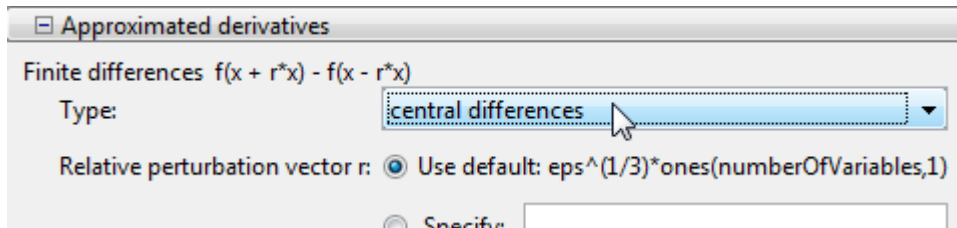
- 4 Press the **Start** button under **Run solver and view results**.

The output screen displays

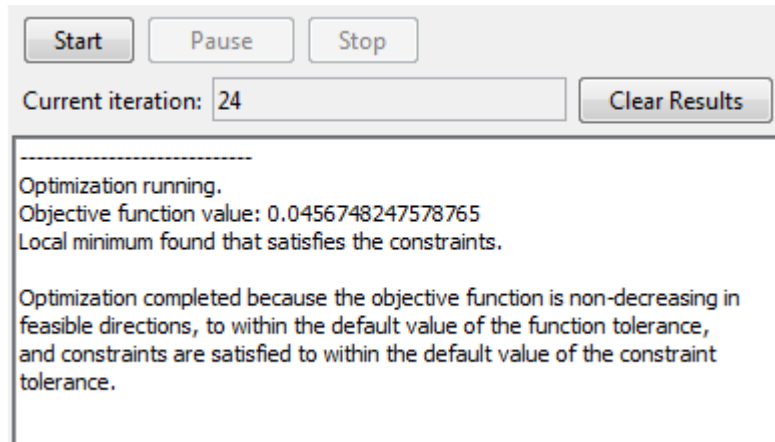


The forward finite difference approximation is inaccurate enough near  $[0 \ 0]$  that the derivative check fails.

- 5 To use the more accurate central differences, select central differences in the **Approximated derivatives > Type** pane:



- 6 Click **Run solver and view results > Clear Results**, then **Start**. This time the derivative check is successful:



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

## Bibliography

- [1] Biggs, M.C., "Constrained Minimization Using Recursive Quadratic Programming," *Towards Global Optimization* (L.C.W. Dixon and G.P. Szergo, eds.), North-Holland, pp 341-349, 1975.
- [2] Brayton, R.K., S.W. Director, G.D. Hachtel, and L. Vidigal, "A New Algorithm for Statistical Circuit Design Based on Quasi-Newton Methods and Function Splitting," *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, pp 784-794, Sept. 1979.
- [3] Broyden, C.G., "The Convergence of a Class of Double-rank Minimization Algorithms,"; *J. Inst. Maths. Applics.*, Vol. 6, pp 76-90, 1970.
- [4] Conn, N.R., N.I.M. Gould, and Ph.L. Toint, *Trust-Region Methods*, MPS/SIAM Series on Optimization, SIAM and MPS, 2000.
- [5] Dantzig, G., *Linear Programming and Extensions*, Princeton University Press, Princeton, 1963.
- [6] Dantzig, G.B., A. Orden, and P. Wolfe, "Generalized Simplex Method for Minimizing a Linear Form Under Linear Inequality Restraints," *Pacific Journal Math.*, Vol. 5, pp. 183-195, 1955.
- [7] Davidon, W.C., "Variable Metric Method for Minimization," *A.E.C. Research and Development Report*, ANL-5990, 1959.
- [8] Dennis, J.E., Jr., "Nonlinear least-squares," *State of the Art in Numerical Analysis* ed. D. Jacobs, Academic Press, pp 269-312, 1977.
- [9] Dennis, J.E., Jr. and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall Series in Computational Mathematics, Prentice-Hall, 1983.
- [10] Fleming, P.J., "Application of Multiobjective Optimization to Compensator Design for SISO Control Systems," *Electronics Letters*, Vol. 22, No. 5, pp 258-259, 1986.
- [11] Fleming, P.J., "Computer-Aided Control System Design of Regulators using a Multiobjective Optimization Approach," *Proc. IFAC Control Applications of Nonlinear Prog. and Optim.*, Capri, Italy, pp 47-52, 1985.

- [12] Fletcher, R., "A New Approach to Variable Metric Algorithms," *Computer Journal*, Vol. 13, pp 317-322, 1970.
- [13] Fletcher, R., "Practical Methods of Optimization," John Wiley and Sons, 1987.
- [14] Fletcher, R. and M.J.D. Powell, "A Rapidly Convergent Descent Method for Minimization," *Computer Journal*, Vol. 6, pp 163-168, 1963.
- [15] Forsythe, G.F., M.A. Malcolm, and C.B. Moler, *Computer Methods for Mathematical Computations*, Prentice Hall, 1976.
- [16] Gembicki, F.W., "Vector Optimization for Control with Performance and Parameter Sensitivity Indices," Ph.D. Thesis, Case Western Reserve Univ., Cleveland, Ohio, 1974.
- [17] Gill, P.E., W. Murray, M.A. Saunders, and M.H. Wright, "Procedures for Optimization Problems with a Mixture of Bounds and General Linear Constraints," *ACM Trans. Math. Software*, Vol. 10, pp 282-298, 1984.
- [18] Gill, P.E., W. Murray, and M.H. Wright, *Numerical Linear Algebra and Optimization*, Vol. 1, Addison Wesley, 1991.
- [19] Gill, P.E., W. Murray, and M.H. Wright, *Practical Optimization*, London, Academic Press, 1981.
- [20] Goldfarb, D., "A Family of Variable Metric Updates Derived by Variational Means," *Mathematics of Computing*, Vol. 24, pp 23-26, 1970.
- [21] Grace, A.C.W., "Computer-Aided Control System Design Using Optimization Techniques," Ph.D. Thesis, University of Wales, Bangor, Gwynedd, UK, 1989.
- [22] Han, S.P., "A Globally Convergent Method for Nonlinear Programming," *J. Optimization Theory and Applications*, Vol. 22, p. 297, 1977.
- [23] Hock, W. and K. Schittkowski, "A Comparative Performance Evaluation of 27 Nonlinear Programming Codes," *Computing*, Vol. 30, p. 335, 1983.
- [24] Hollingdale, S.H., *Methods of Operational Analysis in Newer Uses of Mathematics* (James Lighthill, ed.), Penguin Books, 1978.
- [25] Levenberg, K., "A Method for the Solution of Certain Problems in Least Squares," *Quart. Appl. Math.* Vol. 2, pp 164-168, 1944.



- [26] Madsen, K. and H. Schjaer-Jacobsen, "Algorithms for Worst Case Tolerance Optimization," *IEEE Transactions of Circuits and Systems*, Vol. CAS-26, Sept. 1979.
- [27] Marquardt, D., "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *SIAM J. Appl. Math.* Vol. 11, pp 431-441, 1963.
- [28] Moré, J.J., "The Levenberg-Marquardt Algorithm: Implementation and Theory," *Numerical Analysis*, ed. G. A. Watson, Lecture Notes in Mathematics 630, Springer Verlag, pp 105-116, 1977.
- [29] *NAG Fortran Library Manual*, Mark 12, Vol. 4, E04UAF, p. 16.
- [30] Nelder, J.A. and R. Mead, "A Simplex Method for Function Minimization," *Computer J.*, Vol.7, pp 308-313, 1965.
- [31] Nocedal, J. and S. J. Wright. *Numerical Optimization*, Second Edition. Springer Series in Operations Research, Springer Verlag, 2006.
- [32] Powell, M.J.D., "The Convergence of Variable Metric Methods for Nonlinearly Constrained Optimization Calculations," *Nonlinear Programming 3*, (O.L. Mangasarian, R.R. Meyer and S.M. Robinson, eds.), Academic Press, 1978.
- [33] Powell, M.J.D., "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations," *Numerical Analysis*, G.A.Watson ed., Lecture Notes in Mathematics, Springer Verlag, Vol. 630, 1978.
- [34] Powell, M.J.D., "A Fortran Subroutine for Solving Systems of Nonlinear Algebraic Equations," *Numerical Methods for Nonlinear Algebraic Equations*, (P. Rabinowitz, ed.), Ch.7, 1970.
- [35] Powell, M.J.D., "Variable Metric Methods for Constrained Optimization," *Mathematical Programming: The State of the Art*, (A. Bachem, M. Grottschel 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.

# Examining Results

---

- “Current Point and Function Value” on page 3-2
- “Exit Flags and Exit Messages” on page 3-3
- “Iterations and Function Counts” on page 3-10
- “First-Order Optimality Measure” on page 3-12
- “Iterative Display” on page 3-17
- “Output Structures” on page 3-26
- “Lagrange Multiplier Structures” on page 3-27
- “Hessian” on page 3-30
- “Plot Functions” on page 3-33
- “Output Functions” on page 3-39

# 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 3-5

“Exit Message Options” on page 3-8

### Exit Flags

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

- Positive exit flags correspond to successful outcomes.
- Negative exit flags correspond to unsuccessful outcomes.
- The zero exit flag corresponds to the solver being halted by exceeding an iteration limit or limit on the number of function evaluations (see “Iterations and Function Counts” on page 3-10, and also see “Tolerances and Stopping Criteria” on page 2-84).

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

---

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

---

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

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

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

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

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

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

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

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

Equation solved.

`fsolve` completed because the vector of function values is near zero as measured by the 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 `fsolve` `exitflag`, you find that an exit flag value 1 means “Function converged to a solution  $x$ .” In other words, `fsolve` reports `myfcn` is nearly zero at  $x = [0.5671 \ 0.5671]$ .

## Exit Messages

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

Many examples in this documentation show exit messages. For example, see “Minimize Rosenbrock’s Function at the Command Line” on page 1-20, or “Step 3: Invoke `fminunc`”



using the options.” on page 6-14. 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 gradi  
[xfinal fval exitflag] = fminunc(@sin,0,opts)
```

This yields the following results:

[Local minimum found.](#)

Optimization completed because the [size of the gradient](#) is less than the value of the [optimality tolerance](#).

[<stopping criteria details>](#)

xfinal =

-1.5708

fval =

-1.0000

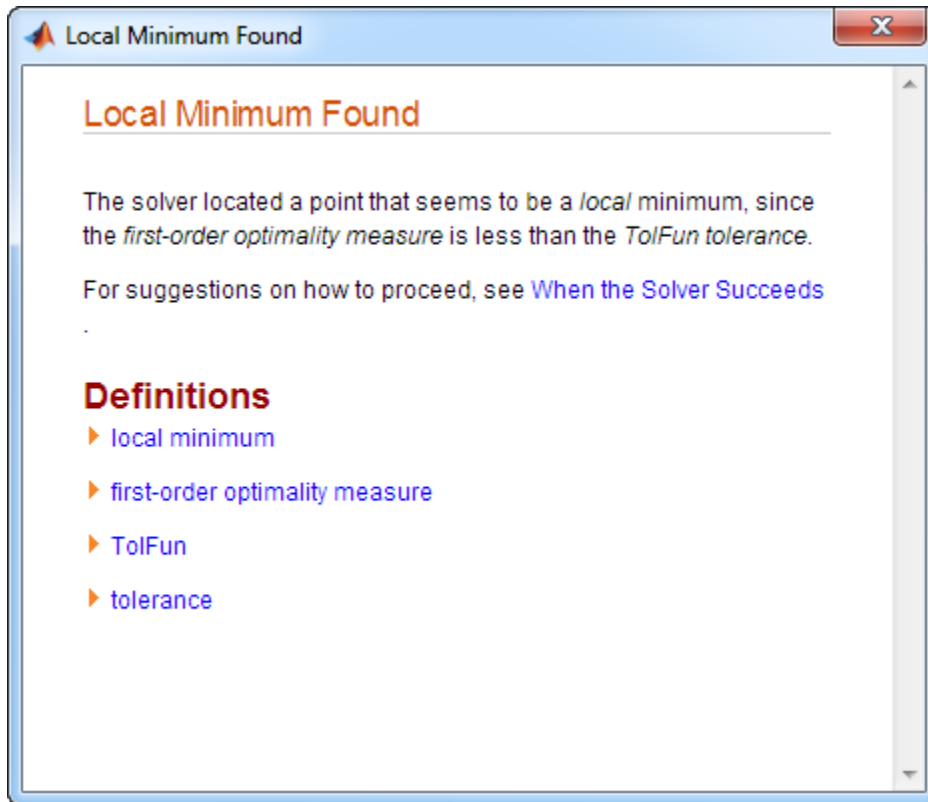
exitflag =

1

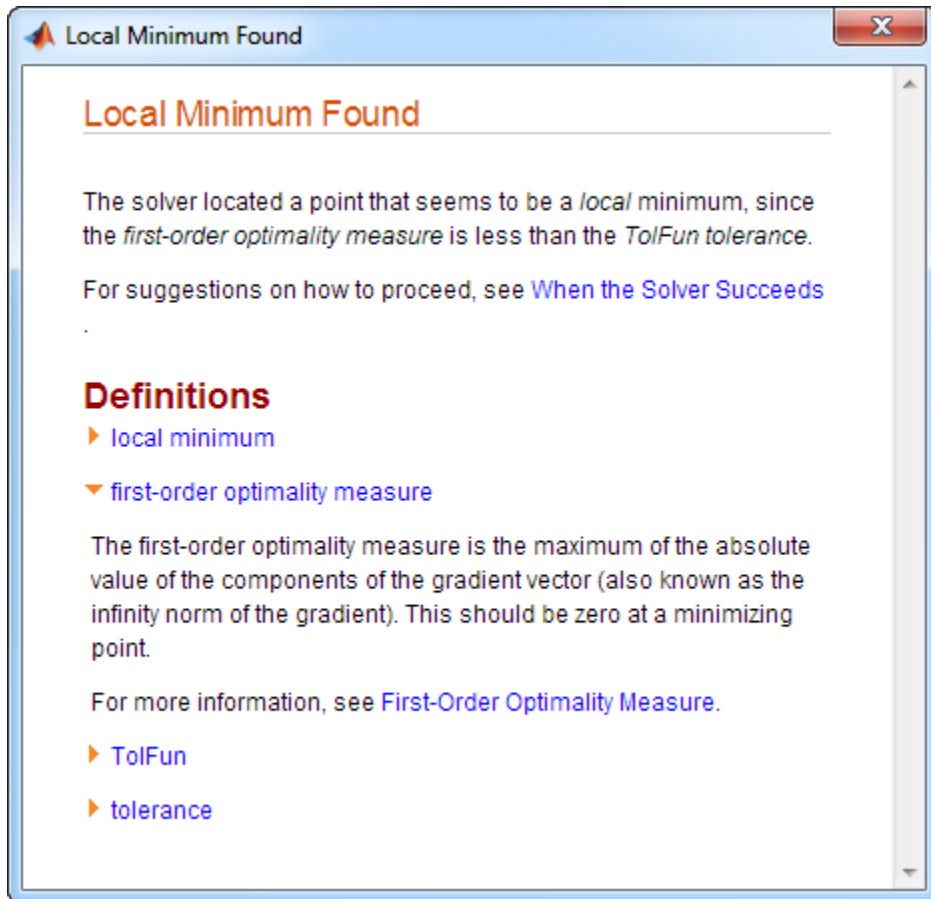
Each of the underlined words or phrases contains a link that provides more information.

- 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 3-17. 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 <code>exitflag ≤ 0</code>	None
'notify-detailed'	Detailed only if <code>exitflag ≤ 0</code>	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);
```

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

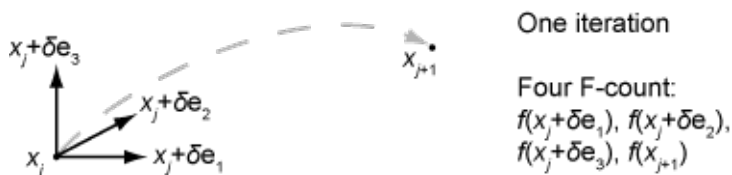
### More About

- “Solver Outputs and Iterative Display”

## Iterations and Function Counts

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

At any step, intermediate calculations may involve evaluating the objective function and constraints, if any, at points near the current iterate  $x_i$ . For example, the solver may estimate a gradient by finite differences. At each of these nearby points, the function count (F-count) is increased by one. The figure “Typical Iteration in 3-D Space” on page 3-10 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  $i$ th coordinate direction.



### Typical Iteration in 3-D Space

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

F-count is a header in the iterative display for many solvers. For an example, see “Interpret the Result” on page 1-20.

The F-count appears in the output structure as `output.funcCount`. This enables you to access the evaluation count programmatically. For more information on output structures, see “Output Structures” on page 3-26.

Sometimes a solver attempts a step, and rejects the attempt. The `trust-region`, `trust-region-reflective`, and `trust-region-dogleg` algorithms count these failed attempts as iterations, and report the (unchanged) result in the iterative display.

The interior-point, active-set, and levenberg-marquardt algorithms do not count such an attempt as an iteration, and do not report the attempt in the iterative display. All attempted steps increase the F-count, regardless of the algorithm.

## **See Also**

### **More About**

- “Solver Outputs and Iterative Display”

## First-Order Optimality Measure

### In this section...

“What Is First-Order Optimality Measure?” on page 3-12

“Stopping Rules Related to First-Order Optimality” on page 3-12

“Unconstrained Optimality” on page 3-13

“Constrained Optimality Theory” on page 3-13

“Constrained Optimality in Solver Form” on page 3-15

### 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 3-13, “Constrained Optimality Theory” on page 3-13, and “Constrained Optimality in Solver Form” on page 3-15.

### 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  $\mu$  times `OptimalityTolerance`, where  $\mu$  is either:

- The infinity norm (maximum) of the gradient of the objective function at  $x_0$
- 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 3-5 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 3-15.

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:

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

The vector  $\lambda$ , which is the concatenation of  $\lambda_g$  and  $\lambda_h$ , is the Lagrange multiplier vector. Its length is the total number of constraints.

The KKT conditions are:

$$\nabla_x L(x, \lambda) = 0, \quad (3-2)$$

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

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

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

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

The optimality measure associated with “Equation 3-3” is

$$\|\overrightarrow{\lambda_g g(x)}\|, \quad (3-6)$$

where the norm in “Equation 3-6” means infinity norm (maximum) of the vector  $\overrightarrow{\lambda_g g(x)}$ .

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 2-84.

## 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”:

$$\|\nabla_x L(x, \lambda)\| = \left\| \nabla f(x) + A^T \lambda_{ineq\,lin} + Aeq^T \lambda_{eq\,lin} + \sum \lambda_{ineq\,nonlin, i} \nabla c_i(x) + \sum \lambda_{eq\,nonlin, i} \nabla ceq_i(x) \right\|, \quad (3-7)$$

$$\left\| \overrightarrow{|l_i - x_i| \lambda_{lower, i}} \overrightarrow{|x_i - u_i| \lambda_{upper, i}} \overrightarrow{|(Ax - b)_i| \lambda_{ineq\,lin, i}} \overrightarrow{|c_i(x)| \lambda_{ineq\,nonlin, i}} \right\|, \quad (3-8)$$

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 3-27. The summations in “Equation 3-7” range over all constraints. If a bound is  $\pm Inf$ , 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  $i$ th 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 3-17  
 “Common Headings” on page 3-18  
 “Function-Specific Headings” on page 3-18

## Introduction

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

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

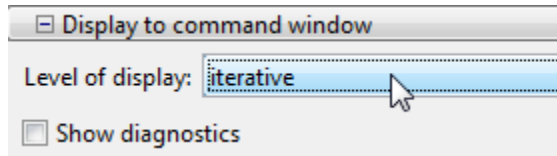
```
options = optimoptions(@fminunc,'Display','iter','Algorithm','quasi-newton');
[x fval exitflag output] = fminunc(@sin,0,options);
```

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

Local minimum found.

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

You can also obtain iterative display by using the Optimization app. Select **Display to command window > Level of display > iterative** or **iterative with detailed message**.



Iterative display is available for all solvers except:

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

## Common Headings

The following table lists some common headings of iterative display.

Heading	Information Displayed
<code>f(x)</code> or <code>Fval</code>	Current objective function value. For <code>fsolve</code> , the square of the norm of the function value vector.
First-order optimality	First-order optimality measure (see “First-Order Optimality Measure” on page 3-12).
Func-count or F-count	Number of function evaluations; see “Iterations and Function Counts” on page 3-10.
Iteration or Iter	Iteration number; see “Iterations and Function Counts” on page 3-10.
Norm of step	Size of the current step (size is the Euclidean norm, or 2-norm). For the 'trust-region' or 'trust-region-reflective' algorithms, when there are constraints <code>Norm of step</code> is the norm of $D*s$ . Here $s$ is the step, and $D$ is a diagonal scaling matrix described in the algorithm description, trust-region subproblem section.

## Function-Specific Headings

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

- “fgoalattain, fmincon, fminimax, and fsemif” on page 3-19
- “fminbnd and fzero” on page 3-20
- “fminsearch” on page 3-21
- “fminunc” on page 3-22
- “fsolve” on page 3-22
- “intlinprog” on page 3-22
- “linprog” on page 3-23
- “lsqin” on page 3-24
- “lsqnonlin and lsqcurvefit” on page 3-24
- “quadprog” on page 3-25

### **fgoalattain, fmincon, fminimax, and fsemif**

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

<b>fgoalattain, fmincon, fminimax, or fsemif Heading</b>	<b>Information Displayed</b>
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 6-24).
Directional derivative	Gradient of the objective function along the search direction.
Feasibility	Maximum constraint violation, where satisfied inequality constraints count as 0.
Line search steplength	Multiplicative factor that scales the search direction (see “Equation 6-45”).
Max constraint	Maximum violation among all constraints, both internally constructed and user-provided; can be negative when no constraint is binding.
Objective value	Objective function value of the nonlinear programming reformulation of the minimax problem for fminimax.

<b>fgoalattain, fmincon, fminimax, or fsemif Heading</b>	<b>Information Displayed</b>
Procedure	<p>Hessian update procedures:</p> <ul style="list-style-type: none"> <li>• Infeasible start point</li> <li>• Hessian not updated</li> <li>• Hessian modified</li> <li>• Hessian modified twice</li> </ul> <p>For more information, see “Updating the Hessian Matrix” on page 6-30.</p> <p>QP subproblem procedures:</p> <ul style="list-style-type: none"> <li>• <b>dependent</b> — There are dependent (redundant) equality constraints that the solver detected and removed.</li> <li>• <b>Infeasible</b> — The QP subproblem with linearized constraints is infeasible.</li> <li>• <b>Overly constrained</b> — The QP subproblem with linearized constraints is infeasible.</li> <li>• <b>Unbounded</b> — The QP subproblem is feasible with large negative curvature.</li> <li>• <b>Ill-posed</b> — The QP subproblem search direction is too small.</li> <li>• <b>Unreliable</b> — The QP subproblem seems to be ill-conditioned.</li> </ul>
Steplength	Multiplicative factor that scales the search direction (see “Equation 6-45”).
Trust-region radius	Current trust-region radius.

**fminbnd and fzero**

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



<b>fminbnd or fzero Heading</b>	<b>Information Displayed</b>
Procedure	Procedures for <code>fminbnd</code> : <ul style="list-style-type: none"> <li>• <code>initial</code></li> <li>• <code>golden</code> (golden section search)</li> <li>• <code>parabolic</code> (parabolic interpolation)</li> </ul> Procedures for <code>fzero</code> : <ul style="list-style-type: none"> <li>• <code>initial</code> (initial point)</li> <li>• <code>search</code> (search for an interval containing a zero)</li> <li>• <code>bisection</code></li> <li>• <code>interpolation</code> (linear interpolation or inverse quadratic interpolation)</li> </ul>
<code>x</code>	Current point for the algorithm

### **fminsearch**

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

<b>fminsearch Heading</b>	<b>Information Displayed</b>
<code>min f(x)</code>	Minimum function value in the current simplex.
Procedure	Simplex procedure at the current iteration. Procedures include: <ul style="list-style-type: none"> <li>• <code>initial simplex</code></li> <li>• <code>expand</code></li> <li>• <code>reflect</code></li> <li>• <code>shrink</code></li> <li>• <code>contract inside</code></li> <li>• <code>contract outside</code></li> </ul> For details, see “ <code>fminsearch</code> Algorithm” on page 6-11.

**fminunc**

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

<b>fminunc Heading</b>	<b>Information Displayed</b>
CG-iterations	Number of conjugate gradient iterations taken in the current iteration (see “Preconditioned Conjugate Gradient Method” on page 6-24)
Line search steplength	Multiplicative factor that scales the search direction (see “Equation 6-11”)

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

**fsolve**

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

<b>fsolve Heading</b>	<b>Information Displayed</b>
Directional derivative	Gradient of the function along the search direction
Lambda	$\lambda_k$ value defined in “Levenberg-Marquardt Method” on page 12-7
Residual	Residual (sum of squares) of the function
Trust-region radius	Current trust-region radius (change in the norm of the trust-region radius)

**intlinprog**

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

<b>intlinprog Heading</b>	<b>Information Displayed</b>
nodes explored	Cumulative number of explored nodes.
total time (s)	Time in seconds since <code>intlinprog</code> started.
num int solution	Number of integer feasible points found.

<b>intlinprog Heading</b>	<b>Information Displayed</b>
integer fval	Objective function value of the best integer feasible point found. This is an upper bound for the final objective function value.
relative gap (%)	$\frac{100(b - a)}{ b  + 1},$ <p>where</p> <ul style="list-style-type: none"> <li>• <math>b</math> is the objective function value of the best integer feasible point.</li> <li>• <math>a</math> is the best lower bound on the objective function value.</li> </ul> <p><b>Note</b> While you specify <code>RelativeGapTolerance</code> as a decimal number, the iterative display and output <code>.relativegap</code> report the gap in 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.</p>

## linprog

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

<b>linprog Heading</b>	<b>Information Displayed</b>
Primal Infeas A*x-b or Primal Infeas	<p>Primal infeasibility, a measure of the constraint violations, which should be zero at a solution.</p> <p>For definitions, see “Predictor-Corrector” on page 9-4 ('interior-point') or “Main Algorithm” on page 9-8 ('interior-point-legacy') or “Dual-Simplex Algorithm” on page 9-11.</p>

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 9-4. For the definition of dual infeasibility, see "Predictor-Corrector" on page 9-4 ('interior-point') or "Main Algorithm" on page 9-8 ('interior-point-legacy') or "Dual-Simplex Algorithm" on page 9-11.
Upper Bounds {x} +s-ub	Upper bound feasibility. {x} means those x with finite upper bounds. This is the $r_u$ residual in "Interior-Point-Legacy Linear Programming" on page 9-8.
Duality Gap x'*z +s'*w	Duality gap (see "Interior-Point-Legacy Linear Programming" on page 9-8) between the primal objective and the dual objective. s and w appear in this equation only if there are finite upper bounds.
Total Rel Error	Total relative error, described at the end of "Main Algorithm" on page 9-8.
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 9-7.
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 "Interior-Point Linear Least Squares" on page 12-6. For iterative display details, see "quadprog" on page 3-25.

### lsqnonlin and lsqcurvefit

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

<b>lsqnonlin or lsqcurvefit Heading</b>	<b>Information Displayed</b>
Directional derivative	Gradient of the function along the search direction
Lambda	$\lambda_k$ value defined in "Levenberg-Marquardt Method" on page 12-7
Resnorm	Value of the squared 2-norm of the residual at x
Residual	Residual vector of the function

### **quadprog**

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

<b>quadprog Heading</b>	<b>Information Displayed</b>
Primal Infeas	Primal infeasibility, defined as $\max(\text{norm}(\text{Aeq} \cdot x - \text{beq}, \text{inf}), \text{abs}(\min(0, \min(\text{A} \cdot x - \text{b}))))$
Dual Infeas	Dual infeasibility, defined as $\text{norm}(\text{H} \cdot x + \text{f} - \text{A} \cdot \text{lambda\_ineqlin} - \text{Aeq} \cdot \text{lambda\_eqlin}, \text{inf})$
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 11-9.

## Output Structures

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

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

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

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

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

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

## 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 3-12. Solvers return estimated Lagrange multipliers in a structure. The structure is called `lambda`, since the conventional symbol for Lagrange multipliers is the Greek letter lambda ( $\lambda$ ). The structure separates the multipliers into the following types, called fields:

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

To access, for example, the nonlinear inequality field of a Lagrange multiplier structure, enter `lambda.ineqnonlin`. 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  $x(1) + x(2) \leq 1$ . 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  $x(1)^2 + x(2)^2 \leq 1$ .



```
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

In this section...
“fminunc Hessian” on page 3-30
“fmincon Hessian” on page 3-31

### fminunc Hessian

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

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

- **Quasi-Newton Algorithm** — `fminunc` returns an estimated Hessian matrix at the solution. It computes the estimate by finite differences.
- **Trust-Region Algorithm** — `fminunc` returns a Hessian matrix at the next-to-last iterate.
  - If you supply a Hessian in the objective function, `fminunc` returns this Hessian.
  - If you supply a `HessMult` function, `fminunc` returns the `Hinfo` matrix from the `HessMult` function. For more information, see `HessMult` in the `trust-region` section of the `fminunc` options table.
  - Otherwise, `fminunc` returns an approximation from a sparse finite difference algorithm on the gradients.

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

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

## fmincon Hessian

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

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

The  $\lambda_i$  are Lagrange multipliers; see “First-Order Optimality Measure” on page 3-12 and “Lagrange Multiplier Structures” on page 3-27. The Hessian of the Lagrangian is

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

fmincon has four algorithms, with several options for Hessians, as described in “fmincon Trust Region Reflective Algorithm” on page 6-22, “fmincon Active Set Algorithm” on page 6-27, and “fmincon Interior Point Algorithm” on page 6-37. fmincon returns the following for the Hessian:

- **active-set or sqp Algorithm** — fmincon returns the Hessian approximation it computes at the next-to-last iterate. fmincon computes a quasi-Newton approximation of the Hessian matrix at the solution in the course of its iterations. This approximation does not, in general, match the true Hessian in every component, but only in certain subspaces. Therefore the Hessian that fmincon returns can be inaccurate. For more details of the active-set calculation, see “SQP Implementation” on page 6-30.
- **trust-region-reflective Algorithm** — fmincon returns the Hessian it computes at the next-to-last iterate.
  - If you supply a Hessian in the objective function, fmincon returns this Hessian.
  - If you supply a HessMult function, fmincon returns the Hinfo matrix from the HessMult function. For more information, see **Trust-Region-Reflective Algorithm** in fmincon options.
  - Otherwise, fmincon returns an approximation from a sparse finite difference algorithm on the gradients.

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

The reason the trust-region-reflective algorithm returns the Hessian at the next-to-last point is for efficiency. fmincon uses the Hessian internally to compute its

next step. When `fmincon` reaches a stopping condition, it does not need to compute the next step, so does not compute the Hessian.

- **interior-point Algorithm**

- If the Hessian option is `lbfgs` or `fin-diff-grads`, or if you supply a Hessian multiply function (`HessMult`), `fmincon` returns `[]` for the Hessian.
- If the Hessian option is `bfgs` (the default), `fmincon` returns a quasi-Newton approximation to the Hessian at the final point. This Hessian can be inaccurate, as in the `active-set` or `sqp` algorithm Hessian.
- If the Hessian option is `user-supplied`, `fmincon` returns the user-supplied Hessian at the final point.

## See Also

### More About

- “Including Gradients and Hessians” on page 2-25
- “Hessian as an Input” on page 16-103

## Plot Functions

### In this section...

“Plot an Optimization During Execution” on page 3-33

“Using a Plot Function” on page 3-33

### 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
- **Optimization app > Options > Plot functions**

You can also use a custom-written plot function. Write a function file using the same structure as an output function. For more information on this structure, see “Output Function Syntax” on page 15-37.

### Using a Plot Function

This example shows how to use a plot function to view the progress of the `fmincon` interior-point algorithm. The problem is taken from the Getting Started “Solve a Constrained Nonlinear Problem, Solver-Based” on page 1-13. The first part of the example shows how to run the optimization using the Optimization app. The second part shows how to run the optimization from the command line.

---

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

---

### Running the Optimization Using the Optimization App

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

```
function [f g H] = rosenboth(x)
% ROSENBOTh returns both the value y of Rosenbrock's function
```

% and also the value g of its gradient and H the Hessian.

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

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

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

Save this file as `rosenboth.m`.

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

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

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

Save this file as `unitdisk2.m`.

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

**3** Set up the optimization:

- Choose the `fmincon` solver.
- Choose the Interior point algorithm.
- Set the objective function to `@rosenboth`.
- Choose Gradient supplied for the objective function derivative.
- Set the start point to `[0 0]`.
- Set the nonlinear constraint function to `@unitdisk2`.
- Choose Gradient supplied for the nonlinear constraint derivatives.

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

**Problem Setup and Results**

Solver:

Algorithm:

Problem

Objective function:

Derivatives:

Start point:

Constraints:

Linear inequalities:    A:     b:

Linear equalities:    Aeq:     beq:

Bounds:    Lower:     Upper:

Nonlinear constraint function:

Derivatives:

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

Plot functions

Current point     Function count     Function value

Max constraint     Current step     First order optimality

Custom function:

- 5 Click the **Start** button under **Run solver and view results**.
- 6 The output appears as follows in the Optimization app.

Run solver and view results

Start    Pause    Stop

Current iteration: 24    Clear Results

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

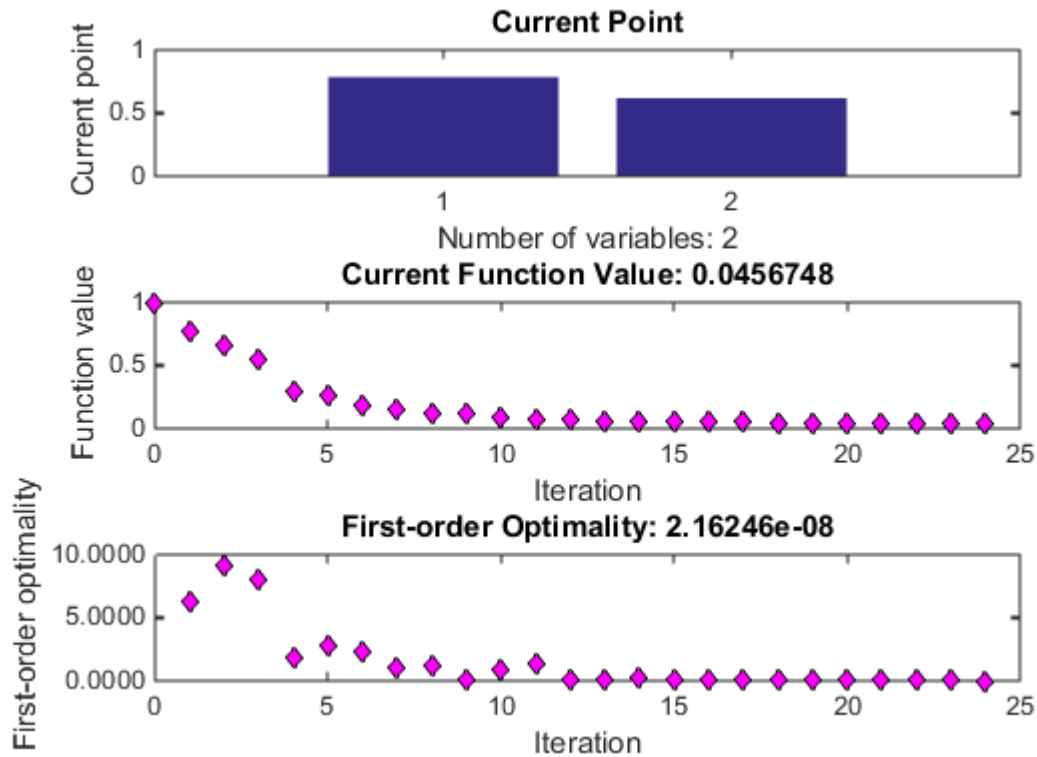
Optimization completed because the objective function is non-decreasing in feasible directions, to within the default value of the function tolerance, and constraints are satisfied to within the default value of the constraint tolerance.

Final point:

1 ▲	2
0.786	0.618

In addition, the following three plots appear in a separate window.





- The “Current Point” plot graphically shows the minimizer  $[0.786, 0.618]$ , which is reported as the **Final point** in the **Run solver and view results** pane. This plot updates at each iteration, showing the intermediate iterates.
- The “Current Function Value” plot shows the objective function value at all iterations. This graph is nearly monotone, showing `fmincon` reduces the objective function at almost every iteration.
- The “First-order Optimality” plot shows the first-order optimality measure at all iterations.

### Running the Optimization from the Command Line

- 1 Write the nonlinear objective and constraint functions, including the derivatives, as shown in “Running the Optimization Using the Optimization App” on page 3-33.

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

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

- 3 Call `fmincon`:

```
x = fmincon(@rosenboth,[0 0],[[],[],[],[],[],[],[]],...  
    @unitdisk2,options)
```

- 4 `fmincon` gives the following output:

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

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

```
x =  
    0.7864    0.6177
```

`fmincon` also displays the three plot functions, shown at the end of “Running the Optimization Using the Optimization App” on page 3-33.

## See Also

### More About

- “Output Function Syntax” on page 15-37
- “Output Function for Problem-Based Optimization” on page 7-31

# Output Functions

**In this section...**

“What Is an Output Function?” on page 3-39

“Example: Using Output Functions” on page 3-39

## 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 Syntax” on page 15-37 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 7-31.

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

## Example: Using Output Functions

- “What the Example Contains” on page 3-39
- “Writing the Output Function” on page 3-40
- “Writing the Example Function File” on page 3-41
- “Running the Example” on page 3-43

### What the Example Contains

The following example continues the one in “Nonlinear Inequality Constraints” on page 6-79, which calls the function `fmincon` at the command line to solve a nonlinear, constrained optimization problem. The example in this section uses a function file to call `fmincon`. The file also contains all the functions needed for the example, including:

- The objective function

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

The code for the file is here: “Writing the Example Function File” on page 3-41.

#### Writing the Output Function

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

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

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

In general, `outfun` can be any MATLAB function, but in this example, it is a nested function of the function file described in “Writing the Example Function File” on page 3-41. The following code defines the output function:

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

switch state
    case 'init'
        hold on
    case 'iter'
        % Concatenate current point and objective function
        % value with history. x must be a row vector.
        history.fval = [history.fval; optimValues.fval];
        history.x = [history.x; x];
        % Concatenate current search direction with
        % searchdir.
        searchdir = [searchdir;...
                    optimValues.searchdirection'];
        plot(x(1),x(2),'o');
        % Label points with iteration number.
```

```

        % Add .15 to x(1) to separate label from plotted 'o'
        text(x(1)+.15,x(2),num2str(optimValues.iteration));
    case 'done'
        hold off
    otherwise
end
end
end

```

See “Using Handles to Store Function Parameters” (MATLAB) for more information about nested functions.

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

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

The example uses the following fields of `optimValues`:

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

For more information about these arguments, see “Output Function Syntax” on page 15-37.

### Writing the Example Function File

To create the function file for this example:

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

```

function [history,searchdir] = runfmincon

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

% call optimization

```

```
x0 = [-1 1];
options = optimoptions(@fmincon,'OutputFcn',@outfun,...
    'Display','iter','Algorithm','active-set');
xsol = fmincon(@objfun,x0,[],[],[],[],[],[],@confun,options);

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

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

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

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

```
end
end
```

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

### Running the Example

To run the example, enter:

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

This displays the following iterative output in the Command Window.

Iter	F-count	f(x)	Max constraint	Line search steplength	Directional derivative	First-order optimality	Procedure
0	3	1.8394	0.5				Infeasible start point
1	6	1.85127	-0.09197	1	0.109	0.778	
2	9	0.300167	9.33	1	-0.117	0.313	Hessian modified twice
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	0.035323	-0.03303	1	-0.0542	0.0271	
7	25	0.0235566	0.003184	1	-0.0271	0.00587	
8	28	0.0235504	9.031e-08	1	-0.0146	8.51e-07	

Active inequalities (to within options.ConstraintTolerance = 1e-06):

lower	upper	ineqlin	ineqnonlin
		1	
		2	

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

```
history =
```

```
struct with fields:
```

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

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

```
history.fval
```

```
ans =
```

```
1.8394
```

```
1.8513
0.3002
0.5298
0.1870
0.0729
0.0353
0.0236
0.0236
```

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

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

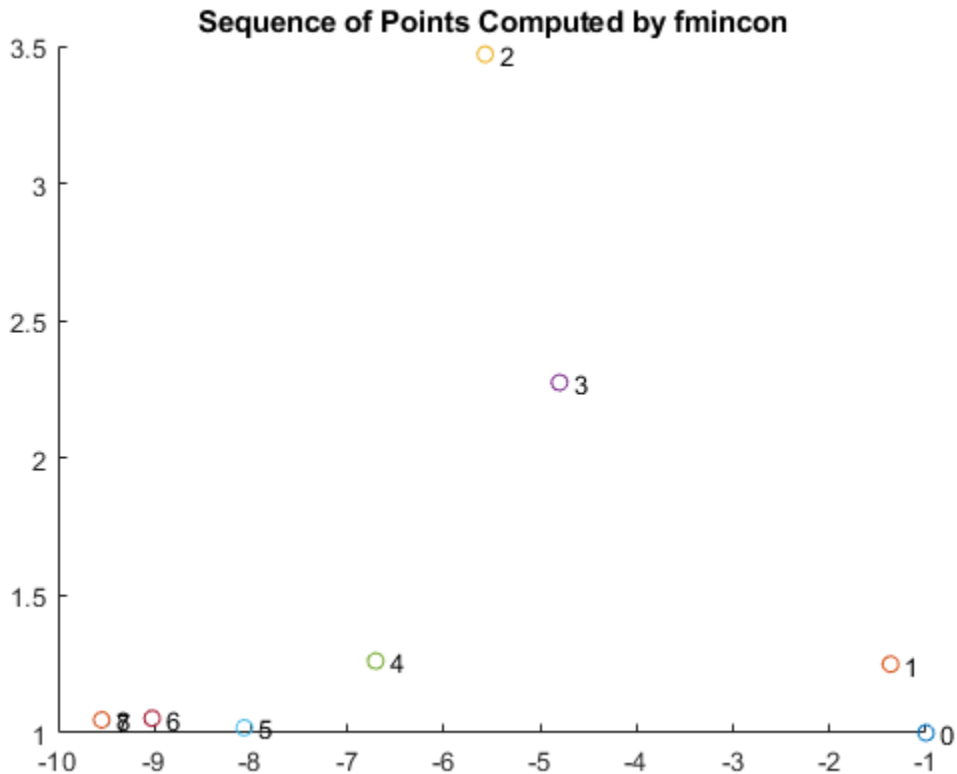
```
history.x
```

```
ans =
```

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

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





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

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

`searchdir =`

```

-0.3679    0.2500
-4.2029    2.2199
 0.7708   -1.1947
-3.8108   -2.0268
-1.3625   -0.2432

```

```
-0.9552    0.0346  
-0.5241   -0.0061  
-0.0003    0.0003
```

## See Also

### More About

- “Output Function Syntax” on page 15-37
- “Output Function for Problem-Based Optimization” on page 7-31

# Steps to Take After Running a Solver

---

- “Overview of Next Steps” on page 4-2
- “When the Solver Fails” on page 4-3
- “Solver Takes Too Long” on page 4-11
- “When the Solver Might Have Succeeded” on page 4-15
- “When the Solver Succeeds” on page 4-22
- “Local vs. Global Optima” on page 4-27
- “Optimizing a Simulation or Ordinary Differential Equation” on page 4-32

# 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 4-27)
- 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 4-11
- “When the Solver Might Have Succeeded” on page 4-15
- “When the Solver Succeeds” on page 4-22

## When the Solver Fails

### In this section...

“Too Many Iterations or Function Evaluations” on page 4-3

“Converged to an Infeasible Point” on page 4-7

“Problem Unbounded” on page 4-9

“fsolve Could Not Solve Equation” on page 4-10

### 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 4-5
- “5. Center and Scale Your Problem” on page 4-5
- “6. Provide Gradient or Jacobian” on page 4-6
- “7. Provide Hessian” on page 4-7

#### 1. Enable Iterative Display

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

To enable iterative display:

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

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

Call the solver using the `options` structure.

For an example of iterative display, see “Interpret the Result” on page 1-20.

### 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 Optimization app, or in the Options table in the solver's function reference pages.
  - 2 Start the solver from its last calculated point.
- If the solver is not progressing, try the other listed suggestions.

## 2. Relax Tolerances

If `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 using the Optimization app, use the **Stopping criteria** list at the top of the **Options** pane.

To change tolerances at the command line, use `optoptions` as described in “Set and Change Options” on page 2-76.

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 4-23.

#### 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 4-25. Check that an infeasible point does not cause an error in your functions; see “Iterations Can Violate Constraints” on page 2-42.

#### 5. Center and Scale Your Problem

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

##### Example: Centering and Scaling

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

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

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

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

```
x =
     0
 0.5000
```

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

Scale the problem. Set

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

```
y =
     0
     0 % the correct answer
```

Similarly, poor centering can interfere with a solution.

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

```
z =  
    1.0e+005 *  
    10.0000  -10.0000 % looks good, but...  
  
z - [1e6 -1e6] % checking how close z is to 1e6  
  
ans =  
  
    -0.0071    0.0078 % reveals a distance  
  
fcc = @(w)fc([w(1)+1e6;w(2)-1e6]); % centered  
  
w = fminunc(fcc,[.5 .5],opts)  
  
w =  
     0     0 % the correct answer
```

### 6. Provide Gradient or Jacobian

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

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

Provide gradients or Jacobians in the files for your objective function and nonlinear constraint functions. For details of the syntax, see “Writing Scalar Objective Functions” on page 2-23, “Writing Vector and Matrix Objective Functions” on page 2-34, and “Nonlinear Constraints” on page 2-48.

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 2-87.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

For examples using gradients and Jacobians, see “Minimization with Gradient and Hessian” on page 6-16, “Nonlinear Constraints with Gradients” on page 6-81,



“Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115, “Nonlinear Equations with Analytic Jacobian” on page 13-9, and “Nonlinear Equations with Jacobian” on page 13-14.

## 7. Provide Hessian

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

The following solvers and algorithms accept Hessians:

- `fmincon interior-point`. Write the Hessian as a separate function. For an example, see “fmincon Interior-Point Algorithm with Analytic Hessian” on page 6-84.
- `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 6-110.
- `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 6-16.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

## 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 4-9. If `quadprog` returns this result, see “quadprog Converges to an Infeasible Point” on page 4-9

To proceed when the solver found no feasible point, try one or more of the following.

- “1. Check Linear Constraints” on page 4-7
- “2. Check Nonlinear Constraints” on page 4-8

### 1. Check Linear Constraints

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

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

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

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

### 2. Check Nonlinear Constraints

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

- Set your objective function to zero:

```
@(x)0
```

Run your optimization with all constraints and with the zero objective. If you find a feasible point  $x_{new}$ , set  $x_0 = x_{new}$  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  $x_{new}$ , set  $x_0 = x_{new}$  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  $c - \Delta$ , where  $\Delta$  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  $\Delta$
    - 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  $\Delta$  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

$$\lim_{i \rightarrow \infty} f(x_i) = -\infty,$$

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 2-39.
- Try scaling or centering your problem. See Center and Scale Your Problem on page 4-5.
- 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 4-23. `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  $x^2 + 1 = 0$ .

## 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 4-11
2. “Use Appropriate Tolerances” on page 4-11
3. “Use a Plot Function” on page 4-12
4. “Use 'lbfgs' HessianApproximation Option” on page 4-12
5. “Enable CheckGradients” on page 4-12
6. “Use Inf Instead of a Large, Arbitrary Bound” on page 4-13
7. “Use an Output Function” on page 4-13
8. “Use a Sparse Solver or a Multiply Function” on page 4-13
9. “Use Parallel Computing” on page 4-14

### Enable Iterative Display

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

To enable iterative display:

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

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

Call the solver using the `options` structure.

For an example of iterative display, see “Interpret the Result” on page 1-20. For more information, see “What to Look For in Iterative Display” on page 4-4.

### Use Appropriate Tolerances

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

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

To change tolerances at the command line, use `optimoptions` as described in “Set and Change Options” on page 2-76.

### Use a Plot Function

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

To use a plot function:

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

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

Call the solver using the `options` structure.

For an example of using a plot function, see “Using a Plot Function” on page 3-33.

### 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 2-87.

## 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 Syntax” on page 15-37.

For an example of using an output function, see “Example: Using Output Functions” on page 3-39.

## 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 2-14). 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 12-37, “Quadratic

Minimization with Dense, Structured Hessian” on page 11-18, and “Minimization with Dense Structured Hessian, Linear Equalities” on page 6-110.

### **Use Parallel Computing**

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



## When the Solver Might Have Succeeded

### In this section...

“Final Point Equals Initial Point” on page 4-15

“Local Minimum Possible” on page 4-15

### Final Point Equals Initial Point

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

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

- 1 Starting from different points — see [Change the Initial Point](#) on page 4-23.
- 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 4-25. Check that an infeasible point does not cause an error in your functions; see [“Iterations Can Violate Constraints”](#) on page 2-42.
- 3 Changing tolerances, such as `OptimalityTolerance`, `ConstraintTolerance`, and `StepTolerance` — see [Use Appropriate Tolerances](#) on page 4-11.
- 4 Scaling your problem so each coordinate has about the same effect — see [Rescale the Problem](#) on page 4-20.
- 5 Providing gradient and Hessian information — see [Provide Analytic Gradients or Jacobian](#) on page 4-20 and [Provide a Hessian](#) on page 4-21.

### 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 3-12.) To see if the reported solution is reliable, consider the following suggestions.

- “1. Nonsmooth Functions” on page 4-16
- “2. Rerun Starting At Final Point” on page 4-16
- “3. Try a Different Algorithm” on page 4-17
- “4. Change Tolerances” on page 4-19
- “5. Rescale the Problem” on page 4-20
- “6. Check Nearby Points” on page 4-20

- “7. Change Finite Differencing Options” on page 4-20
- “8. Provide Analytic Gradients or Jacobian” on page 4-20
- “9. Provide a Hessian” on page 4-21

### 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 4-24.

### 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 6-129:

```
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 4-16 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'  
[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 4-17 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 4-5.

### 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 4-24 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'`.

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

### 8. Provide Analytic Gradients or Jacobian

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

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

Provide gradients or Jacobians in the files for your objective function and nonlinear constraint functions. For details of the syntax, see “Writing Scalar Objective Functions” on page 2-23, “Writing Vector and Matrix Objective Functions” on page 2-34, and “Nonlinear Constraints” on page 2-48.

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 2-87.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

For examples using gradients and Jacobians, see “Minimization with Gradient and Hessian” on page 6-16, “Nonlinear Constraints with Gradients” on page 6-81, “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115, “Nonlinear Equations with Analytic Jacobian” on page 13-9, and “Nonlinear Equations with Jacobian” on page 13-14.

## 9. Provide a Hessian

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

The following solvers and algorithms accept Hessians:

- `fmincon interior-point`. Write the Hessian as a separate function. For an example, see “fmincon Interior-Point Algorithm with Analytic Hessian” on page 6-84.
- `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 6-110.
- `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 6-16.

If you have a Symbolic Math Toolbox license, you can calculate gradients and Hessians programmatically. For an example, see “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

The example in “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115 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 4-22

“1. Change the Initial Point” on page 4-23

“2. Check Nearby Points” on page 4-24

“3. Check your Objective and Constraint Functions” on page 4-25

## 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 4-27.

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 4-27.

## 2. Check Nearby Points

To see if there are better values than a reported solution, evaluate your objective function and constraints at various nearby points.

For example, with the objective function `ffun` from “What Can Be Wrong If The Solver Succeeds?” on page 4-22, and the final point `xfinal = -1.5056e-008`, calculate `ffun(xfinal±Δ)` for some  $\Delta$ :

```
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 value of 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 2-42.

## Local vs. Global Optima

### In this section...

“Why Didn't the Solver Find the Smallest Minimum?” on page 4-27

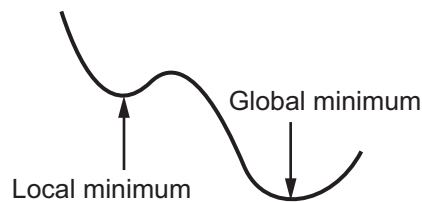
“Searching for a Smaller Minimum” on page 4-28

“Basins of Attraction” on page 4-28

### Why Didn't the Solver Find the Smallest Minimum?

In general, solvers return a local minimum. The result might be a global minimum, but there is no guarantee that it is. This section describes why solvers behave this way, and gives suggestions for ways to search for a global minimum, if needed.

- A local minimum of a function is a point where the function value is smaller than at nearby points, but possibly greater than at a distant point.
- A global minimum is a point where the function value is smaller than at all other feasible points.



Generally, Optimization Toolbox solvers find a local optimum. (This local optimum can be a global optimum.) They find the optimum in the basin of attraction of the starting point. For more information about basins of attraction, see “Basins of Attraction” on page 4-28.

There are some exceptions to this general rule.

- Linear programming and positive definite quadratic programming problems are convex, with convex feasible regions, so there is only one basin of attraction. Indeed, under certain choices of options, `linprog` ignores any user-supplied starting point, and `quadprog` does not require one, though supplying one can sometimes speed a minimization.
- Global Optimization Toolbox functions, such as `simulannealbnd`, attempt to search more than one basin of attraction.

### Searching for a Smaller Minimum

If you need a global optimum, you must find an initial value for your solver in the basin of attraction of a global optimum.

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

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

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

### Basins of Attraction

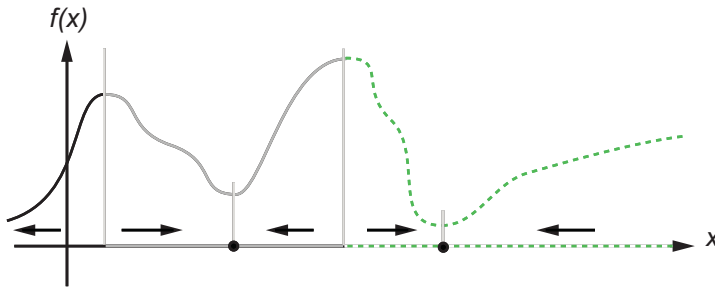
If an objective function  $f(x)$  is smooth, the vector  $-\nabla f(x)$  points in the direction where  $f(x)$  decreases most quickly. The equation of steepest descent, namely

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

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

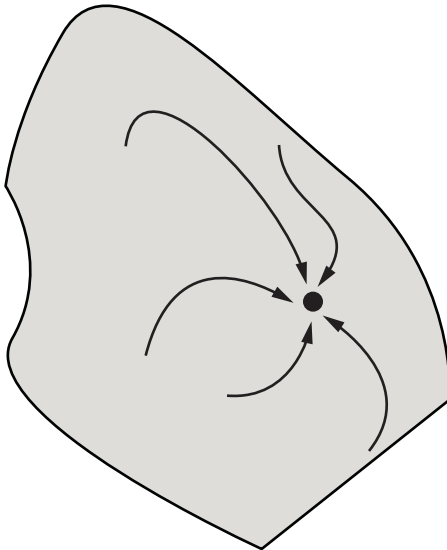
The following figure shows two one-dimensional minima. The figure shows different basins of attraction with different line styles, and shows directions of steepest descent with arrows. For this and subsequent figures, black dots represent local minima. Every

steepest descent path, starting at a point  $x(0)$ , goes to the black dot in the basin containing  $x(0)$ .



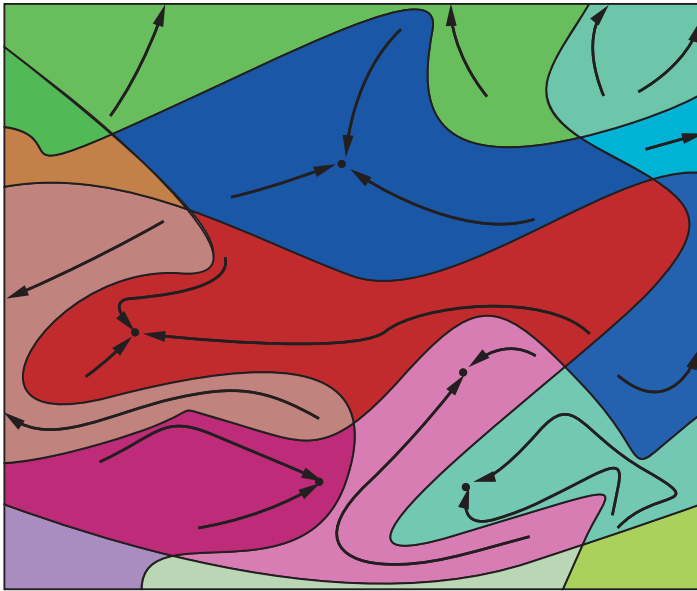
### One-dimensional basins

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



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

The following figure shows even more complicated paths and basins of attraction.



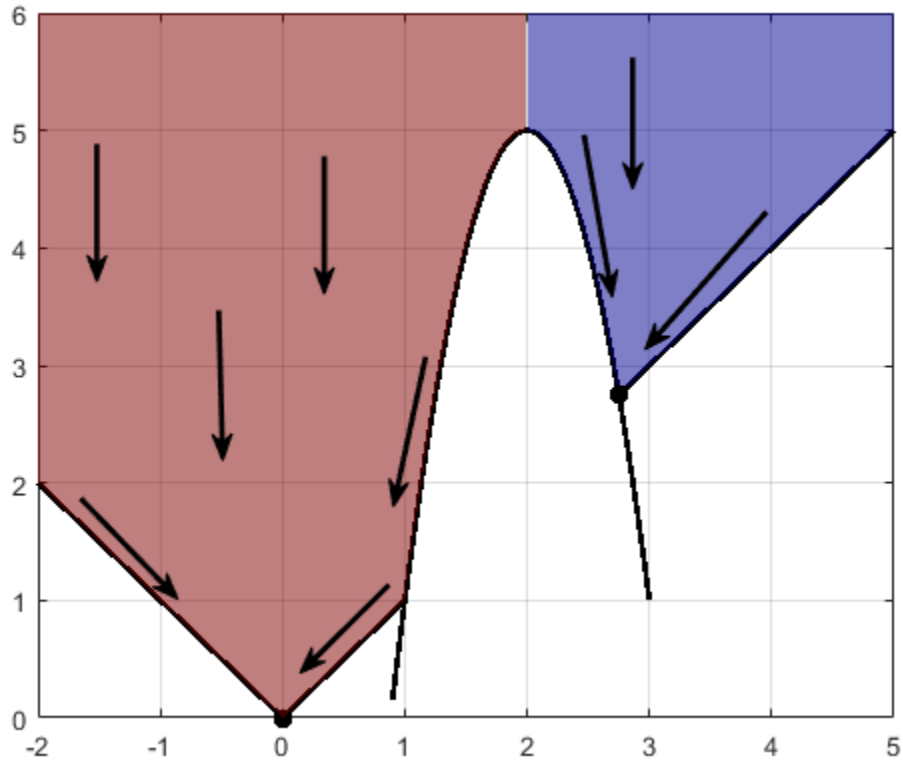
### Several basins of attraction

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

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

The figure shows the two basins of attraction with the final points.





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

## Optimizing a Simulation or Ordinary Differential Equation

<b>In this section...</b>
“What Is Optimizing a Simulation or ODE?” on page 4-32
“Potential Problems and Solutions” on page 4-32
“Bibliography” on page 4-37

### 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 4-32.

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

### Potential Problems and Solutions

- “Problems in Finite Differences” on page 4-32
- “Suggestions for Finite Differences” on page 4-33
- “Problems in Stochastic Functions” on page 4-36
- “Suggestions for Stochastic Functions” on page 4-36
- “Common Calculation of Objective and Constraints” on page 4-36
- “Failure in Objective or Constraint Function Evaluation” on page 4-36
- “Suggestions for Evaluation Failures” on page 4-36

### 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  $\delta$  allows more nonlinearity to affect the finite difference.
- A small value of  $\delta$  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  $\delta$ , 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 4-33
- “Set Larger Finite Differences” on page 4-34
- “Use a Gradient Evaluation Function” on page 4-34
- “Use Tighter ODE Tolerances” on page 4-35

### **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 4-32.

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 4-32 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

Specify `ObjectiveGradient` 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” (MATLAB).

### 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 2-60, or, when using the problem-based approach, “Objective and Constraints Having a Common Function in Serial or Parallel, Problem-Based” on page 2-65.

### 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 2-42, 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 2-42.

### 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” on page 5-2
- “Optimization App Alternatives” on page 5-15

## Optimization App

---

**Note** The Optimization app warns that it will be removed in a future release. For alternatives, see “Optimization App Alternatives” on page 5-15.

---

In this section...
“Optimization App Basics” on page 5-2
“Specifying Certain Options” on page 5-8
“Importing and Exporting Your Work” on page 5-11

### Optimization App Basics

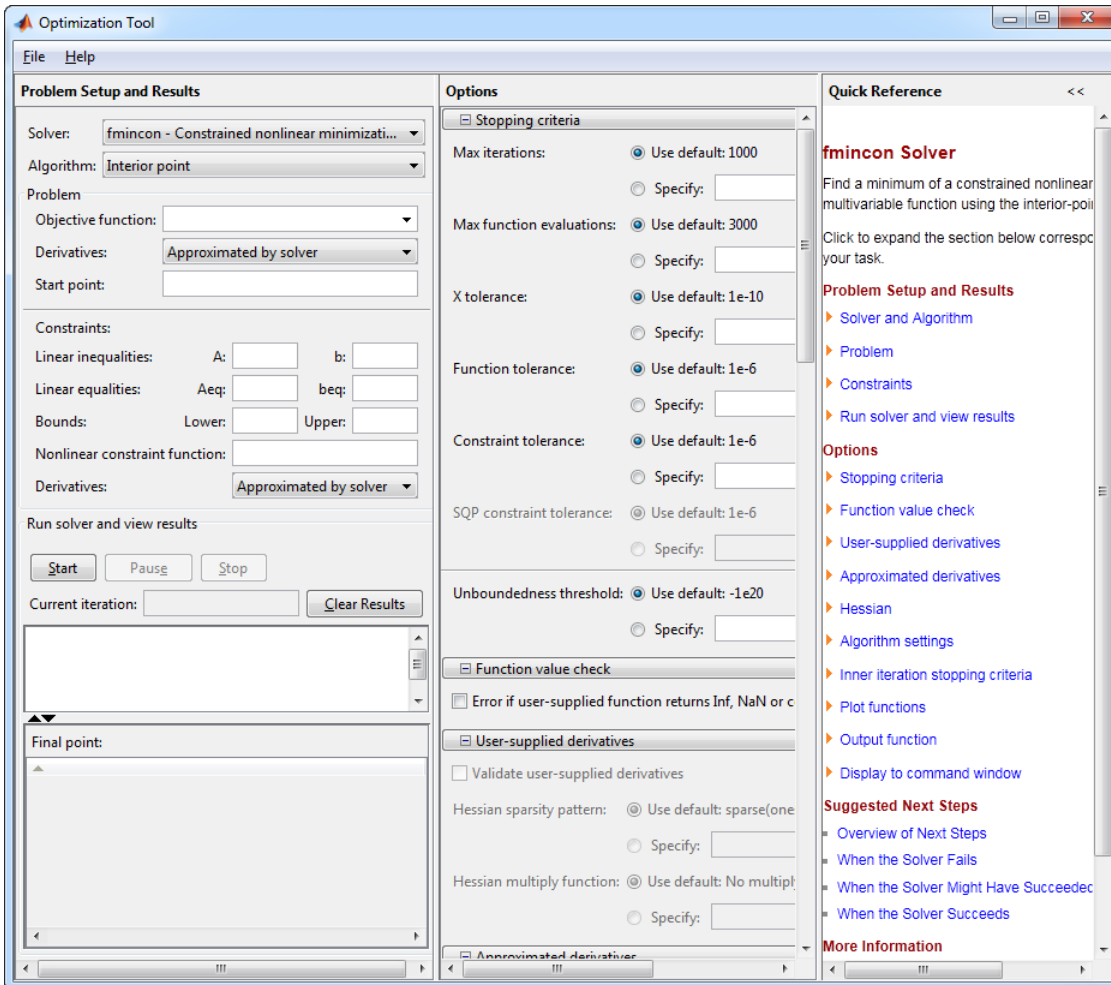
- “How to Open the Optimization App” on page 5-2
- “Examples that Use the Optimization App” on page 5-4
- “Steps for Using the Optimization App” on page 5-4
- “Pausing and Stopping” on page 5-5
- “Viewing Results” on page 5-6
- “Final Point” on page 5-7
- “Starting a New Problem” on page 5-7

### How to Open the Optimization App

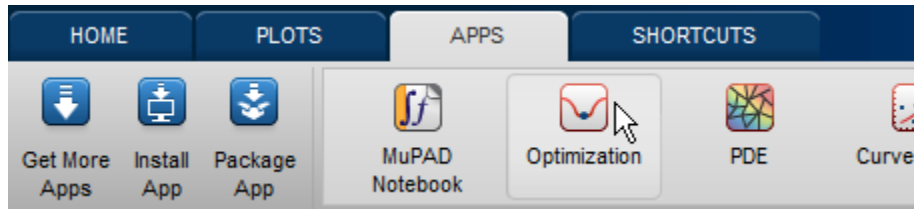
To open the Optimization app, type

```
optimtool
```

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



You can also start the Optimization app from the MATLAB **Apps** tab.



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

### Examples that Use the Optimization App

The following documentation examples use the optimization app:

- “Solve a Constrained Nonlinear Problem, Solver-Based” on page 1-13
- “Optimization App with the `fmincon` Solver” on page 6-97
- “Optimization App with the `lsqin` Solver” on page 12-33
- “Plot Functions” on page 3-33
- “`fmincon` Interior-Point Algorithm with Analytic Hessian” on page 6-84

### Steps for Using the Optimization App

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

7. Import and export problem, options, and results

4. Specify options

1. Select solver and algorithm

2. Specify function to minimize

3. Set problem parameters for selected solver

5. Run solver

6. View solver status and results

## Pausing and Stopping

While a solver is running, you can

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

You can export your results after stopping the algorithm. For details, see “Exporting Your Work” on page 5-11.

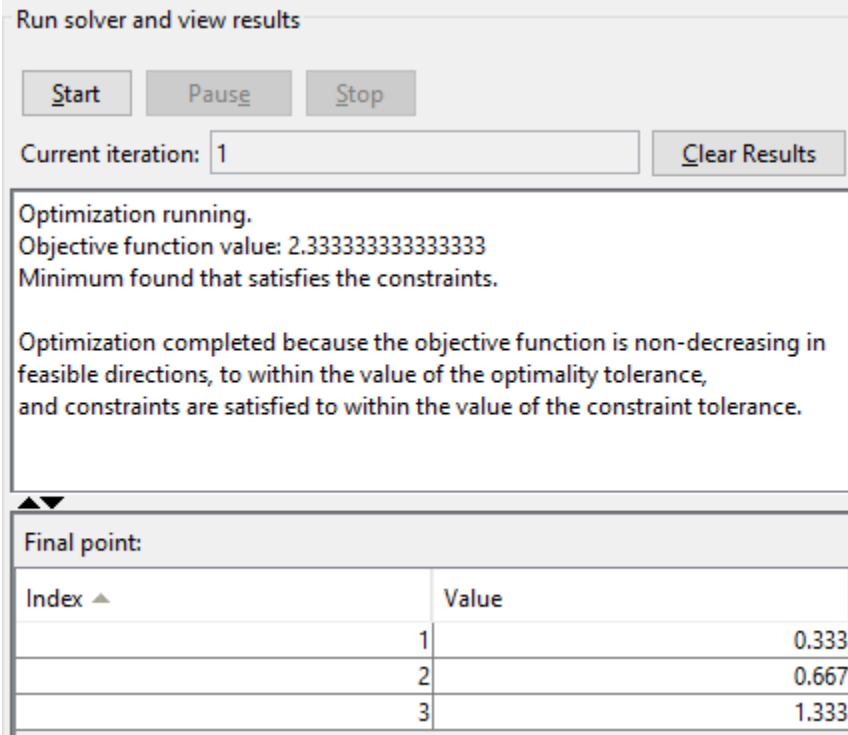
### Viewing Results

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

### Sorting the Displayed Results

Depending on the solver and problem, results can be in the form of a table. If the table has multiple rows, sort the table by clicking a column heading. Click the heading again to sort the results in reverse.

For example, suppose you use the Optimization app to solve the `lsqlin` problem described in “Optimization App with the `lsqlin` Solver” on page 12-33. The result appears as follows.



The screenshot shows a window titled "Run solver and view results". At the top, there are three buttons: "Start", "Pause", and "Stop". Below these is a text field for "Current iteration:" containing the number "1", and a "Clear Results" button. The main area contains text indicating that optimization is running and has completed. Below this is a table titled "Final point:" with two columns: "Index" and "Value".

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

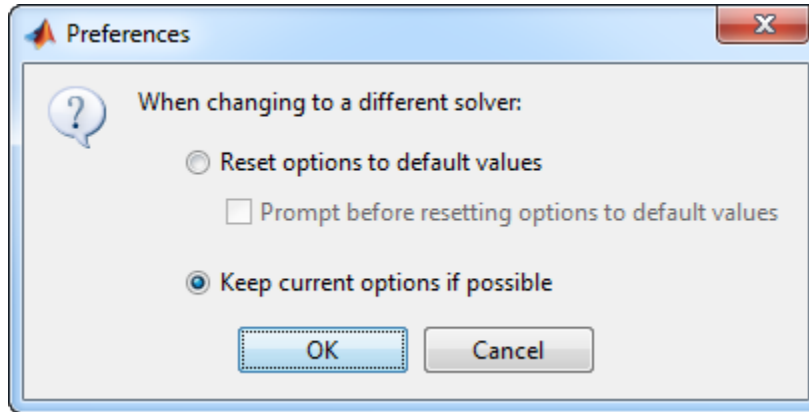
### Resetting Options and Clearing the Problem

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 5-8
- “Output function” on page 5-9
- “Display to Command Window” on page 5-10

#### Plot Functions

You can select a plot function to easily plot various measures of progress while the algorithm executes. Each plot selected draws a separate axis in the figure window. If available for the solver selected, the **Stop** button in the **Run solver and view results** window to interrupt a running solver. You can select a predefined plot function from the



Optimization app, or you can select **Custom function** to write your own. Plot functions not relevant to the solver selected are grayed out. The following lists the available plot functions:

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

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

Write custom plot functions with the same syntax as output functions. For information, see “Output Function Syntax” on page 15-37.

The screenshot shows a dialog box titled "Plot functions". It contains several checkboxes, each followed by a plot function name. The checkboxes for "Current point", "Function count", "Function value", "Max constraint", "Current step", and "First order optimality" are all unchecked and appear to be disabled (grayed out). The checkbox for "Custom function:" is checked. To the right of the "Custom function:" checkbox is an empty text input field.

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

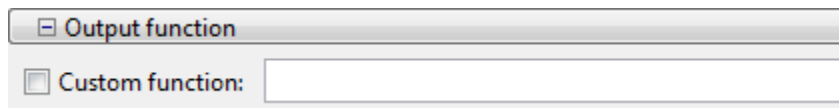
## Output function

**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 Syntax” on page 15-37.



### Display to Command Window

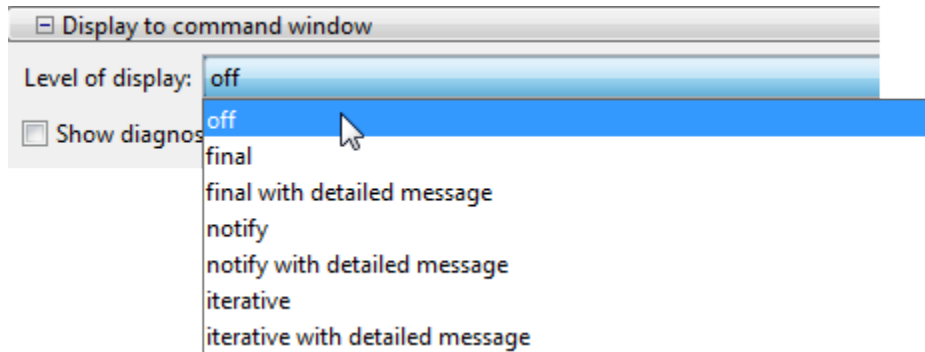
Select **Level of display** to specify the amount of information displayed when you run the algorithm. Choose from the following; depending on the solver, only some may be available:

- `off` (default) — Display no output.
- `final` — Display the reason for stopping at the end of the run.
- `final with detailed message` — Display the detailed reason for stopping at the end of the run.
- `notify` — Display output only if the function does not converge.
- `notify with detailed message` — Display a detailed output only if the function does not converge.
- `iterative` — Display information at each iteration of the algorithm and the reason for stopping at the end of the run.
- `iterative with detailed message` — Display information at each iteration of the algorithm and the detailed reason for stopping at the end of the run.

See “Enhanced Exit Messages” on page 3-5 for information on detailed messages.

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

The graphic below shows the display options for the `fmincon` solver. Some other solvers have fewer options.



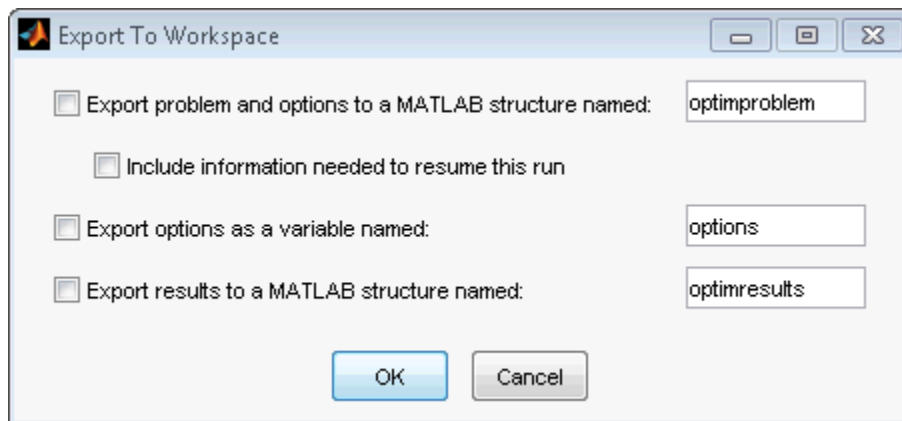
## Importing and Exporting Your Work

- “Exporting Your Work” on page 5-11
- “Importing Your Work” on page 5-13
- “Generating a File” on page 5-13

### 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, to 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 5-15
- “Solver-Based Optimization Problem Setup”

## Optimization App Alternatives

### In this section...

- “Optimize Without Using the App” on page 5-15
- “Set Options Using Live Scripts” on page 5-15
- “Set Options: Command Line or Standard Scripts” on page 5-17
- “Choose Plot Functions” on page 5-19
- “Pass Solver Arguments” on page 5-20

### Optimize Without Using the App

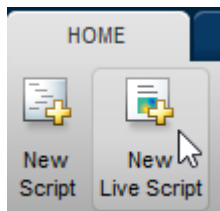
Although the Optimization app affords convenient ways to set options and run problems, it will be removed in a future release. This topic describes other ways to accomplish optimization tasks without using the app.

- Set options easily — “Set Options Using Live Scripts” on page 5-15 or “Set Options: Command Line or Standard Scripts” on page 5-17
- Monitor the optimization — “Choose Plot Functions” on page 5-19
- Pass solver arguments correctly — “Pass Solver Arguments” on page 5-20

### Set Options Using Live Scripts

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

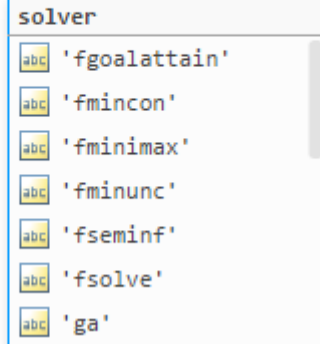
- 1 On the **Home** tab, in the **File** section, click **New Live Script** to create a live script.



- 2 In the Live Editor, set options by entering `options = optimoptions(.MATLAB` shows a list of solvers.

```
options = optimoptions('fgoalattain')
```

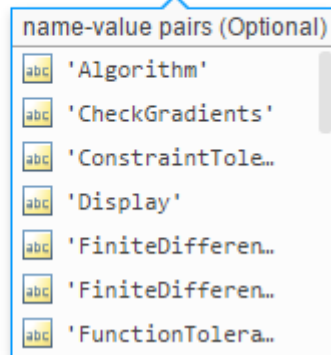
optimoptions(solver,options)



- 3 Select a solver, then enter a comma. MATLAB displays a list of name-value pairs for the solver.

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

optimoptions(solver,options)



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

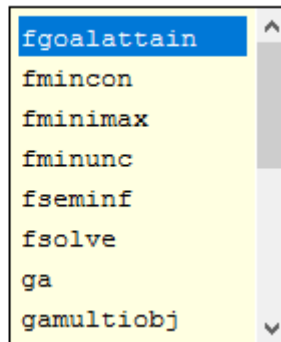
- For help choosing a solver, see “Optimization Decision Table” on page 2-6.
  - For help choosing the solver algorithm, see “Choosing the Algorithm” on page 2-8.
  - To understand the meaning of other options, see “Set Options”.
- 

## 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.

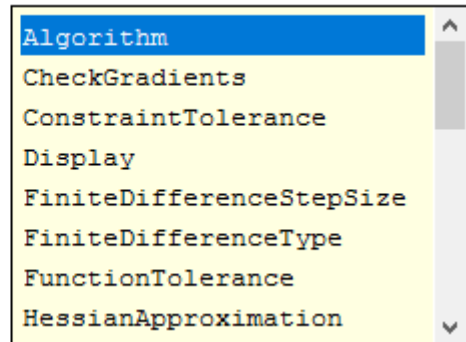
```
>> options = optimoptions('
```



- 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','
```



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)
```

---

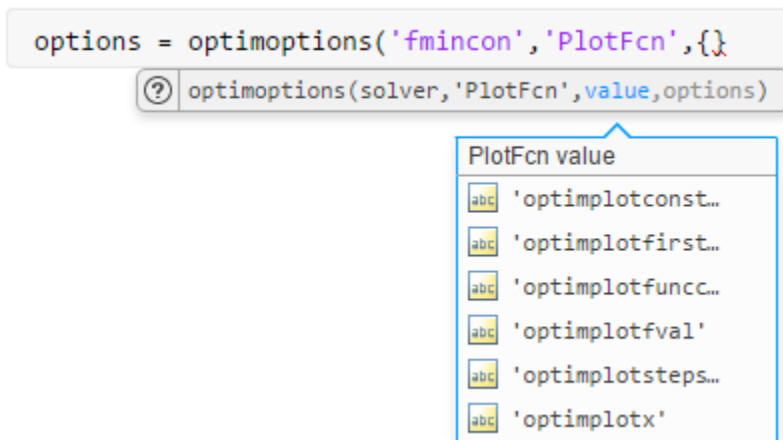
**Tip**

- For help choosing a solver, see “Optimization Decision Table” on page 2-6.
  - For help choosing the solver algorithm, see “Choosing the Algorithm” on page 2-8.
  - To understand the meaning of other options, see “Set Options”.
-

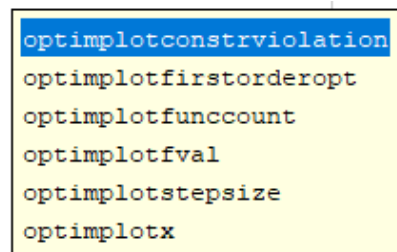
## Choose Plot Functions

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**:



```
options = optimoptions('fmincon', 'PlotFcn', {'optimplot
```

To choose a custom plot function, pass a function handle such as `@myplotfun`. For details on writing a custom plot function, see “Plot Function Syntax” on page 15-47.

`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)
```

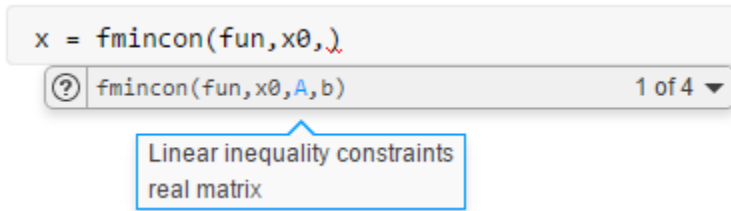
Sometimes the inexperienced will, instead, write

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

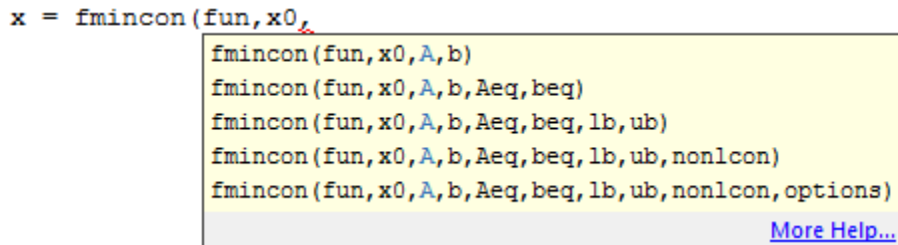
This call throws an error. In this incorrect command, `fmincon` interprets the `lb` argument as representing the `A` matrix, and the `options` argument as representing the `b` vector. The third argument always represents the `A` matrix, and the fourth argument always represents the `b` vector.

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” (MATLAB)
- “Code Sections” (MATLAB)

# Nonlinear algorithms and examples

---

- “Unconstrained Nonlinear Optimization Algorithms” on page 6-2
- “fminsearch Algorithm” on page 6-11
- “fminunc Unconstrained Minimization” on page 6-14
- “Minimization with Gradient and Hessian” on page 6-16
- “Minimization with Gradient and Hessian Sparsity Pattern” on page 6-18
- “Constrained Nonlinear Optimization Algorithms” on page 6-22
- “Tutorial for the Optimization Toolbox™” on page 6-44
- “Banana Function Minimization” on page 6-61
- “Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™” on page 6-72
- “Nonlinear Inequality Constraints” on page 6-79
- “Nonlinear Constraints with Gradients” on page 6-81
- “fmincon Interior-Point Algorithm with Analytic Hessian” on page 6-84
- “Linear or Quadratic Objective with Quadratic Constraints” on page 6-90
- “Nonlinear Equality and Inequality Constraints” on page 6-95
- “Optimization App with the fmincon Solver” on page 6-97
- “Minimization with Bound Constraints and Banded Preconditioner” on page 6-102
- “Minimization with Linear Equality Constraints” on page 6-107
- “Minimization with Dense Structured Hessian, Linear Equalities” on page 6-110
- “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115
- “Using Symbolic Mathematics with Optimization Toolbox™ Solvers” on page 6-129
- “One-Dimensional Semi-Infinite Constraints” on page 6-141
- “Two-Dimensional Semi-Infinite Constraint” on page 6-145
- “Analyzing the Effect of Uncertainty Using Semi-Infinite Programming” on page 6-148

## Unconstrained Nonlinear Optimization Algorithms

<b>In this section...</b>
“Unconstrained Optimization Definition” on page 6-2
“fminunc trust-region Algorithm” on page 6-2
“fminunc quasi-newton Algorithm” on page 6-5

### 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

$$\min \left\{ \frac{1}{2} s^T H s + s^T g \text{ such that } \|D s\| \leq \Delta \right\}, \quad (6-2)$$

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,  $\Delta$  is a positive scalar, and  $\| \cdot \|$  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

$$H \cdot s_2 = -g, \quad (6-3)$$

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  $\Delta$ .

These four steps are repeated until convergence. The trust-region dimension  $\Delta$  is adjusted according to standard rules. In particular, it is decreased if the trial step is not accepted, i.e.,  $f(x + s) \geq f(x)$ . See [46] and [49] for a discussion of this aspect.

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

### Preconditioned Conjugate Gradient Method

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

In a minimization context, you can assume that the Hessian matrix  $H$  is symmetric. However,  $H$  is guaranteed to be positive definite only in the neighborhood of a strong minimizer. Algorithm PCG exits when a direction of negative (or zero) curvature is encountered, i.e.,  $d^T H d \leq 0$ . The PCG output direction,  $p$ , is either a direction of negative curvature or an approximate (*tol* controls how approximate) solution to the Newton system  $Hp = -g$ . In either case  $p$  is used to help define the two-dimensional subspace used in the trust-region approach discussed in “Trust-Region Methods for Nonlinear Minimization” on page 6-2.

## fminunc quasi-newton Algorithm

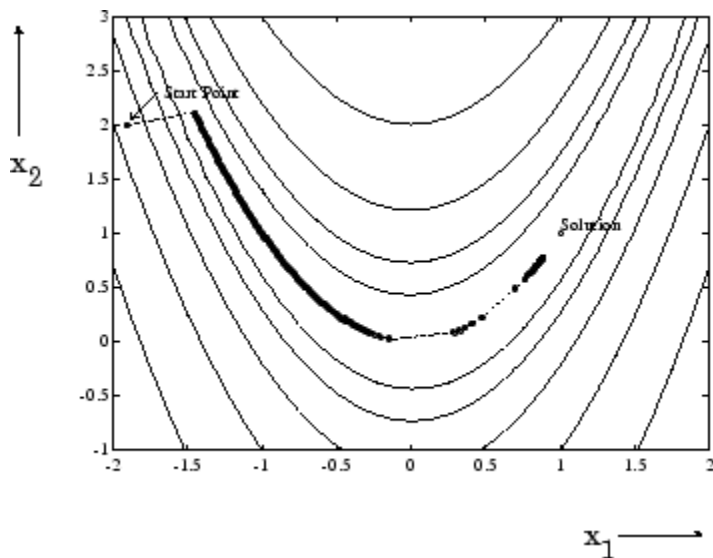
### Basics of Unconstrained Optimization

Although a wide spectrum of methods exists for unconstrained optimization, methods can be broadly categorized in terms of the derivative information that is, or is not, used. Search methods that use only function evaluations (e.g., the simplex search of Nelder and Mead [30]) are most suitable for problems that are not smooth or have a number of discontinuities. Gradient methods are generally more efficient when the function to be minimized is continuous in its first derivative. Higher order methods, such as Newton's method, are only really suitable when the second-order information is readily and easily calculated, because calculation of second-order information, using numerical differentiation, is computationally expensive.

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

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \quad (6-5)$$

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”.

### 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

$$\min_x \frac{1}{2} x^T H x + c^T x + b, \quad (6-6)$$

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.,

$$\nabla f(x^*) = Hx^* + c = 0. \quad (6-7)$$

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

$$x^* = -H^{-1}c. \quad (6-8)$$

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

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k s_k s_k^T H_k^T}{s_k^T H_k s_k}, \quad (6-9)$$

where

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

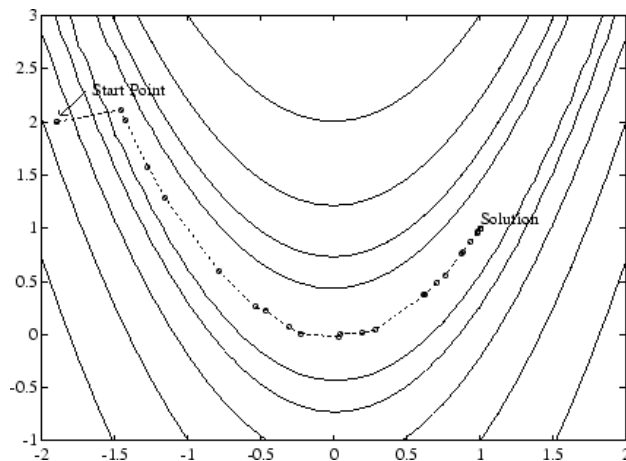
As a starting point,  $H_0$  can be set to any symmetric positive definite matrix, for example, the identity matrix  $I$ . To avoid the inversion of the Hessian  $H$ , you can derive an updating method that avoids the direct inversion of  $H$  by using a formula that makes an approximation of the inverse 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

$$d = -H_k^{-1} \cdot \nabla f(x_k). \quad (6-10)$$

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 6-8. 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".

### Line Search

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

$$x_{k+1} = x_k + \alpha^* d_k, \quad (6-11)$$

where  $x_k$  denotes the current iterate,  $d_k$  is the search direction, and  $\alpha^*$  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  $\alpha$ .
- 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  $\alpha$  satisfies the Wolfe conditions:

$$f(x_k + \alpha d_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T d_k, \quad (6-12)$$

$$\nabla f(x_k + \alpha d_k)^T d_k \geq c_2 \nabla f_k^T d_k, \quad (6-13)$$

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

The first condition (“Equation 6-12”) requires that  $\alpha_k$  sufficiently decreases the objective function. The second condition (“Equation 6-13”) ensures that the step length is not too small. Points that satisfy both conditions (“Equation 6-12” and “Equation 6-13”) are called *acceptable points*.

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

### Hessian Update

Many of the optimization functions determine the direction of search by updating the Hessian matrix at each iteration, using the BFGS method (“Equation 6-9”). The function `fminunc` also provides an option to use the DFP method given in “Quasi-Newton Methods” on page 6-6 (set `HessUpdate` to ‘`dfp`’ in `options` to select the DFP method). The Hessian,  $H$ , is always maintained to be positive definite so that the direction of search,  $d$ , is always in a descent direction. This means that for some arbitrarily small step  $\alpha$  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  $\alpha_k$  and a combination of the search direction  $d$  with past and present gradient evaluations,

$$q_k^T s_k = \alpha_k \left( \nabla f(x_{k+1})^T d - \nabla f(x_k)^T d \right). \quad (6-14)$$

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  $\alpha_k$  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 6-11



## fminsearch Algorithm

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

---

**Note** The keywords for the fminsearch iterative display appear in **bold** after the description of the step.

---

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

$$m = \frac{\sum x(i)}{n}, \quad i = 1 \dots n,$$

and calculate  $f(r)$ .

- 4 If  $f(x(1)) \leq f(r) < f(x(n))$ , accept  $r$  and terminate this iteration. **Reflect**
- 5 If  $f(r) < f(x(1))$ , calculate the expansion point  $s$

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

and calculate  $f(s)$ .

- a If  $f(s) < f(r)$ , accept  $s$  and terminate the iteration. **Expand**
  - b Otherwise, accept  $r$  and terminate the iteration. **Reflect**
- 6 If  $f(r) \geq f(x(n))$ , perform a contraction between  $m$  and the better of  $x(n+1)$  and  $r$ :

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

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

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

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

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

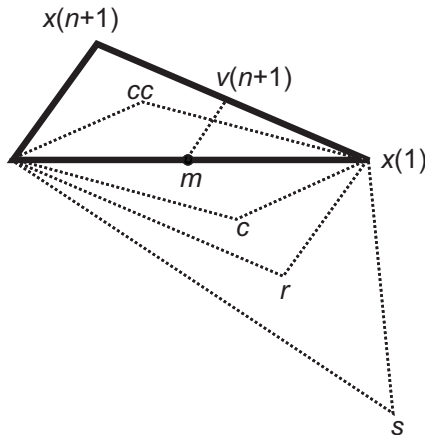
and calculate  $f(cc)$ . If  $f(cc) < f(x(n+1))$ , accept  $cc$  and terminate the iteration. **Contract inside** Otherwise, continue with Step 7 (Shrink).

7 Calculate the  $n$  points

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

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

## fminunc Unconstrained Minimization

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

$$\min_x f(x) = e^{x_1} (4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1). \quad (6-15)$$

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

### Step 1: Write a file `objfun.m`.

This code ships with the toolbox. To view, enter `type objfun`:

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

### Step 2: Set options.

Set options to use the 'quasi-newton' algorithm. Set options because the 'trust-region' algorithm requires that the objective function include a gradient. If you do not set the options, then, depending on your MATLAB version, `fminunc` can issue a warning.

```
options = optimoptions(@fminunc,'Algorithm','quasi-newton');
```

### Step 3: Invoke `fminunc` using the options.

```
x0 = [-1,1]; % Starting guess
[x,fval,exitflag,output] = fminunc(@objfun,x0,options);
```

This produces the following output:

```
Local minimum found.
```

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

View the results, including the first-order optimality measure in the `output` structure.

```
x,fval,exitflag,output.firstorderopt
```

```
x =  
    0.5000   -1.0000  
  
fval =  
    3.6609e-16  
  
exitflag =  
    1  
  
ans =  
    7.3704e-08
```

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

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

## See Also

### Related Examples

- “Minimization with Gradient and Hessian” on page 6-16

### 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

$$f(x) = \sum_{i=1}^{n-1} \left( (x_i^2)(x_{i+1}^2 + 1) + (x_{i+1}^2)(x_i^2 + 1) \right), \quad (6-16)$$

where  $n = 1000$ .

### Step 1: Write a file `brownfgh.m` that computes the objective function, the gradient of the objective, and the sparse tridiagonal Hessian matrix.

The file is lengthy so is not included here. View the code with the command

```
type brownfgh
```

Because `brownfgh` computes the gradient and Hessian values as well as the objective function, you need to use `optimoptions` to indicate that this information is available in `brownfgh`, using the `SpecifyObjectiveGradient` and `HessianFcn` options.

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

```
n = 1000;
xstart = -ones(n,1);
xstart(2:2:n,1) = 1;
options = optimoptions(@fminunc,'Algorithm','trust-region',...
    'SpecifyObjectiveGradient',true,'HessianFcn','objective');
[x,fval,exitflag,output] = fminunc(@brownfgh,xstart,options);
```

This 1000 variable problem is solved in about 7 iterations and 7 conjugate gradient iterations with a positive `exitflag` indicating convergence. The final function value and measure of optimality at the solution  $x$  are both close to zero. For `fminunc`, the first order optimality is the infinity norm of the gradient of the function, which is zero at a local minimum:

```
fval,exitflag,output.firstorderopt
```

```
fval =
```

```
    2.8709e-17
```

```
exitflag =
```

```
    1
```

```
ans =
```

```
    4.7948e-10
```

## See Also

### Related Examples

- “Minimization with Gradient and Hessian Sparsity Pattern” on page 6-18

## Minimization with Gradient and Hessian Sparsity Pattern

This example shows how to solve a nonlinear minimization problem with tridiagonal Hessian matrix approximated by sparse finite differences instead of explicit computation.

The problem is to find  $x$  to minimize

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

where  $n = 1000$ .

To use the trust-region method in `fminunc`, you *must* compute the gradient in `fun`; it is *not* optional as in the quasi-newton method.

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

### Step 1: Write a file `brownfg.m` that computes the objective function and the gradient of the objective.

This function file ships with your software.

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



```

        log(x(i).^2) + ...
        2*(x(i).^2+1).*x(i+1).* ...
        ((x(i+1).^2).^(x(i).^2));
    end

```

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

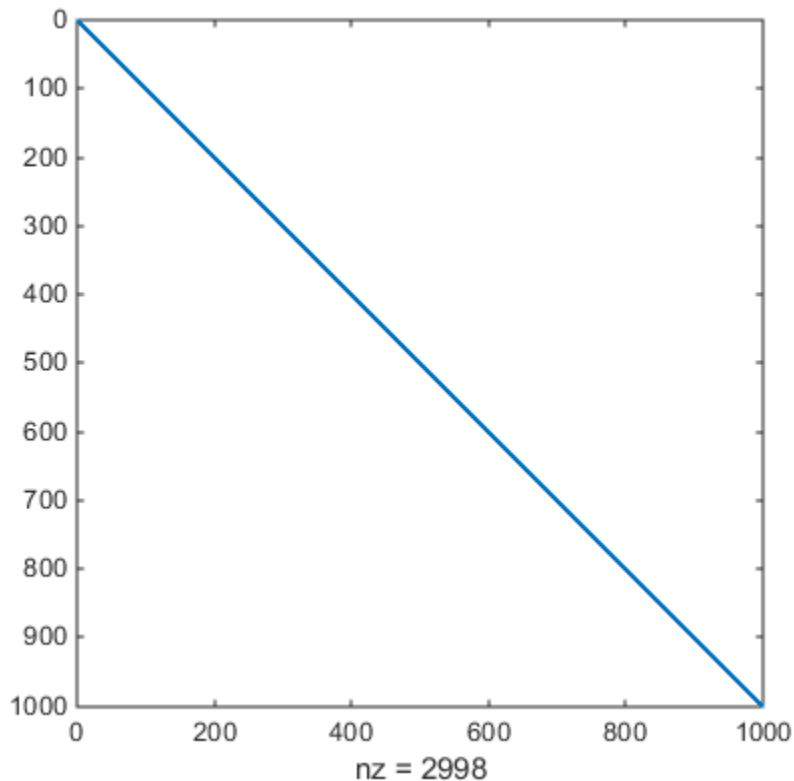
You must also set the `SpecifyObjectiveGradient` option to `true` using `optimoptions`, since the gradient is computed in `brownfg.m`. Then execute `fminunc` as shown in Step 2.

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

```

fun = @brownfg;
load brownhstr % Get Hstr, structure of the Hessian
spy(Hstr) % View the sparsity structure of Hstr

```



```
n = 1000;
xstart = -ones(n,1);
xstart(2:2:n,1) = 1;
options = optimoptions(@fminunc,'Algorithm','trust-region',...
    'SpecifyObjectiveGradient',true,'HessPattern','Hstr');
[x,fval,exitflag,output] = fminunc(fun,xstart,options);
```

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

```
exitflag, fval, output.firstorderopt
```

```
exitflag =
```

```
    1
```

```
fval =
```

```
    7.4738e-17
```

```
ans =
```

```
    7.9822e-10
```

## See Also

### Related Examples

- “Minimization with Gradient and Hessian” on page 6-16

## Constrained Nonlinear Optimization Algorithms

### In this section...

“Constrained Optimization Definition” on page 6-22

“fmincon Trust Region Reflective Algorithm” on page 6-22

“fmincon Active Set Algorithm” on page 6-27

“fmincon SQP Algorithm” on page 6-36

“fmincon Interior Point Algorithm” on page 6-37

“fminbnd Algorithm” on page 6-40

“fseminf Problem Formulation and Algorithm” on page 6-41

### Constrained Optimization Definition

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

$$\min_x f(x)$$

such that one or more of the following holds:  $c(x) \leq 0$ ,  $ceq(x) = 0$ ,  $A \cdot x \leq b$ ,  $Aeq \cdot x = beq$ ,  $l \leq x \leq u$ . There are even more constraints used in semi-infinite programming; see “fseminf Problem Formulation and Algorithm” on page 6-41.

### fmincon Trust Region Reflective Algorithm

#### Trust-Region Methods for Nonlinear Minimization

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

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

$$\min_s \{q(s), s \in N\}. \quad (6-17)$$

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

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

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

$$\min \left\{ \frac{1}{2} s^T H s + s^T g \text{ such that } \|Ds\| \leq \Delta \right\}, \quad (6-18)$$

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,  $\Delta$  is a positive scalar, and  $\| \cdot \|$  is the 2-norm. Good algorithms exist for solving “Equation 6-18” (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 6-18”. However, they require time proportional to several factorizations of  $H$ . Therefore, for large-scale problems a different approach is needed. Several approximation and heuristic strategies, based on “Equation 6-18”, have been proposed in the literature ([42] and [50]). The approximation approach followed in Optimization Toolbox solvers is to restrict the trust-region subproblem to a two-dimensional subspace  $S$  ([39] and [42]). Once the subspace  $S$  has been computed, the work to solve “Equation 6-18” is trivial even if full eigenvalue/eigenvector information is needed (since in the subspace, the problem is only two-dimensional). The dominant work has now shifted to the determination of the subspace.

The two-dimensional subspace  $S$  is determined with the aid of a preconditioned conjugate gradient process described below. The solver defines  $S$  as the linear space spanned by  $s_1$

and  $s_2$ , where  $s_1$  is in the direction of the gradient  $g$ , and  $s_2$  is either an approximate Newton direction, i.e., a solution to

$$H \cdot s_2 = -g, \tag{6-19}$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \tag{6-20}$$

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

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

- 1 Formulate the two-dimensional trust-region subproblem.
- 2 Solve "Equation 6-18" to determine the trial step  $s$ .
- 3 If  $f(x + s) < f(x)$ , then  $x = x + s$ .
- 4 Adjust  $\Delta$ .

These four steps are repeated until convergence. The trust-region dimension  $\Delta$  is adjusted according to standard rules. In particular, it is decreased if the trial step is not accepted, i.e.,  $f(x + s) \geq f(x)$ . See [46] and [49] for a discussion of this aspect.

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

### Preconditioned Conjugate Gradient Method

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

In a minimization context, you can assume that the Hessian matrix  $H$  is symmetric. However,  $H$  is guaranteed to be positive definite only in the neighborhood of a strong minimizer. Algorithm PCG exits when a direction of negative (or zero) curvature is

encountered, i.e.,  $d^T H d \leq 0$ . The PCG output direction,  $p$ , is either a direction of negative curvature or an approximate ( $tol$  controls how approximate) solution to the Newton system  $H p = -g$ . In either case  $p$  is used to help define the two-dimensional subspace used in the trust-region approach discussed in “Trust-Region Methods for Nonlinear Minimization” on page 6-2.

### Linear Equality Constraints

Linear constraints complicate the situation described for unconstrained minimization. However, the underlying ideas described previously can be carried through in a clean and efficient way. The trust-region methods in Optimization Toolbox solvers generate strictly feasible iterates.

The general linear equality constrained minimization problem can be written

$$\min\{f(x) \text{ such that } Ax = b\}, \quad (6-21)$$

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

The method used to solve “Equation 6-21” differs from the unconstrained approach in two significant ways. First, an initial feasible point  $x_0$  is computed, using a sparse least-squares step, so that  $Ax_0 = b$ . Second, Algorithm PCG is replaced with Reduced Preconditioned Conjugate Gradients (RPCG), see [46], in order to compute an approximate reduced Newton step (or a direction of negative curvature in the null space of  $A$ ). The key linear algebra step involves solving systems of the form

$$\begin{bmatrix} C & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \quad (6-22)$$

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

$$\min\{f(x) \text{ such that } l \leq x \leq u\}, \quad (6-23)$$

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  $\infty$ . The method generates a sequence of strictly feasible points. Two techniques are used to maintain feasibility while achieving robust convergence behavior. First, a scaled modified Newton step replaces the unconstrained Newton step (to define the two-dimensional subspace  $S$ ). Second, reflections are used to increase the step size.

The scaled modified Newton step arises from examining the Kuhn-Tucker necessary conditions for “Equation 6-23”,

$$(D(x))^{-2}g = 0, \tag{6-24}$$

where

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

and the vector  $v(x)$  is defined below, for each  $1 \leq i \leq n$ :

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

The nonlinear system “Equation 6-24” is not differentiable everywhere.

Nondifferentiability occurs when  $v_i = 0$ . You can avoid such points by maintaining strict feasibility, i.e., restricting  $l < x < u$ .

The scaled modified Newton step  $s_k$  for the nonlinear system of equations given by “Equation 6-24” is defined as the solution to the linear system

$$\widehat{M}Ds^N = -\widehat{g} \tag{6-25}$$

at the  $k$ th iteration, where

$$\widehat{g} = D^{-1}g = \text{diag}(|v|^{1/2})g, \tag{6-26}$$

and

$$\widehat{M} = D^{-1}HD^{-1} + \text{diag}(g)J^v. \tag{6-27}$$



Here  $J^v$  plays the role of the Jacobian of  $|v|$ . Each diagonal component of the diagonal matrix  $J^v$  equals 0, -1, or 1. If all the components of  $l$  and  $u$  are finite,  $J^v = \text{diag}(\text{sign}(g))$ . At a point where  $g_i = 0$ ,  $v_i$  might not be differentiable.  $J_{ii}^v = 0$  is defined at such a point. Nondifferentiability of this type is not a cause for concern because, for such a component, it is not significant which value  $v_i$  takes. Further,  $|v_i|$  will still be discontinuous at this point, but the function  $|v_i| \cdot g_i$  is continuous.

Second, reflections are used to increase the step size. A (single) reflection step is defined as follows. Given a step  $p$  that intersects a bound constraint, consider the first bound constraint crossed by  $p$ ; assume it is the  $i$ th bound constraint (either the  $i$ th upper or  $i$ th lower bound). Then the reflection step  $p^R = p$  except in the  $i$ th component, where  $p_i^R = -p_i$ .

## 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  $G_i(x)$ ,  $i = 1, \dots, m$ , 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

$$\begin{aligned} \nabla f(x^*) + \sum_{i=1}^m \lambda_i \cdot \nabla G_i(x^*) &= 0 \\ \lambda_i \cdot G_i(x^*) &= 0, \quad i = 1, \dots, m_e \\ \lambda_i &\geq 0, \quad i = m_e + 1, \dots, m, \end{aligned} \tag{6-28}$$

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 ( $\lambda_i$ ,  $i = 1, \dots, m$ ) 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 2-14.

### **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.

$$L(x, \lambda) = f(x) + \sum_{i=1}^m \lambda_i \cdot g_i(x). \quad (6-29)$$

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

$$\begin{aligned}
 \min_{d \in \mathbb{R}^n} \quad & \frac{1}{2}d^T H_k d + \nabla f(x_k)^T d \\
 \nabla g_i(x_k)^T d + g_i(x_k) = 0, \quad & i = 1, \dots, m_e \\
 \nabla g_i(x_k)^T d + g_i(x_k) \leq 0, \quad & i = m_e + 1, \dots, m.
 \end{aligned} \tag{6-30}$$

This subproblem can be solved using any QP algorithm (see, for instance, “Quadratic Programming Solution” on page 6-32). The solution is used to form a new iterate

$$x_k + d = x_k + \alpha_k d_k.$$

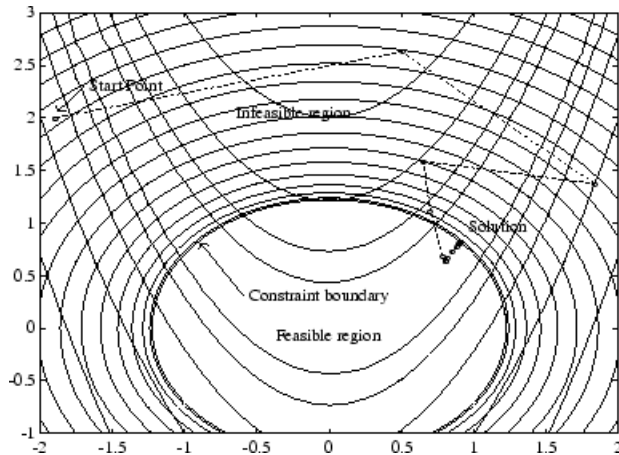
The step length parameter  $\alpha_k$  is determined by an appropriate line search procedure so that a sufficient decrease in a merit function is obtained (see “Updating the Hessian Matrix” on page 6-30). The matrix  $H_k$  is a positive definite approximation of the Hessian matrix of the Lagrangian function (“Equation 6-29”).  $H_k$  can be updated by any of the quasi-Newton methods, although the BFGS method (see “Updating the Hessian Matrix” on page 6-30) appears to be the most popular.

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

Consider Rosenbrock's function with an additional nonlinear inequality constraint,  $g(x)$ ,

$$x_1^2 + x_2^2 - 1.5 \leq 0. \tag{6-31}$$

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 6-30 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 6-30
- “Quadratic Programming Solution” on page 6-32
- “Initialization” on page 6-34
- “Line Search and Merit Function” on page 6-35

### Updating the Hessian Matrix

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

$$H_{k+1} = H_k + \frac{q_k q_k^T}{q_k^T s_k} - \frac{H_k s_k s_k^T H_k^T}{s_k^T H_k s_k}, \quad (6-32)$$

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  $q_k^T s_k$  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,

$$q_k = q_k + wv, \tag{6-33}$$

where

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

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

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

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

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

### Quadratic Programming Solution

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

$$\begin{aligned} \min_{d \in \mathfrak{R}^n} q(d) &= \frac{1}{2}d^T H d + c^T d, \\ A_i d &= b_i, \quad i = 1, \dots, m_e \\ A_i d &\leq b_i, \quad i = m_e + 1, \dots, m. \end{aligned} \tag{6-34}$$

The method used in Optimization Toolbox functions is an active set strategy (also known as a projection method) similar to that of Gill et al., described in [18] and [17]. It has been modified for both Linear Programming (LP) and Quadratic Programming (QP) problems.

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

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

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

$$Z_k = Q[:, l + 1 : m], \tag{6-35}$$

where

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

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

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

$$q(p) = \frac{1}{2} p^T Z_k^T H Z_k p + c^T Z_k p. \quad (6-36)$$

Differentiating this with respect to  $p$  yields

$$\nabla q(p) = Z_k^T H Z_k p + Z_k^T c. \quad (6-37)$$

$\nabla q(p)$  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

$$Z_k^T H Z_k p = -Z_k^T c. \quad (6-38)$$

A step is then taken of the form

$$x_{k+1} = x_k + \alpha \hat{d}_k, \quad \text{where } \hat{d}_k = Z_k p. \quad (6-39)$$

At each iteration, because of the quadratic nature of the objective function, there are only two choices of step length  $\alpha$ . A step of unity along  $\hat{d}_k$  is the exact step to the minimum of the function restricted to the null space of  $\bar{A}_k$ . If such a step can be taken, without violation of the constraints, then this is the solution to QP ("Equation 6-34"). Otherwise, the step along  $\hat{d}_k$  to the nearest constraint is less than unity and a new constraint is included in the active set at the next iteration. The distance to the constraint boundaries in any direction  $\hat{d}_k$  is given by

$$\alpha = \min_{i \in \{1, \dots, m\}} \left\{ \frac{-(A_i x_k - b_i)}{A_i \widehat{d}_k} \right\}, \quad (6-40)$$

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

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

$$\overline{A}_k^T \lambda_k = c. \quad (6-41)$$

If all elements of  $\lambda_k$  are positive,  $x_k$  is the optimal solution of QP ("Equation 6-34"). However, if any component of  $\lambda_k$  is negative, and the component does not correspond to an equality constraint, then the corresponding element is deleted from the active set and a new iterate is sought.

**Initialization**

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

$$\begin{aligned} & \min \quad \gamma \text{ such that} \\ & \gamma \in \Re, x \in \Re^n \\ & A_i x = b_i, \quad i = 1, \dots, m_e \\ & A_i x - \gamma \leq b_i, \quad i = m_e + 1, \dots, m. \end{aligned} \quad (6-42)$$

The notation  $A_i$  indicates the  $i$ th row of the matrix  $A$ . You can find a feasible point (if one exists) to "Equation 6-42" by setting  $x$  to a value that satisfies the equality constraints. You can determine this value by solving an under- or overdetermined set of linear equations formed from the set of equality constraints. If there is a solution to this problem, then the slack variable  $\gamma$  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).

$$\widehat{d}_k = -Z_k Z_k^T g_k. \quad (6-43)$$



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

$$H\hat{d}_1 = -g_k, \quad (6-44)$$

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

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

### Line Search and Merit Function

The solution to the QP subproblem produces a vector  $d_k$ , which is used to form a new iterate

$$x_{k+1} = x_k + \alpha d_k. \quad (6-45)$$

The step length parameter  $\alpha_k$  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.

$$\Psi(x) = f(x) + \sum_{i=1}^{m_e} r_i \cdot g_i(x) + \sum_{i=m_e+1}^m r_i \cdot \max[0, g_i(x)]. \quad (6-46)$$

Powell recommends setting the penalty parameter

$$r_i = (r_{k+1})_i = \max_i \left\{ \lambda_i, \frac{(r_k)_i + \lambda_i}{2} \right\}, \quad i = 1, \dots, m. \quad (6-47)$$

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

$$r_i = \frac{\|\nabla f(x)\|}{\|\nabla g_i(x)\|}, \quad (6-48)$$

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

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

## **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 6-27). 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 6-30”. These routines are more efficient in both memory usage and speed than the active-set routines.

### **Reformulated Feasibility Routines**

The `sqp` algorithm has two new approaches to the solution of “Equation 6-30” when constraints are not satisfied.

- The `sqp` algorithm combines the objective and constraint functions into a merit function. The algorithm attempts to minimize the merit function subject to relaxed constraints. This modified problem can lead to a feasible solution. However, this approach has more variables than the original problem, so the problem size in “Equation 6-30” increases. The increased size can slow the solution of the subproblem.

These routines are based on the articles by Spellucci [60] and Tone [61]. The `sqp` algorithm sets the penalty parameter for the merit function “Equation 6-46” according to the suggestion in [41].

- Suppose nonlinear constraints are not satisfied, and an attempted step causes the constraint violation to grow. The `sqp` algorithm attempts to obtain feasibility using a second-order approximation to the constraints. The second-order technique can lead to a feasible solution. However, this technique can slow the solution by requiring more evaluations of the nonlinear constraint functions.

## fmincon Interior Point Algorithm

### Barrier Function

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

$$\min_x f(x), \text{ subject to } h(x) = 0 \text{ and } g(x) \leq 0. \quad (6-49)$$

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

$$\min_{x,s} f_\mu(x,s) = \min_{x,s} f(x) - \mu \sum_i \ln(s_i), \text{ subject to } h(x) = 0 \text{ and } g(x) + s = 0. \quad (6-50)$$

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  $\mu$  decreases to zero, the minimum of  $f_\mu$  should approach the minimum of  $f$ . The added logarithmic term is called a barrier function. This method is described in [40], [41], and [51].

The approximate problem “Equation 6-50” is a sequence of equality constrained problems. These are easier to solve than the original inequality-constrained problem “Equation 6-49”.

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

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

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

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

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

The parameter  $\nu$  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  $f_{\mu}$ :

$$H = \nabla^2 f(x) + \sum_i \lambda_i \nabla^2 g_i(x) + \sum_j \lambda_j \nabla^2 h_j(x). \quad (6-51)$$

- $J_g$  denotes the Jacobian of the constraint function  $g$ .
- $J_h$  denotes the Jacobian of the constraint function  $h$ .
- $S = \text{diag}(s)$ .
- $\lambda$  denotes the Lagrange multiplier vector associated with constraints  $g$
- $\Lambda = \text{diag}(\lambda)$ .
- $y$  denotes the Lagrange multiplier vector associated with  $h$ .

- $e$  denote the vector of ones the same size as  $g$ .

“Equation 6-52” defines the direct step  $(\Delta x, \Delta s)$ :

$$\begin{bmatrix} H & 0 & J_h^T & J_g^T \\ 0 & S\Lambda & 0 & -S \\ J_h & 0 & I & 0 \\ J_g & -S & 0 & I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \\ -\Delta y \\ -\Delta \lambda \end{bmatrix} = - \begin{bmatrix} \nabla f - J_h^T y - J_g^T \lambda \\ S\lambda - \mu e \\ h \\ g + s \end{bmatrix}. \quad (6-52)$$

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 (MATLAB) in the MATLAB `ldl` function reference page.) This is the most computationally expensive step. One result of this factorization is a determination of whether the projected Hessian is positive definite or not; if not, the algorithm uses a conjugate gradient step, described in the next section.

### Conjugate Gradient Step

The conjugate gradient approach to solving the approximate problem “Equation 6-50” is similar to other conjugate gradient calculations. In this case, the algorithm adjusts both  $x$  and  $s$ , keeping the slacks  $s$  positive. The approach is to minimize a quadratic approximation to the approximate problem in a trust region, subject to linearized constraints.

Specifically, let  $R$  denote the radius of the trust region, and let other variables be defined as in “Direct Step” on page 6-38. The algorithm obtains Lagrange multipliers by approximately solving the KKT equations

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

in the least-squares sense, subject to  $\lambda$  being positive. Then it takes a step  $(\Delta x, \Delta s)$  to approximately solve

$$\min_{\Delta x, \Delta s} \nabla f^T \Delta x + \frac{1}{2} \Delta x^T \nabla_{xx}^2 L \Delta x + \mu e^T S^{-1} \Delta s + \frac{1}{2} \Delta s^T S^{-1} \Lambda \Delta s, \quad (6-53)$$

subject to the linearized constraints

$$g(x) + J_g \Delta x + \Delta s = 0, \quad h(x) + J_h \Delta x = 0. \quad (6-54)$$

To solve “Equation 6-54”, the algorithm tries to minimize a norm of the linearized constraints inside a region with radius scaled by  $R$ . Then “Equation 6-53” is solved with the constraints being to match the residual from solving “Equation 6-54”, staying within the trust region of radius  $R$ , and keeping  $s$  strictly positive. For details of the algorithm and the derivation, see [40], [41], and [51]. For another description of conjugate gradients, see “Preconditioned Conjugate Gradient Method” on page 6-24.

### 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 2-28.
- **HessianMultiplyFcn** — Give a separate function for Hessian-times-vector evaluation. For details, see “Including Hessians” on page 2-28 and “Hessian Multiply Function” on page 2-31.
- **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

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

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

$$\max_{w_j \in I_j} K_j(x, w_j) \leq 0 \quad \text{for all } j = 1, \dots, |K|, \quad (6-55)$$

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  $k_j(x, w_j)$  to the functions  $K_j(x, w_j)$ , for each  $x$  that the solver visits.

`fseminf` considers only the maxima of the interpolation function  $\kappa_j(x, w_j)$ , instead of  $K_j(x, w_j)$ , in "Equation 6-55". This reduces the original problem, minimizing a semi-infinately 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  $j$ th 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 the Optimization Toolbox™

This example shows how to use two nonlinear optimization solvers and how to set options. The nonlinear solvers that we use in this example are `fminunc` and `fmincon`.

All the principles outlined in this example apply to the other nonlinear solvers, such as `fgoalattain`, `fminimax`, `lsqnonlin`, `lsqcurvefit`, and `fsolve`.

The example starts with minimizing an objective function, then proceeds to minimize the same function with additional parameters. After that, the example shows how to minimize the objective function when there is a constraint, and finally shows how to get a more efficient and/or accurate solution by providing gradients or Hessian, or by changing some options.

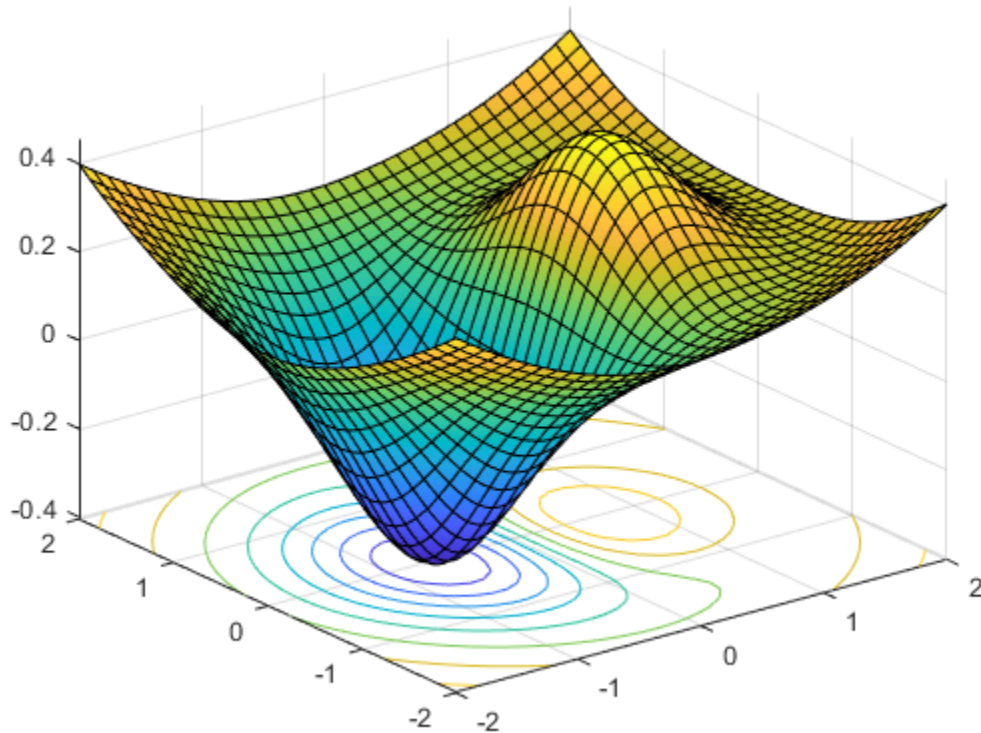
### Unconstrained Optimization Example

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 get an idea of 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. For now, this function is simple enough to define as an anonymous function:

```
fun = @(x) f(x(1),x(2));
```

Take a guess at the solution:

```
x0 = [-.5; 0];
```

Set optimization options to not use `fminunc`'s default large-scale algorithm, since that algorithm requires the objective function gradient to be provided:

```
options = optimoptions('fminunc','Algorithm','quasi-newton');
```

View the iterations as the solver calculates:

```
options.Display = 'iter';
```

Call `fminunc`, an unconstrained nonlinear minimizer:

```
[x, fval, exitflag, output] = fminunc(fun,x0,options);
```

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

The solver found a solution at:

```
uncx = x
```

```
uncx = 2×1
```

```
-0.6691  
0.0000
```

The function value at the solution is:

```
uncf = fval
```

```
uncf = -0.4052
```

We will use the number of function evaluations as a measure of efficiency in this example. The total number of function evaluations is:

```
output.funcCount
```

```
ans = 18
```

## Unconstrained Optimization with Additional Parameters

We will now pass extra parameters as additional arguments to the objective function. We show two different ways of doing this - using a MATLAB file, or using a nested function.

Consider the objective function from the previous section:

$$f(x, y) = x \exp(- (x^2 + y^2)) + (x^2 + y^2)/20.$$

We parametrize the function with (a,b,c) in the following way:

$$f(x, y, a, b, c) = (x - a) \exp(- ((x - a)^2 + (y - b)^2)) + ((x - a)^2 + (y - b)^2)/c.$$

This function is a shifted and scaled version of the original objective function.

### Method 1: MATLAB file Function

Suppose we have a MATLAB file objective function called `bowlpeakfun` defined as:

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
```

### Method 2: Nested Function

Consider the following function that 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
```

In this method, the parameters (a,b,c) are visible to the nested objective function called `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
```

You can see both methods produced identical answers, so use whichever one you find most convenient.

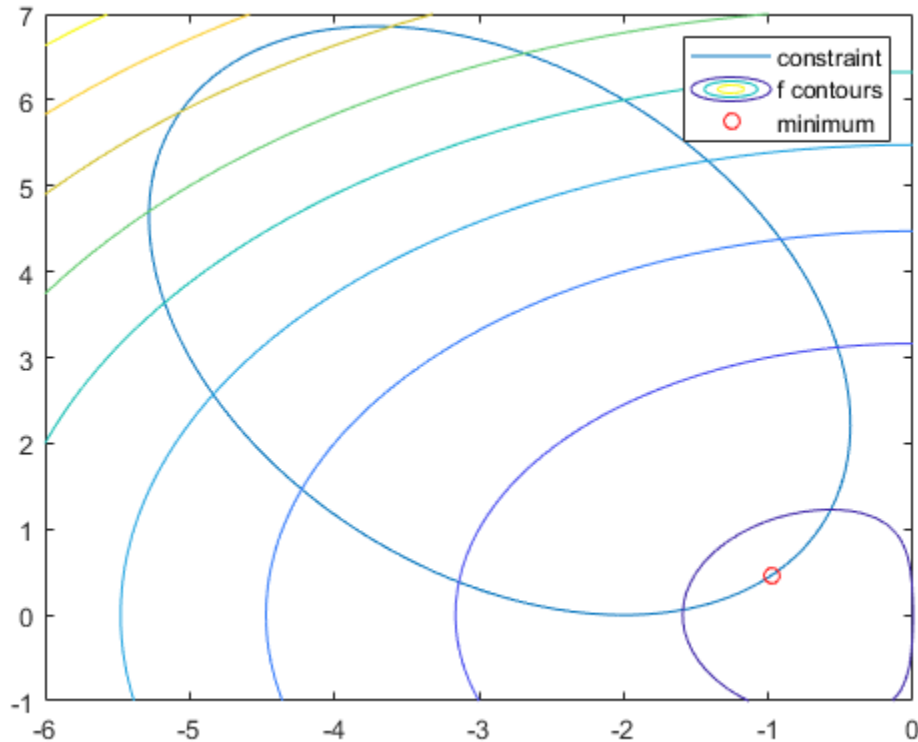
### Constrained Optimization Example: Inequalities

Consider the above problem with a constraint:

$$\begin{aligned} &\text{minimize } x \exp(-x^2 + y^2) + (x^2 + y^2)/20, \\ &\text{subject to } xy/2 + (x + 2)^2 + (y - 2)^2/2 \leq 2. \end{aligned}$$

The constraint set is the interior of a tilted ellipse. Look at the contours of the objective function plotted together with the tilted ellipse

```
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. We are about to calculate the minimum that was just plotted. Take a guess at the solution:

```
x0 = [-2 1];
```

Set optimization options: use the interior-point algorithm, and turn on the display of results at each iteration:

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

Solvers require that nonlinear constraint functions give two outputs: one for nonlinear inequalities, the second for nonlinear equalities. So we write the constraint using the `deal` function to give both outputs:



```
gfun = @(x) deal(g(x(1),x(2)),[]);
```

Call the nonlinear constrained solver. There are no linear equalities or inequalities or bounds, so pass `[]` for those arguments:

```
[x,fval,exitflag,output] = fmincon(fun,x0,[],[],[],[],[],[],gfun,options);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of 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.

A solution to this problem has been found at:

```
x
```

```
x = 1×2
```

```
-0.9727    0.4686
```

The function value at the solution is:

```
fval
```

```
fval = -0.2449
```

The total number of function evaluations was:

```
Fevals = output.funcCount
```

```
Fevals = 38
```

The inequality constraint is satisfied at the solution.

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

```
c = -2.4608e-06
```

```
ceq =
```

```
 []
```

Since  $c(x)$  is close to 0, the constraint is "active," meaning the constraint affects the solution. Recall the unconstrained solution was found at

```
uncx
```

```
uncx = 2×1
```

```
 -0.6691
```

```
  0.0000
```

and the unconstrained objective function was found to be

```
uncf
```

```
uncf = -0.4052
```

The constraint moved the solution, and increased the objective by

```
fval-uncf
```

```
ans = 0.1603
```

### **Constrained Optimization Example: User-Supplied Gradients**

Optimization problems can be solved more efficiently and accurately if gradients are supplied by the user. This example shows how this may be performed. We again solve the inequality-constrained problem

$$\text{minimize } x \exp(-(x^2 + y^2)) + (x^2 + y^2)/20,$$

$$\text{subject to } xy/2 + (x + 2)^2 + (y - 2)^2/2 \leq 2.$$

To provide the gradient of  $f(x)$  to `fmincon`, we 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 nargin > 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 nargin > 2
    gc = [x(2)/2+2*(x(1)+2);
          x(1)/2+x(2)-2];
    gceq = [];
end
```

Make a guess at the solution:

```
x0 = [-2; 1];
```

Set optimization options: we continue to use the same algorithm for comparison purposes.

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

We also 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);
```

There should be fewer function counts this time, since `fmincon` does not need to estimate gradients using finite differences.

```
options.Display = 'iter';
```

Call the solver:

```
[x,fval,exitflag,output] = fmincon(@onehump,x0,[],[],[],[],[],[], ...  
    @tiltellipse,options);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of 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 the current example are similar.

The solution to this problem has been found at:

```
xold = x
```

```
xold = 2×1
    -0.9727
     0.4686
```

The function value at the solution is:

```
minfval = fval
minfval = -0.2449
```

The total number of function evaluations was:

```
Fgradevals = output.funcCount
Fgradevals = 14
```

Compare this to the number of function evaluations without gradients:

```
Fevals
Fevals = 38
```

### Changing the Default Termination Tolerances

This time we solve the same constrained problem

$$\begin{aligned} &\text{minimize } x \exp(-(x^2 + y^2)) + (x^2 + y^2)/20, \\ &\text{subject to } xy/2 + (x + 2)^2 + (y - 2)^2/2 \leq 2, \end{aligned}$$

more accurately by overriding the default termination criteria (`options.StepTolerance` and `options.OptimalityTolerance`). We continue to use gradients. The default values for `fmincon`'s interior-point algorithm are `options.StepTolerance = 1e-10`, `options.OptimalityTolerance = 1e-6`.

Override two default termination criteria: termination tolerances on `X` and `fval`.

```
options = optimoptions(options,...
    'StepTolerance',1e-15,...
    'OptimalityTolerance',1e-8);
```

Call the solver:

```
[x,fval,exitflag,output] = fmincon(@onehump,x0,[],[],[],[],[],[], ...  
    @tiltellipse,options);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of 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.

We now choose to see more decimals in the solution, in order to see more accurately the difference that the new tolerances make.

```
format long
```

The optimizer found a solution at:

```
x
```

```
x = 2×1
```

```
-0.972742227363546  
0.468569289098342
```

Compare this to the previous value:

```
xold
```

```
xold = 2×1
```

```
-0.972742694488360  
0.468569966693330
```

The change is

```
x - xold
```

```
ans = 2×1  
10-6 ×
```

```
0.467124813385844  
-0.677594988729435
```

The function value at the solution is:

```
fval
```

```
fval =  
-0.244893137879894
```

The solution improved by

```
fval - minfval
```

```
ans =  
-3.996450220755676e-07
```

(this is negative since the new solution is smaller)

The total number of function evaluations was:

```
output.funcCount
```

```
ans =  
15
```

Compare this to the number of function evaluations when the problem is solved with user-provided gradients but with the default tolerances:

```
Fgradevals
```

```
Fgradevals =  
14
```

### Constrained Optimization Example with User-Supplied Hessian

If you give not only a gradient, but also a Hessian, solvers are even more accurate and efficient.

`fmincon`'s interior-point solver takes a Hessian matrix as a separate function (not part of the objective function). The Hessian function  $H(x,\lambda)$  should evaluate the Hessian of the Lagrangian; see the User's Guide for the definition of this term.

Solvers calculate the values `lambda.ineqnonlin` and `lambda.eqlin`; your Hessian function tells solvers how to use these values.

In this problem we have but one inequality constraint, so the Hessian is:

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 have been set back to the defaults. There should be fewer function counts this time.

```
options.Display = 'iter';
```

Call the solver:



```
[x,fval,exitflag,output] = fmincon(@onehump,x0,[],[],[],[],[],[], ...
    @tiltellipse,options);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of 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.

There were fewer, and different iterations this time.

The solution to this problem has been found at:

x

x = 2×1

```
-0.972742246093537
 0.468569316215571
```

The function value at the solution is:

fval

```
fval =
-0.244893121872758
```

The total number of function evaluations was:

```
output.funcCount
```

```
ans =  
    11
```

Compare this to the number using only gradient evaluations, with the same default tolerances:

```
Fgradevals
```

```
Fgradevals =  
    14
```

## See Also

### More About

- “Passing Extra Parameters” on page 2-70
- “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  $x_0 = [-1.9, 2]$ .

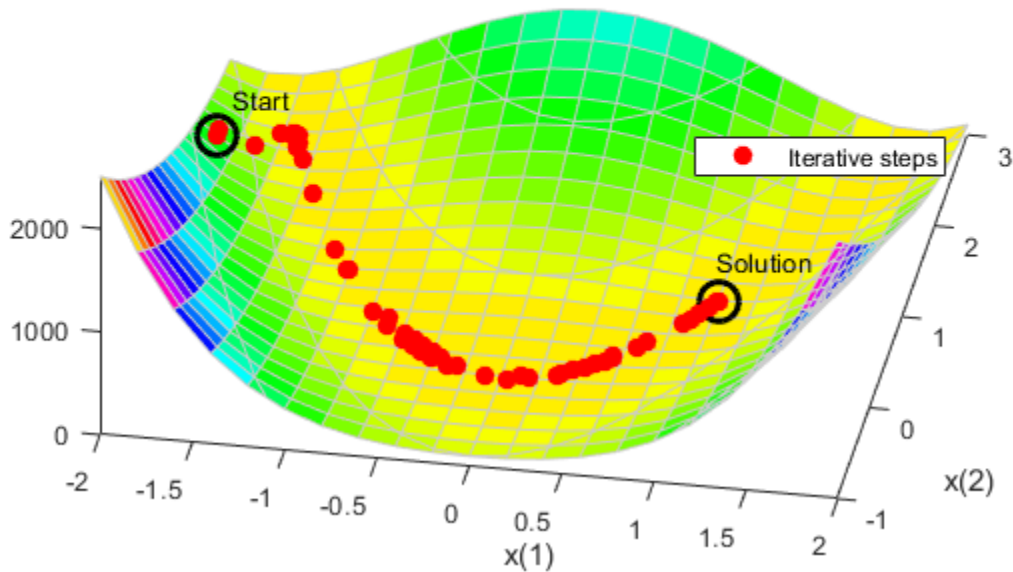
### Optimization Without Derivatives

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 6-11.

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
```

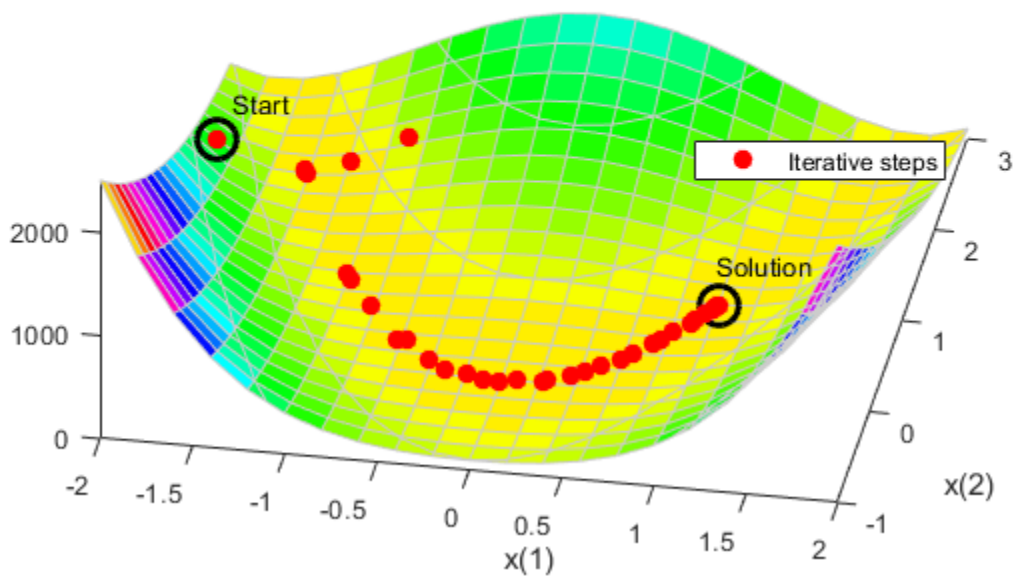
### Optimization with Estimated Derivatives

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

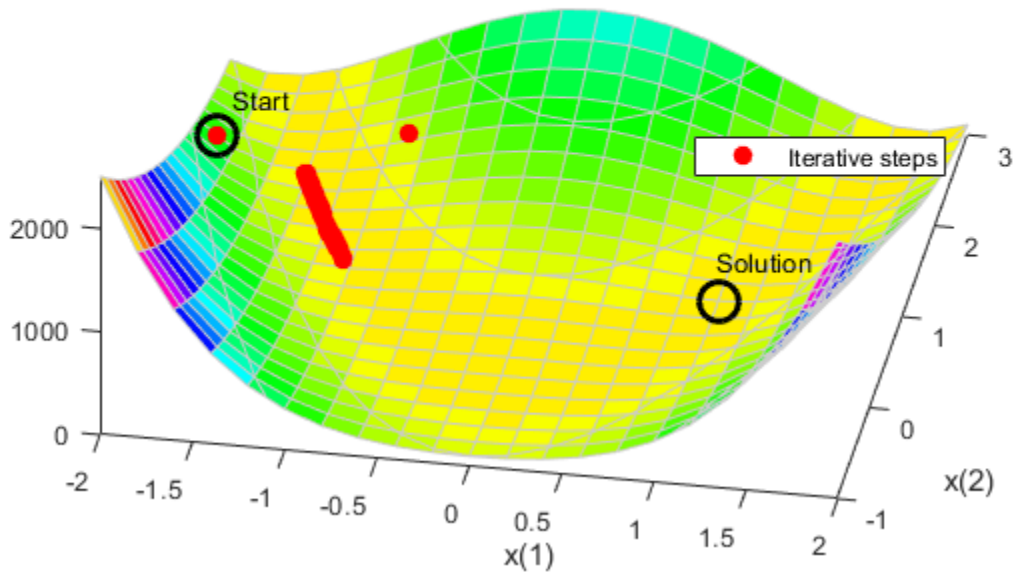
### Optimization with Steepest Descent

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

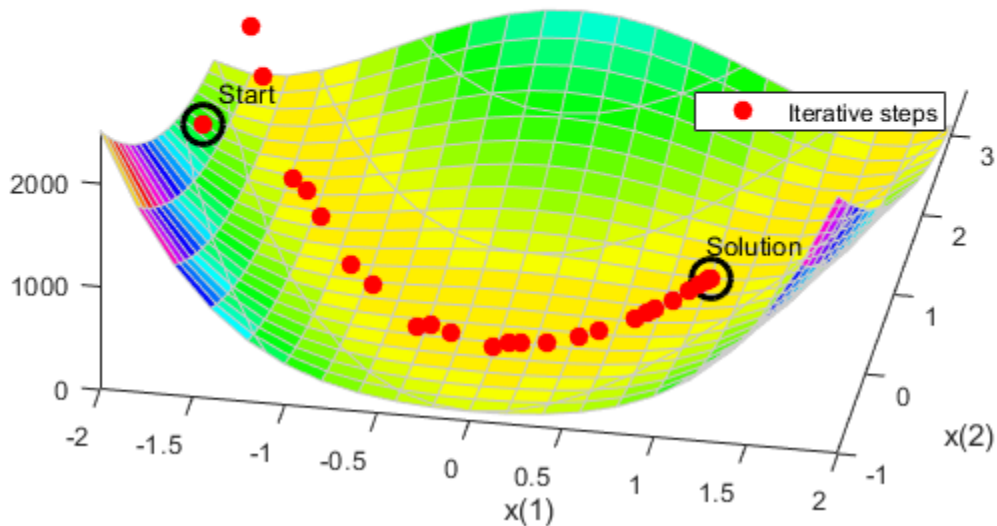
### Optimization with Analytic Gradient

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)])
```

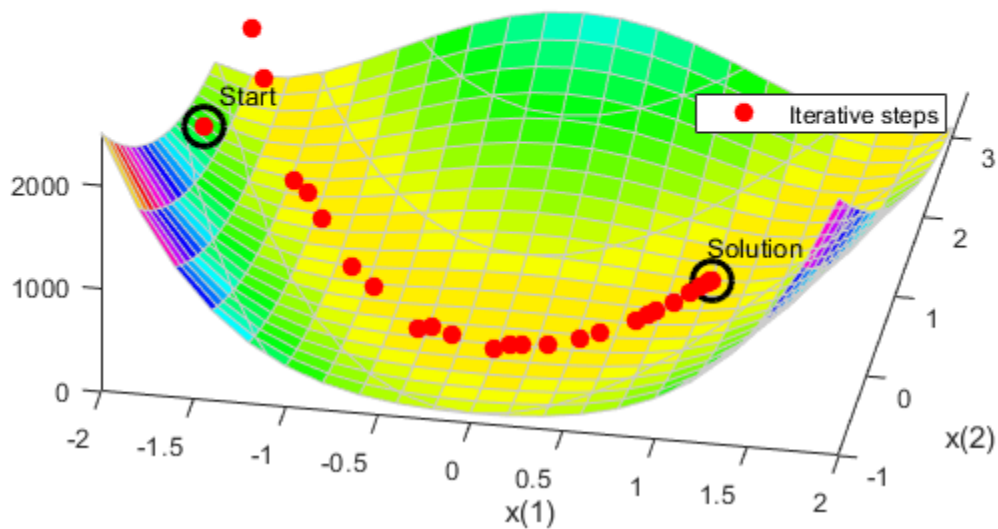
Number of solver iterations for fminunc with gradient was 31

### 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...
```

Number of solver iterations for fminunc with gradient and Hessian was 31

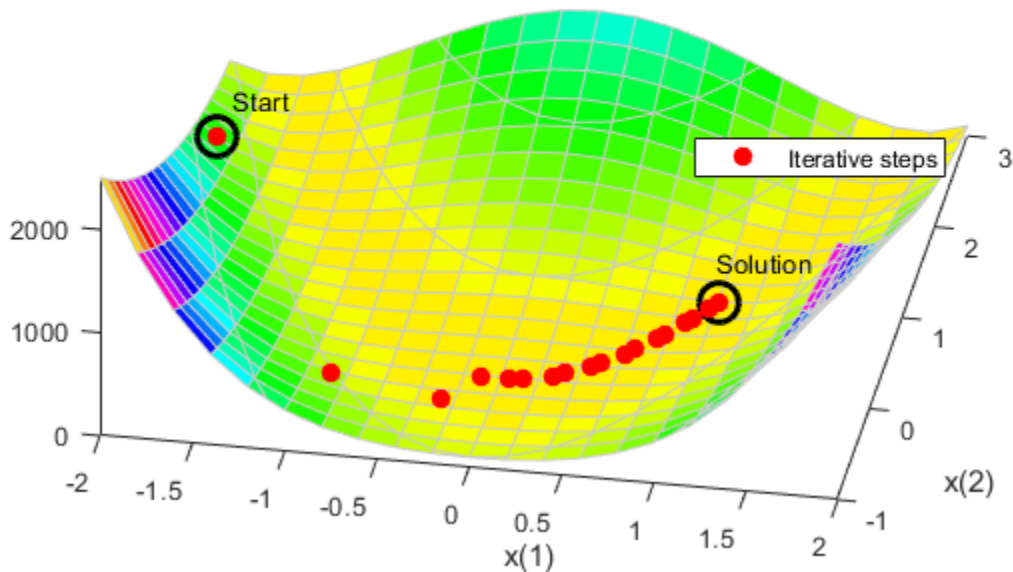
### Optimization with a Least Squares Solver

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)])
```

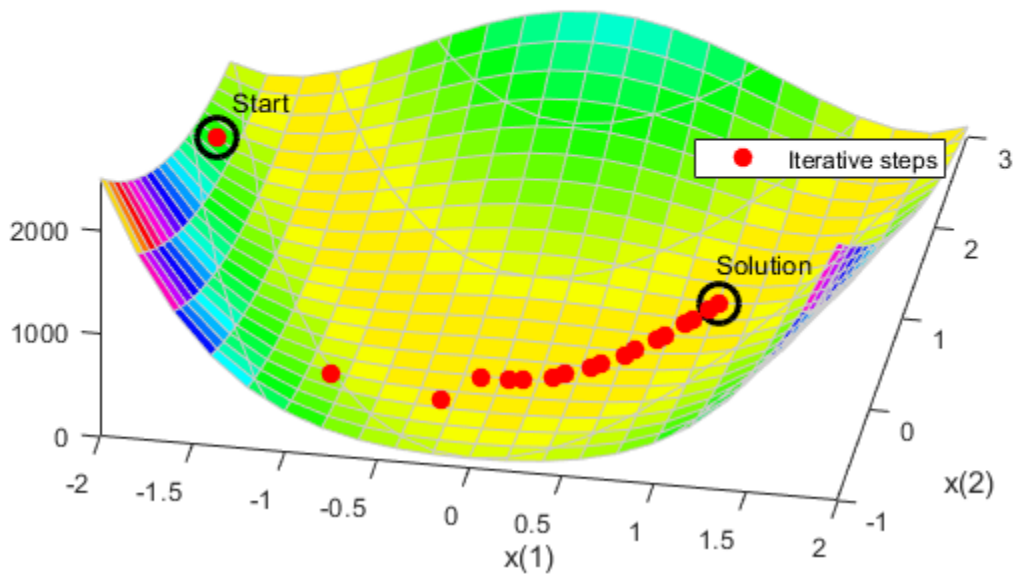
Number of solver iterations for `lsqnonlin` was 28

### 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 1-13
- “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115

## 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);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of 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
```

Generation	Func-count	Best f(x)	Mean f(x)	Stall 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(gcf)) == 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);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of 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.

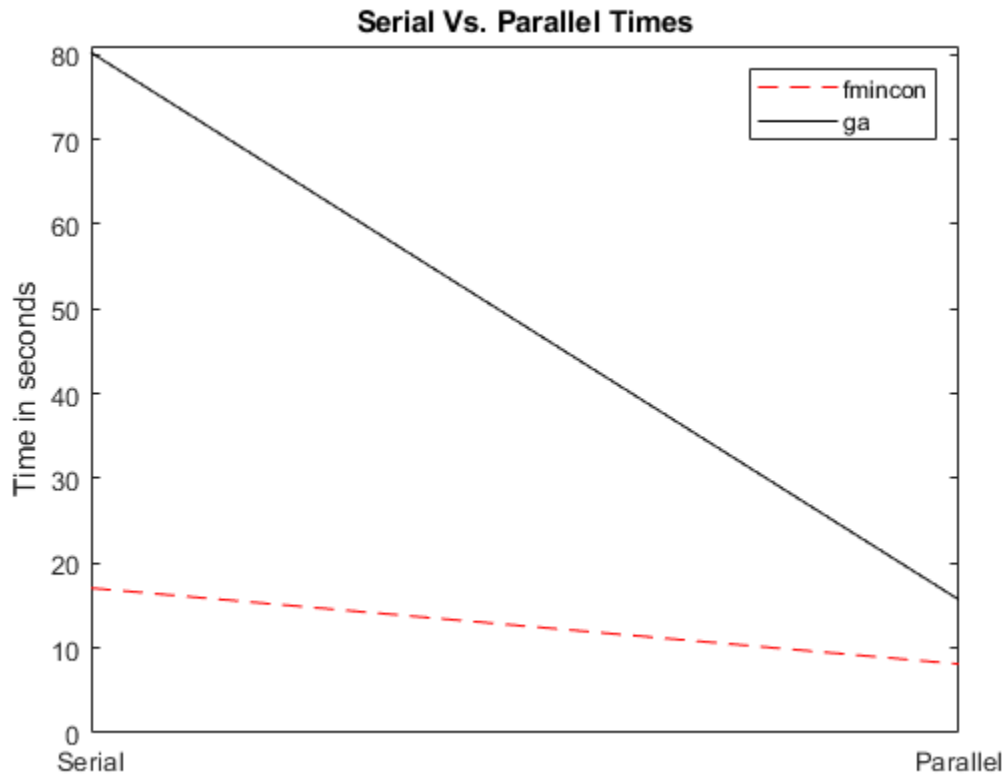
```
rng default % to get the same evaluations as the previous run
if gaAvailable
    gaoptions = optimoptions(gaoptions,'UseParallel',true);
    startTime = tic;
    gasol = ga(@expensive_objfun,nvar,[],[],[],[],[],[],[],gaoptions);
    time_ga_parallel = toc(startTime);
    fprintf('Parallel GA optimization takes %g seconds.\n',time_ga_parallel);
end
```

Generation	Func-count	Best f(x)	Mean f(x)	Stall 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

$$\min_x f(x) = e^{x_1} (4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1). \quad (6-56)$$

subject to the constraints

$$\begin{array}{rclclclcl} x_1x_2 & - & x_1 & - & x_2 & \leq & -1.5, \\ x_1x_2 & & & \geq & & & -10. \end{array}$$

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

$$\begin{array}{rclclclcl} x_1x_2 & - & x_1 & - & x_2 & + & 1.5 & \leq & 0, \\ -x_1x_2 - 10 & \leq & 0. \end{array} \quad (6-57)$$

### Step 1: Write a file `objfun.m` for the objective function.

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

### Step 2: Write a file `confun.m` for the constraints.

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

### Step 3: Invoke constrained optimization routine.

```
x0 = [-1,1]; % Make a starting guess at the solution
options = optimoptions(@fmincon,'Algorithm','sqp');
```

```
[x,fval] = ...  
fmincon(@objfun,x0,[],[],[],[],[],[],@confun,options);
```

fmincon produces the solution  $x$  with function value  $fval$ :

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

You can evaluate the constraints at the solution by entering

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

This returns numbers close to zero, such as

```
c =  
  
   1.0e-14 *  
  
    0.5107  
   -0.5329
```

```
ceq =
```

```
 []
```

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

## See Also

### Related Examples

- “Nonlinear Equality and Inequality Constraints” on page 6-95
- “Nonlinear Constraints with Gradients” on page 6-81

## Nonlinear Constraints with Gradients

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

Consider how to solve

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

subject to the constraints

$$\begin{array}{rclclcl} x_1x_2 & - & x_1 & - & x_2 & \leq & -1.5, \\ x_1x_2 & & & & & \geq & -10. \end{array}$$

To solve the problem using analytically determined gradients, do the following.

### Step 1: Write a file for the objective function and gradient.

```
function [f,gradf] = objfungrad(x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
% Gradient of the objective function:
if nargin > 1
    gradf = [ f + exp(x(1)) * (8*x(1) + 4*x(2)),
             exp(x(1))*(4*x(1)+4*x(2)+2)];
end
```

### Step 2: Write a file for the nonlinear constraints and the gradients of the nonlinear constraints.

```
function [c,ceq,DC,DCEq] = confungrad(x)
c(1) = 1.5 + x(1) * x(2) - x(1) - x(2); % Inequality constraints
c(2) = -x(1) * x(2)-10;
% No nonlinear equality constraints
ceq=[];
% Gradient of the constraints:
if nargin > 2
    DC= [x(2)-1, -x(2);
```

```

        x(1)-1, -x(1)];
    DCeq = [];
end

```

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

$$\nabla f = \begin{bmatrix} e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) + e^{x_1}(8x_1 + 4x_2) \\ e^{x_1}(4x_1 + 4x_2 + 2) \end{bmatrix}. \quad (6-58)$$

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

$$\begin{bmatrix} \frac{\partial c_1}{\partial x_1} & \frac{\partial c_2}{\partial x_1} \\ \frac{\partial c_1}{\partial x_2} & \frac{\partial c_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 - 1 & -x_2 \\ x_1 - 1 & -x_1 \end{bmatrix}. \quad (6-59)$$

Since you are providing the gradient of the objective in `objfungrad.m` and the gradient of the constraints in `confungrad.m`, you *must* tell `fmincon` that these files contain this additional information. Use `optimoptions` to turn the options `SpecifyObjectiveGradient` and `SpecifyConstraintGradient` to `true` in the example's existing options:

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

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

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

### Step 3: Invoke the constrained optimization routine.

```

x0 = [-1,1];           % Starting guess
options = optimoptions(@fmincon,'Algorithm','sqp');
options = optimoptions(options,'SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true);
lb = []; ub = [];     % No upper or lower bounds
[x,fval] = fmincon(@objfungrad,x0,[],[],[],[],[],lb,ub,...
    @confungrad,options);

```



The results:

```
x, fval
```

```
x =  
   -9.5474    1.0474
```

```
fval =  
    0.0236
```

```
[c,ceq] = confungrad(x) % Check the constraint values at x
```

```
c =  
    1.0e-13 *  
   -0.1066  
    0.1066
```

```
ceq =  
    []
```

## See Also

### Related Examples

- “fmincon Interior-Point Algorithm with Analytic Hessian” on page 6-84
- “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115

## fmincon Interior-Point Algorithm with Analytic Hessian

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

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

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

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

```
if nargout > 2

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

end
```

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

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

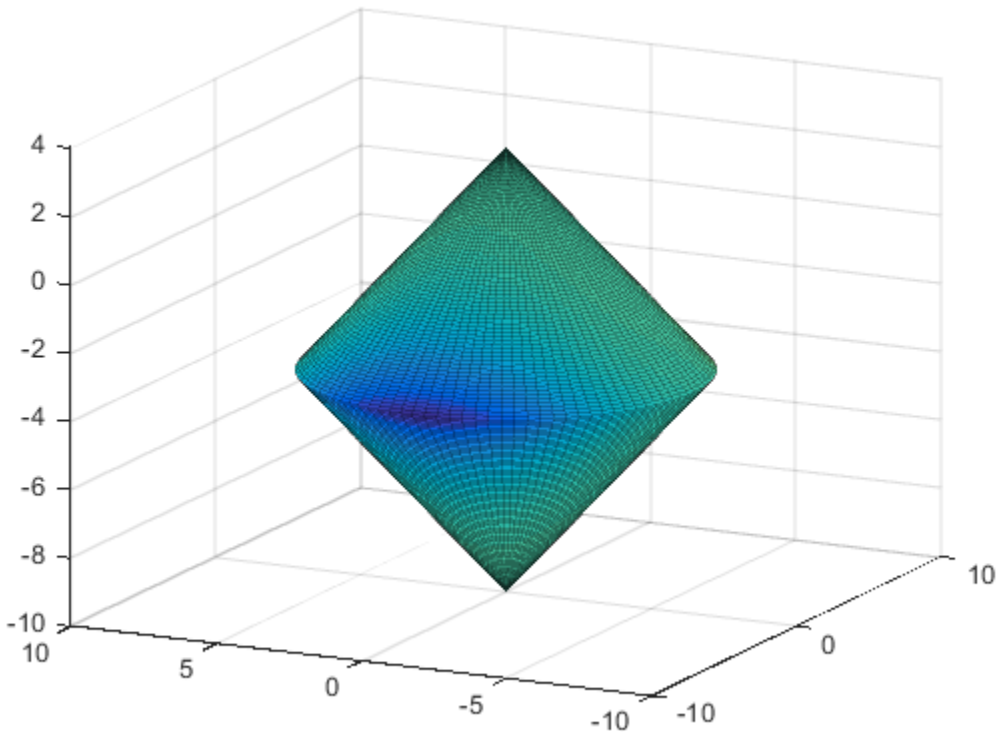
```
if nargout > 1

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

end
```

end

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



The Hessian of the Lagrangian is given by the equation:

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

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

```

function h = hessinterior(x,lambda)

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

```

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

- 1 Set the problem as in the following figure.

The screenshot shows the Optimization app interface with the following settings:

- Solver:** `fmincon - Constrained nonlinear minimization`
- Algorithm:** `Interior point`
- Problem:**
  - Objective function:** `@bigtoleft`
  - Derivatives:** `Gradient supplied`
  - Start point:** `[-1,-1,-1]`
- Constraints:**
  - Linear inequalities:** A:  b:
  - Linear equalities:** Aeq:  beq:
  - Bounds:** Lower:  Upper:
  - Nonlinear constraint function:** `@twocone`
  - Derivatives:** `Gradient supplied`

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

Display to command window

Level of display:

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

Hessian

Hessian:

Hessian function

Hessian multiply function

Function:

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

Run solver and view results

Current iteration:

-----

Optimization running.  
 Objective function value: -2894.1247991783007  
 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.

▲▼

Final point:

1 ▲	2	3
-6.5	-0	-3.5

To perform the minimization at the command line:

- 1 Set options as follows:

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

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

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

Examine the solution, objective function value, exit flag, and number of function evaluations and iterations:

```
x,fval,mflag,output.funcCount,output.iterations
```

```
x =
```

```
    -6.5000    -0.0000    -3.5000
```

```
fval =
```

```
    -2.8941e+03
```

```
mflag =
```

```
     1
```

```
ans =
```

```
     7
```

```
ans =
```

```
     6
```

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

```
options = optimoptions(@fmincon,'Algorithm','interior-point',...  
    'Display','off','SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true,
```

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

```
x,output.funcCount,output.iterations
```

```
x =
```

```
    -6.5000    -0.0000    -3.5000
```

```
ans =
```

```
    13
```

```
ans =
```

```
     9
```

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

## See Also

### Related Examples

- “Linear or Quadratic Objective with Quadratic Constraints” on page 6-90
- “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115

## Linear or Quadratic Objective with Quadratic Constraints

This example shows how to solve an optimization problem that has a linear or quadratic objective and quadratic inequality constraints. It shows how to generate and use the gradient and Hessian of the objective and constraint functions.

### Quadratic Constrained Problem

Suppose that you can put your problem in the form

$$\min_x \left( \frac{1}{2} x^T Q x + f^T x + c \right)$$

subject to

$$\frac{1}{2} x^T H_i x + k_i^T x + d_i \leq 0,$$

where  $1 \leq i \leq m$ . Assume that at least one  $H_i$  is nonzero; otherwise, you can use `quadprog` or `linprog` to solve this problem. With nonzero  $H_i$ , the constraints are nonlinear, and the “Optimization Decision Table” on page 2-6 states that `fmincon` is the appropriate solver.

The example assumes that the quadratic matrices are symmetric. This is 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 the  $H_i$  are  $N$ -by- $N$  matrices,  $f$  and the  $k_i$  are  $N$ -by-1 vectors, and  $c$  and the  $d_i$  are scalars.

### Objective Function

Formulate the problem using `fmincon` syntax. Assume that  $x$  and  $f$  are column vectors. ( $x$  is a column vector when the initial vector  $x_0$  is.)

```
function [y,grady] = quadobj(x,Q,f,c)
y = 1/2*x'*Q*x + f'*x + c;
if nargin > 1
    grady = Q*x + f;
end
```



### Constraint Function

For consistency and easy indexing, place every quadratic constraint matrix in one cell array. Similarly, place the linear and constant terms in cell arrays.

```
function [y,yeq,grady,gradyeq] = quadconstr(x,H,k,d)
jj = length(H); % jj is the number of inequality constraints
y = zeros(1,jj);
for i = 1:jj
    y(i) = 1/2*x'*H{i}*x + k{i}'*x + d{i};
end
yeq = [];

if nargout > 2
    grady = zeros(length(x),jj);
    for i = 1:jj
        grady(:,i) = H{i}*x + k{i};
    end
end
gradyeq = [];
```

### Numeric Example

For example, suppose that you have the following problem.

```
Q = [3,2,1;
     2,4,0;
     1,0,5];
f = [-24;-48;-130];
c = -2;

rng default % for reproducibility
% Two sets of random quadratic constraints:
H{1} = gallery('randcorr',3); % random positive definite matrix
H{2} = gallery('randcorr',3);
k{1} = randn(3,1);
k{2} = randn(3,1);
d{1} = randn;
d{2} = randn;
```

### Hessian

Create a Hessian function. The Hessian of the Lagrangian is given by the equation

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

fmincon calculates an approximate set of Lagrange multipliers  $\lambda_i$ , 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. Examine the number of function evaluations.

```
output
```

```

output =

        iterations: 9
         funcCount: 10
    constrviolation: 0
         stepsize: 5.3547e-04
         algorithm: 'interior-point'
    firstorderopt: 1.5851e-05
         cgiterations: 0
         message: 'Local minimum found that satisfies the constraints.

Optimization compl...'
    
```

`fmincon` used just 10 function evaluations to solve the problem.

Compare this to the solution without the Hessian.

```

options.HessianFcn = [];
[x2,fval2,eflag2,output2,lambda2] = fmincon(fun,[0;0;0],...
    [],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]);
output2

output2 =
    
```

```

        iterations: 17
         funcCount: 22
    constrviolation: 0
         stepsize: 2.8475e-04
         algorithm: 'interior-point'
    firstorderopt: 1.7680e-05
         cgiterations: 0
         message: 'Local minimum found that satisfies the constraints.

Optimization compl...'
    
```

This time `fmincon` used about twice as many iterations and function evaluations. The solutions are the same to within tolerances.

### Extension to Quadratic Equality Constraints

If you also have quadratic equality constraints, you can use essentially the same technique. The problem is the same, with the additional constraints

$$\frac{1}{2}x^T J_i x + p_i^T x + q_i = 0.$$

Reformulate your constraints to use the  $J_i$ ,  $p_i$ , and  $q_i$  variables. The `lambda.eqnonlin(i)` structure has the Lagrange multipliers for equality constraints.

## See Also

### Related Examples

- “fmincon Interior-Point Algorithm with Analytic Hessian” on page 6-84

### More About

- “Including Gradients and Hessians” on page 2-25
- “Including Gradients in Constraint Functions” on page 2-49

## Nonlinear Equality and Inequality Constraints

You can include nonlinear constraints by writing a function that computes both equality and inequality constraint values. A nonlinear constraint function has the syntax

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

The function  $c(x)$  represents the constraint  $c(x) \leq 0$ . The function  $ceq(x)$  represents the constraint  $ceq(x) = 0$ .

---

**Note** You must have the nonlinear constraint function return both  $c(x)$  and  $ceq(x)$ , even if you have only one type of nonlinear constraint. If a constraint does not exist, have the function return `[]` for that constraint.

---

For example, if you have the nonlinear equality constraint  $x_1^2 + x_2 = 1$  and the nonlinear inequality constraint  $x_1x_2 \geq -10$ , rewrite them as

$$\begin{aligned}x_1^2 + x_2 - 1 &= 0, \\ -x_1x_2 - 10 &\leq 0,\end{aligned}$$

and then solve the problem using the following steps.

For this example, solve the problem

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

subject to these nonlinear constraints.

### Step 1: Write a file objfun.m.

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

### Step 2: Write a file confuneq.m for the nonlinear constraints.

```
function [c,ceq] = confuneq(x)
% Nonlinear inequality constraints
```

```
c = -x(1)*x(2) - 10;  
% Nonlinear equality constraints  
ceq = x(1)^2 + x(2) - 1;
```

### **Step 3: Invoke constrained optimization routine.**

```
x0 = [-1,1]; % Make a starting guess at the solution  
options = optimoptions(@fmincon,'Algorithm','sqp');  
[x,fval] = fmincon(@objfun,x0,[],[],[],[],[],[],...  
    @confuneq,options);
```

After 21 function evaluations, the solution produced is

```
x,fval  
x =  
   -0.7529    0.4332  
fval =  
    1.5093
```

```
[c,ceq] = confuneq(x) % Check the constraint values at x
```

```
c =  
   -9.6739  
  
ceq =  
  -2.2204e-16
```

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

## **See Also**

### **Related Examples**

- “Nonlinear Inequality Constraints” on page 6-79
- “Optimization App with the `fmincon` Solver” on page 6-97

## Optimization App with the fmincon Solver

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

---

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

---

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

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

subject to the constraints

$$\begin{array}{ll} 0.5 \leq x_1 & \text{(bound)} \\ -x_1 - x_2 + 1 \leq 0 & \text{(linear inequality)} \\ \left. \begin{array}{l} -x_1^2 - x_2^2 + 1 \leq 0 \\ -9x_1^2 - x_2^2 + 9 \leq 0 \\ -x_1^2 + x_2 \leq 0 \\ -x_2^2 + x_1 \leq 0 \end{array} \right\} & \text{(nonlinear inequality)} \end{array}$$

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

### Step 1: Write a file objcfun.m for the objective function.

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

### Step 2: Write a file nonlconstr.m for the nonlinear constraints.

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

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

- 1 Enter `optimtool` in the Command Window to open the Optimization app.
- 2 Select `fmincon` from the selection of solvers and change the **Algorithm** field to `Active set`.

Solver: `fmincon - Constrained nonlinear minimization`

Algorithm: `Active set`

- 3 Enter `@objcfun` in the **Objective function** field to call the `objcfun.m` file.
- 4 Enter `[3;1]` in the **Start point** field.

Objective function: `@objcfun`

Derivatives: `Approximated by solver`

Start point: `[3;1]`

- 5 Define the constraints.
  - Set the bound  $0.5 \leq x_1$  by entering `[0.5, -Inf]` in the **Lower** field. The `-Inf` entry means there is no lower bound on  $x_2$ .
  - Set the linear inequality constraint by entering `[-1 -1]` in the **A** field and enter `-1` in the **b** field.
  - Set the nonlinear constraints by entering `@nonlconstr` in the **Nonlinear constraint function** field.

Constraints:

Linear inequalities: A: `[-1 -1]` b: `-1`

Linear equalities: Aeq:  beq:

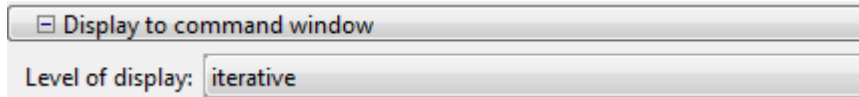
Bounds: Lower: `[0.5,-Inf]` Upper:

Nonlinear constraint function: `@nonlconstr`

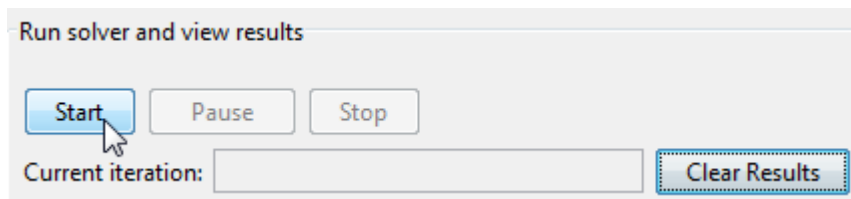
Derivatives: `Approximated by solver`



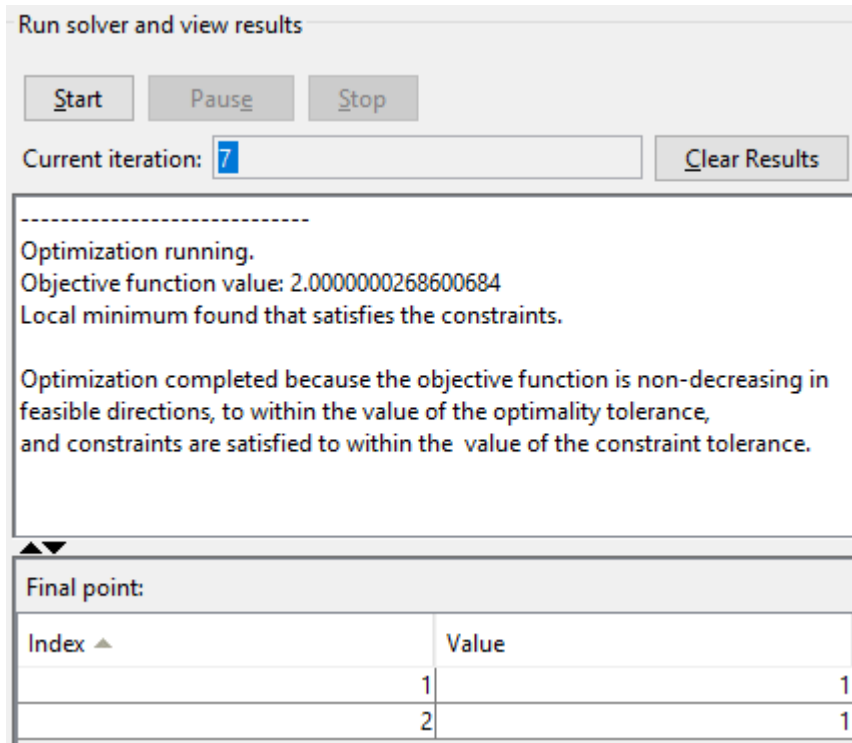
- 6 In the **Options** pane, expand the **Display to command window** option if necessary, and select **Iterative** to show algorithm information at the Command Window for each iteration.



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



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



- The **Current iteration** value when the algorithm terminated, which for this example is 7.
- The final value of the objective function when the algorithm terminated:  
 Objective function value: 2.0000000268595803
- The algorithm termination message:  
 Local minimum found that satisfies the constraints.  
 Optimization completed because the objective function is non-decreasing in feasible directions, to within the value of the optimality tolerance, and constraints are satisfied to within the value of the constraint tolerance.
- The final point, which for this example is  
 1  
 1

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

```

      Max      Line search  Directional  First-order
Iter F-count   f(x)  constraint  steplength  derivative  optimality  Procedure
  0     3         10          2              1         -5.22         1.74      Infeasible start point
  1     6     4.84298     -0.1322              1         -4.39         4.08      Hessian modified twice
  2     9     4.0251     -0.01168              1         -3.85         1.09
  3    12     2.42704     -0.03214              1         -3.04         0.995     Hessian modified twice
  4    15     2.03615     -0.004728              1         -2.82         0.0664    Hessian modified twice
  5    18     2.00033     -5.596e-05              1         -2.81         0.000522  Hessian modified twice
  6    21         2     -5.326e-09              1
Active inequalities (to within options.ConstraintTolerance = 1e-06):
  lower      upper      ineqlin  ineqnonlin
           3
           4

```

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.

## See Also

### Related Examples

- “Solve a Constrained Nonlinear Problem, Solver-Based” on page 1-13

## Minimization with Bound Constraints and Banded Preconditioner

The goal in this problem is to minimize the nonlinear function

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

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

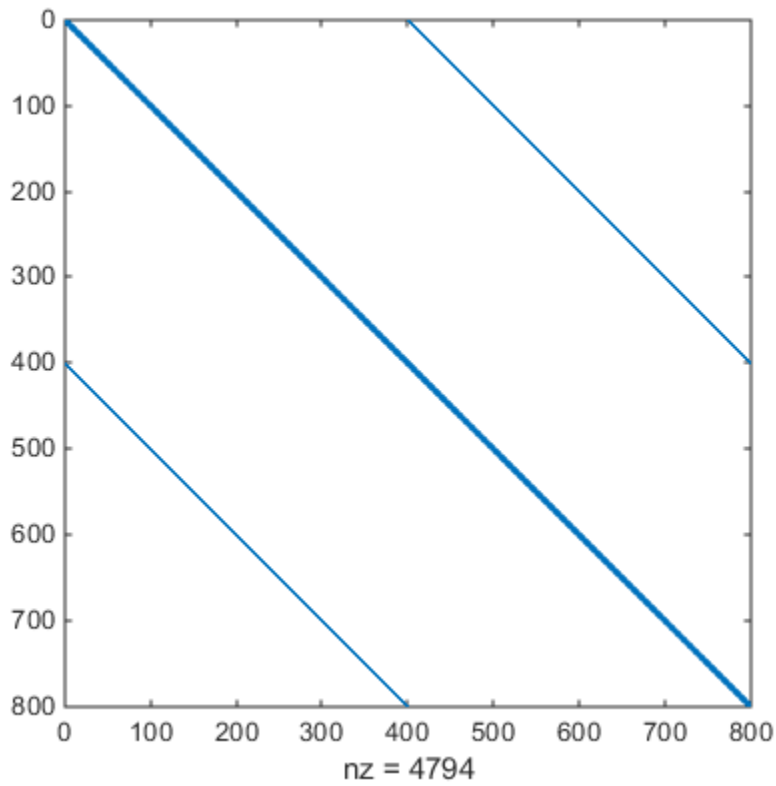
### Step 1: Write a file `tbroyfg.m` that computes the objective function and the gradient of the objective

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

```
type tbroyfg
```

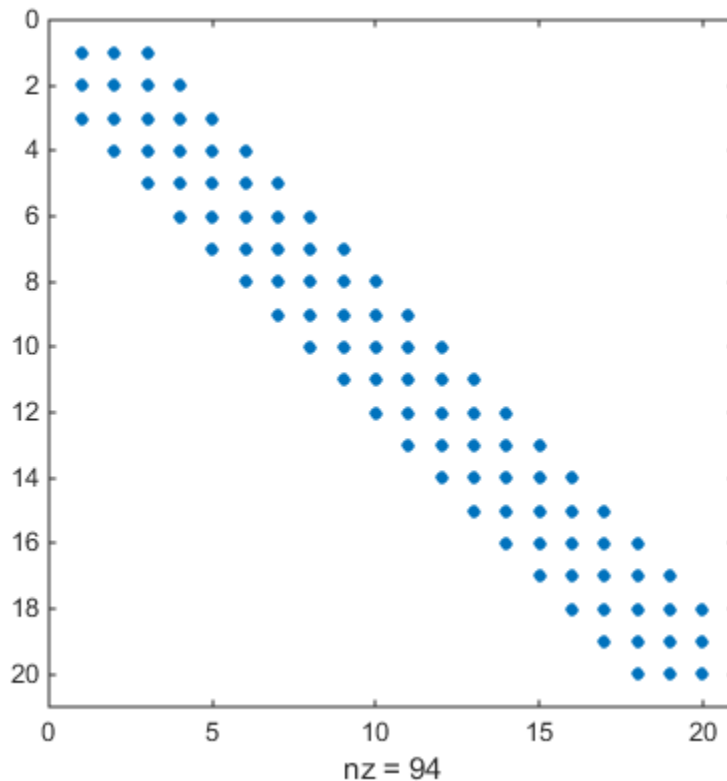
The sparsity pattern of the Hessian matrix has been predetermined and stored in the file `tbroyhstr.mat`. The sparsity structure for the Hessian of this problem is banded, as you can see in the following `spy` plot.

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



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

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



Use `optimoptions` to set the `HessPattern` parameter to `Hstr`. When a problem as large as this has obvious sparsity structure, not setting the `HessPattern` parameter requires a huge amount of unnecessary memory and computation. This is because `fmincon` attempts to use finite differencing on a full Hessian matrix of 640,000 nonzero entries.

You must also set the `SpecifyObjectiveGradient` parameter to `true` using `optimoptions`, since the gradient is computed in `tbroyfg.m`. Then execute `fmincon` as shown in Step 2 on page 6-105.

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

```

fun = @tbroyfg;
load tbroyhstr          % Get Hstr, structure of the Hessian
n = 800;
xstart = -ones(n,1); xstart(2:2:n) = 1;
lb = -10*ones(n,1); ub = -lb;
options = optimoptions('fmincon','SpecifyObjectiveGradient',true,'HessPattern',Hstr,...
    'Algorithm','trust-region-reflective');

[x,fval,exitflag,output] = ...
    fmincon(fun,xstart,[],[],[],[],lb,ub,[],options);

```

The `exitflag`, `fval`, first-order optimality measure (`output.firstorderopt`), and number of iterations (`output.iterations`) are:

```

exitflag,fval,output.firstorderopt,output.iterations
exitflag =

```

```

    3

```

```

fval =

```

```

    270.4790

```

```

ans =

```

```

    0.0163

```

```

ans =

```

```

    7

```

For bound constrained problems, the first-order optimality is the infinity norm of  $v \cdot g$ , where  $v$  is defined as in “Box Constraints” on page 6-25, and  $g$  is the gradient.

Because of the five-banded center stripe, you can improve the solution by using a five-banded preconditioner instead of the default diagonal preconditioner. Using the `optimoptions` function, reset the `PrecondBandWidth` parameter to 2 and solve the

problem again. (The bandwidth is the number of upper (or lower) diagonals, not counting the main diagonal.)

```
fun = @tbroyfg;
load tbroyhstr          % Get Hstr, structure of the Hessian
n = 800;
xstart = -ones(n,1); xstart(2:2:n,1) = 1;
lb = -10*ones(n,1); ub = -lb;
options = optimoptions('fmincon','SpecifyObjectiveGradient',true,'HessPattern',Hstr, .
    'Algorithm','trust-region-reflective','PrecondBandWidth',2);

[x,fval,exitflag,output] = ...
    fmincon(fun,xstart,[],[],[],[],lb,ub,[],options);
```

The number of iterations increases by two. But the first-order optimality measure is reduced by a factor of  $1e-3$ :

```
exitflag,fval,output.firstorderopt,output.iterations
exitflag =
```

```
3
```

```
fval =
```

```
270.4790
```

```
ans =
```

```
7.5340e-05
```

```
ans =
```

```
9
```



## Minimization with Linear Equality Constraints

The trust-region reflective method for `fmincon` can handle linear equality constraints if no other constraints exist. Suppose you want to minimize

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

subject to some linear equality constraints. The objective function is coded in the function `brownfgh.m`. This example takes  $n = 1000$ . Furthermore, the `browneq.mat` file contains matrices `Aeq` and `beq` that represent the linear constraints  $Aeq \cdot x = beq$ . `Aeq` has 100 rows representing 100 linear constraints (so `Aeq` is a 100-by-1000 matrix).

### Step 1: Write a file `brownfgh.m` that computes the objective function, the gradient of the objective, and the sparse tridiagonal Hessian matrix.

The file is lengthy so is not included here. View the code with the command

```
type brownfgh
```

Because `brownfgh` computes the gradient and Hessian values as well as the objective function, you need to use `optimoptions` to indicate that this information is available in `brownfgh`, using the `SpecifyObjectiveGradient` and `Hessian` options.

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

```
load browneq
```

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

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

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

```
fun = @brownfgh;
load browneq           % Get Aeq and beq, the linear equalities
```

```
n = 1000;
xstart = -ones(n,1); xstart(2:2:n) = 1;
options = optimoptions('fmincon','SpecifyObjectiveGradient',true,'HessianFcn','objectiv
    'Algorithm','trust-region-reflective');
[x,fval,exitflag,output] = ...
    fmincon(fun,xstart,[],[],Aeq,beq,[],[],[],options);
```

fmincon prints the following exit message:

```
Local minimum possible.
```

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

The `exitflag` value of 3 also indicates that the algorithm terminated because the change in the objective function value was less than the tolerance `FunctionTolerance`. The final function value is given by `fval`. Constraints are satisfied, as you see in `output.constrviolation`

```
exitflag,fval,output.constrviolation
```

```
exitflag =
```

```
3
```

```
fval =
```

```
205.9313
```

```
ans =
```

```
2.2071e-13
```

The linear equalities are satisfied at `x`.

```
norm(Aeq*x-beq)
```

```
ans =
```

```
1.1858e-12
```

## **See Also**

### **More About**

- “Problem-Based Nonlinear Minimization with Linear Constraints” on page 7-20

## Minimization with Dense Structured Hessian, Linear Equalities

### In this section...

“Hessian Multiply Function for Lower Memory” on page 6-110

“Step 1: Write a file brownvv.m that computes the objective function, the gradient, and the sparse part of the Hessian.” on page 6-111

“Step 2: Write a function to compute Hessian-matrix products for  $H$  given a matrix  $Y$ .” on page 6-111

“Step 3: Call a nonlinear minimization routine with a starting point and linear equality constraints.” on page 6-112

“Preconditioning” on page 6-114

### 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

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

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

$$H = \hat{H} - V V^T.$$

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

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

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

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

### **Step 1: Write a file `brownvv.m` that computes the objective function, the gradient, and the sparse part of the Hessian.**

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

```
type brownvv
```

Because `brownvv` computes the gradient as well as the objective function, the example (Step 3 on page 6-112) 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

$$W = \text{hmfleq1}(\text{Hinfo}, Y)$$

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);
%HMFLQ1 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,  $A_{eq}$  and  $b_{eq}$ , 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:

Iteration	f(x)	Norm of step	First-order 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('fleg1'); % 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.



## Symbolic Math Toolbox Calculates Gradients and Hessians

If you have a Symbolic Math Toolbox license, you can easily calculate analytic gradients and Hessians for objective and constraint functions. There are two relevant Symbolic Math Toolbox functions:

- `jacobian` generates the gradient of a scalar function, and generates a matrix of the partial derivatives of a vector function. So, for example, you can obtain the Hessian matrix, the second derivatives of the objective function, by applying `jacobian` to the gradient. Part of this example shows how to use `jacobian` to generate symbolic gradients and Hessians of objective and constraint functions.
- `matlabFunction` generates either an anonymous function or a file that calculates the values of a symbolic expression. This example shows how to use `matlabFunction` to generate files that evaluate the objective and constraint function and their derivatives at arbitrary points.

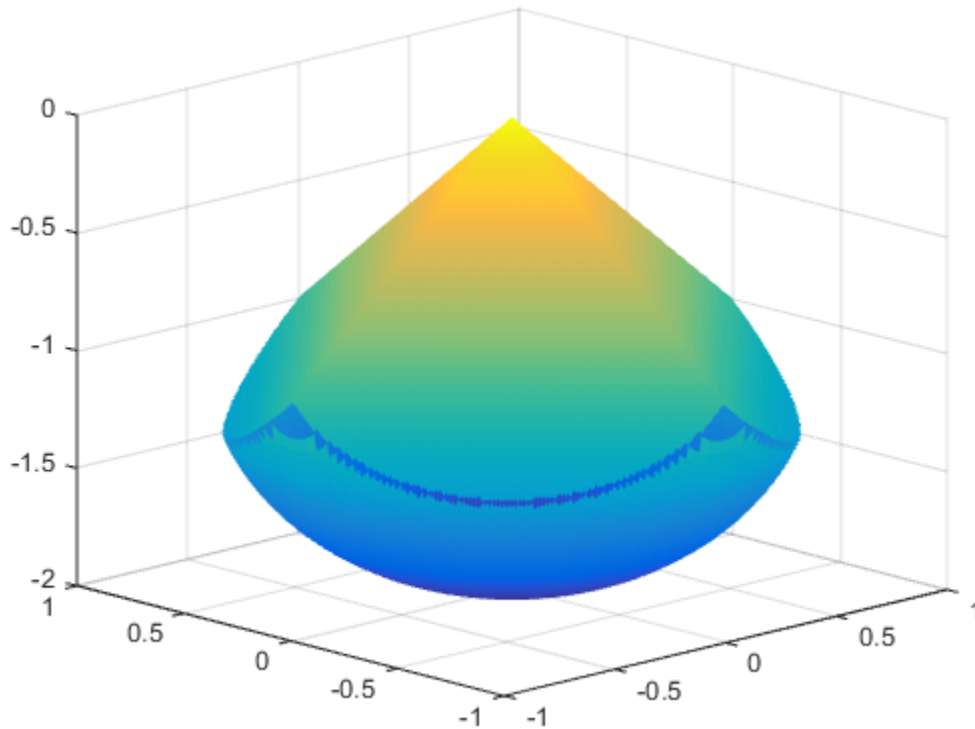
Consider the electrostatics problem of placing 10 electrons in a conducting body. The electrons will arrange themselves so as to minimize their total potential energy, subject to the constraint of lying inside the body. It is well known that all the electrons will be on the boundary of the body at a minimum. The electrons are indistinguishable, so there is no unique minimum for this problem (permuting the electrons in one solution gives another valid solution). This example was inspired by Dolan, Moré, and Munson [58].

This example is a conducting body defined by the following inequalities:

$$z \leq -|x| - |y| \tag{6-60}$$

$$x^2 + y^2 + (z + 1)^2 \leq 1. \tag{6-61}$$

This body looks like a pyramid on a sphere.



There is a slight gap between the upper and lower surfaces of the figure. This is an artifact of the general plotting routine used to create the figure. This routine erases any rectangular patch on one surface that touches the other surface.

The syntax and structures of the two sets of toolbox functions differ. In particular, symbolic variables are real or complex scalars, but Optimization Toolbox functions pass vector arguments. So there are several steps to take to generate symbolically the objective function, constraints, and all their requisite derivatives, in a form suitable for the interior-point algorithm of `fmincon`:

- 1 “Create the Variables” on page 6-117
- 2 “Include the Linear Constraints” on page 6-118

- 3 “Create the Nonlinear Constraints, Their Gradients and Hessians” on page 6-120
- 4 “Create the Objective Function, Its Gradient and Hessian” on page 6-121
- 5 “Create the Objective Function File” on page 6-121
- 6 “Create the Constraint Function File” on page 6-122
- 7 “Generate the Hessian Files” on page 6-123
- 8 “Run the Optimization” on page 6-123
- 9 “Clear the Symbolic Variable Assumptions” on page 6-128

To see the efficiency in using gradients and Hessians, see “Compare to Optimization Without Gradients and Hessians” on page 6-126. For a problem-based approach to this problem without using derivative information, see “Constrained Electrostatic Nonlinear Optimization, Problem-Based” on page 7-13.

## Create the Variables

Generate a symbolic vector  $x$  as a 30-by-1 vector composed of real symbolic variables  $x_{ij}$ ,  $i$  between 1 and 10, and  $j$  between 1 and 3. These variables represent the three coordinates of electron  $i$ :  $x_{i1}$  corresponds to the  $x$  coordinate,  $x_{i2}$  corresponds to the  $y$  coordinate, and  $x_{i3}$  corresponds to the  $z$  coordinate.

```
x = cell(3, 10);
for i = 1:10
    for j = 1:3
        x{j,i} = sprintf('x%d%d',i,j);
    end
end
x = x(:); % now x is a 30-by-1 vector
x = sym(x, 'real');
```

The vector  $x$  is:

```
x
x =
    x11
    x12
    x13
    x21
    x22
    x23
    x31
```

x32  
 x33  
 x41  
 x42  
 x43  
 x51  
 x52  
 x53  
 x61  
 x62  
 x63  
 x71  
 x72  
 x73  
 x81  
 x82  
 x83  
 x91  
 x92  
 x93  
 x101  
 x102  
 x103

### Include the Linear Constraints

Write the linear constraints in “Equation 6-60”,

$$z \leq -|x| - |y|,$$

as a set of four linear inequalities for each electron:

xi3	-	xi1	-	xi2	≤	0
xi3	-	xi1	+	xi2	≤	0
xi3	+	xi1	-	xi2	≤	0
xi3	+	xi1	+	xi2	≤	0

Therefore there are a total of 40 linear inequalities for this problem.

Write the inequalities in a structured way:

$$B = [1, 1, 1; -1, 1, 1; 1, -1, 1; -1, -1, 1];$$

```

A = zeros(40,30);
for i=1:10
    A(4*i-3:4*i,3*i-2:3*i) = B;
end

b = zeros(40,1);

```

You can see that  $A*x \leq b$  represents the inequalities:

$A*x$

```

ans =
    x11 + x12 + x13
    x12 - x11 + x13
    x11 - x12 + x13
    x13 - x12 - x11
    x21 + x22 + x23
    x22 - x21 + x23
    x21 - x22 + x23
    x23 - x22 - x21
    x31 + x32 + x33
    x32 - x31 + x33
    x31 - x32 + x33
    x33 - x32 - x31
    x41 + x42 + x43
    x42 - x41 + x43
    x41 - x42 + x43
    x43 - x42 - x41
    x51 + x52 + x53
    x52 - x51 + x53
    x51 - x52 + x53
    x53 - x52 - x51
    x61 + x62 + x63
    x62 - x61 + x63
    x61 - x62 + x63
    x63 - x62 - x61
    x71 + x72 + x73
    x72 - x71 + x73
    x71 - x72 + x73
    x73 - x72 - x71
    x81 + x82 + x83
    x82 - x81 + x83
    x81 - x82 + x83
    x83 - x82 - x81
    x91 + x92 + x93

```

```
x92 - x91 + x93
x91 - x92 + x93
x93 - x92 - x91
x101 + x102 + x103
x102 - x101 + x103
x101 - x102 + x103
x103 - x102 - x101
```

## Create the Nonlinear Constraints, Their Gradients and Hessians

The nonlinear constraints in “Equation 6-61” ,

$$x^2 + y^2 + (z + 1)^2 \leq 1,$$

are also structured. Generate the constraints, their gradients, and Hessians as follows:

```
c = sym(zeros(1,10));
i = 1:10;
c = (x(3*i-2).^2 + x(3*i-1).^2 + (x(3*i)+1).^2 - 1).';

gradc = jacobian(c,x).'; % .' performs transpose

hessc = cell(1, 10);
for i = 1:10
    hessc{i} = jacobian(gradc(:,i),x);
end
```

The constraint vector  $c$  is a row vector, and the gradient of  $c(i)$  is represented in the  $i$ th column of the matrix  $gradc$ . This is the correct form, as described in “Nonlinear Constraints” on page 2-48.

The Hessian matrices,  $hessc\{1\} \dots hessc\{10\}$ , are square and symmetric. It is better to store them in a cell array, as is done here, than in separate variables such as  $hessc1, \dots, hessc10$ .

Use the `.'` syntax to transpose. The `'` syntax means conjugate transpose, which has different symbolic derivatives.

## Create the Objective Function, Its Gradient and Hessian

The objective function, potential energy, is the sum of the inverses of the distances between each electron pair:

$$\text{energy} = \sum_{i < j} \frac{1}{|x_i - x_j|}.$$

The distance is the square root of the sum of the squares of the differences in the components of the vectors.

Calculate the energy, its gradient, and its Hessian as follows:

```
energy = sym(0);
for i = 1:3:25
    for j = i+3:3:28
        dist = x(i:i+2) - x(j:j+2);
        energy = energy + 1/sqrt(dist.'*dist);
    end
end

gradenergy = jacobian(energy,x).';

hessenergy = jacobian(gradenergy,x);
```

## Create the Objective Function File

The objective function should have two outputs, `energy` and `gradenergy`. Put both functions in one vector when calling `matlabFunction` to reduce the number of subexpressions that `matlabFunction` generates, and to return the gradient only when the calling function (`fmincon` in this case) requests both outputs. This example shows placing the resulting files in your current folder. Of course, you can place them anywhere you like, as long as the folder is on the MATLAB path.

```
curdir = [pwd filesep]; % You may need to use curdir = pwd
filename = [curdir,'demoenergy.m'];
matlabFunction(energy,gradenergy,'file',filename,'vars',{x});
```

This syntax causes `matlabFunction` to return `energy` as the first output, and `gradenergy` as the second. It also takes a single input vector `{x}` instead of a list of inputs `x11, ..., x103`.

The resulting file `demoenergy.m` contains, in part, the following lines or similar ones:

```
function [energy,gradenergy] = demoenergy(in1)
%DEMOENERGY
%   [ENERGY,GRADENERGY] = DEMOENERGY(IN1)
...
x101 = in1(28,:);
...
energy = 1./t140.^(1./2) + ...;
if nargout > 1
    ...
    gradenergy = [(t174.*(t185 - 2.*x11))./2 - ...];
end
```

This function has the correct form for an objective function with a gradient; see “Writing Scalar Objective Functions” on page 2-23.

## Create the Constraint Function File

Generate the nonlinear constraint function, and put it in the correct format.

```
filename = [currdir,'democonstr.m'];
matlabFunction(c,[],gradc,[],'file',filename,'vars',{x},...
    'outputs',{'c','ceq','gradc','gradceq'});
```

The resulting file `democonstr.m` contains, in part, the following lines or similar ones:

```
function [c,ceq,gradc,gradceq] = democonstr(in1)
%DEMOCONST
%   [C,CEQ,GRADC,GRADCEQ] = DEMOCONST(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 2-48.



## Generate the Hessian Files

To generate the Hessian of the Lagrangian for the problem, first generate files for the energy Hessian and for the constraint Hessians.

The Hessian of the objective function, `hessenergy`, is a very large symbolic expression, containing over 150,000 symbols, as evaluating `size(char(hessenergy))` shows. So it takes a substantial amount of time to run `matlabFunction(hessenergy)`.

To generate a file `hessenergy.m`, run the following two lines:

```
filename = [currdir,'hessenergy.m'];
matlabFunction(hessenergy,'file',filename,'vars',{x});
```

In contrast, the Hessians of the constraint functions are small, and fast to compute:

```
for i = 1:10
    ii = num2str(i);
    thename = ['hessc',ii];
    filename = [currdir,thename,'.m'];
    matlabFunction(hessc{i},'file',filename,'vars',{x});
end
```

After generating all the files for the objective and constraints, put them together with the appropriate Lagrange multipliers in a file `hessfinal.m` as follows:

```
function H = hessfinal(X,lambda)
%
% Call the function hessenergy to start
H = hessenergy(X);

% Add the Lagrange multipliers * the constraint Hessians
H = H + hessc1(X) * lambda.ineqnonlin(1);
H = H + hessc2(X) * lambda.ineqnonlin(2);
H = H + hessc3(X) * lambda.ineqnonlin(3);
H = H + hessc4(X) * lambda.ineqnonlin(4);
H = H + hessc5(X) * lambda.ineqnonlin(5);
H = H + hessc6(X) * lambda.ineqnonlin(6);
H = H + hessc7(X) * lambda.ineqnonlin(7);
H = H + hessc8(X) * lambda.ineqnonlin(8);
H = H + hessc9(X) * lambda.ineqnonlin(9);
H = H + hessc10(X) * lambda.ineqnonlin(10);
```

## Run the Optimization

Start the optimization with the electrons distributed randomly on a sphere of radius 1/2 centered at  $[0,0,-1]$ :

```
rng default % for reproducibility
Xinitial = randn(3,10); % columns are normal 3-D vectors
for j=1:10
    Xinitial(:,j) = Xinitial(:,j)/norm(Xinitial(:,j))/2;
    % this normalizes to a 1/2-sphere
end
Xinitial(3,:) = Xinitial(3,:) - 1; % center at [0,0,-1]
Xinitial = Xinitial(:); % Convert to a column vector
```

Set the options to use the interior-point algorithm, and to use gradients and the Hessian:

```
options = optimoptions(@fmincon,'Algorithm','interior-point','SpecifyObjectiveGradient'
    'SpecifyConstraintGradient',true,'HessianFcn',@hessfinal,'Display','final');
```

Call `fmincon`:

```
[xfinal fval exitflag output] = fmincon(@demoenergy,Xinitial,...
    A,b,[],[],[],[],@democonstr,options);
```

The solution takes 19 iterations and only 28 function evaluations:

```
xfinal,fval,exitflag,output.iterations,output.funcCount
```

```
xfinal =
-0.0317
 0.0317
-1.9990
 0.6356
-0.6356
-1.4381
 0.0000
-0.0000
-0.0000
 0.0000
-1.0000
-1.0000
 1.0000
-0.0000
-1.0000
-1.0000
-0.0000
-1.0000
 0.6689
 0.6644
```

```
-1.3333  
-0.6667  
0.6667  
-1.3333  
0.0000  
1.0000  
-1.0000  
-0.6644  
-0.6689  
-1.3333
```

```
fval =
```

```
34.1365
```

```
exitflag =
```

```
1
```

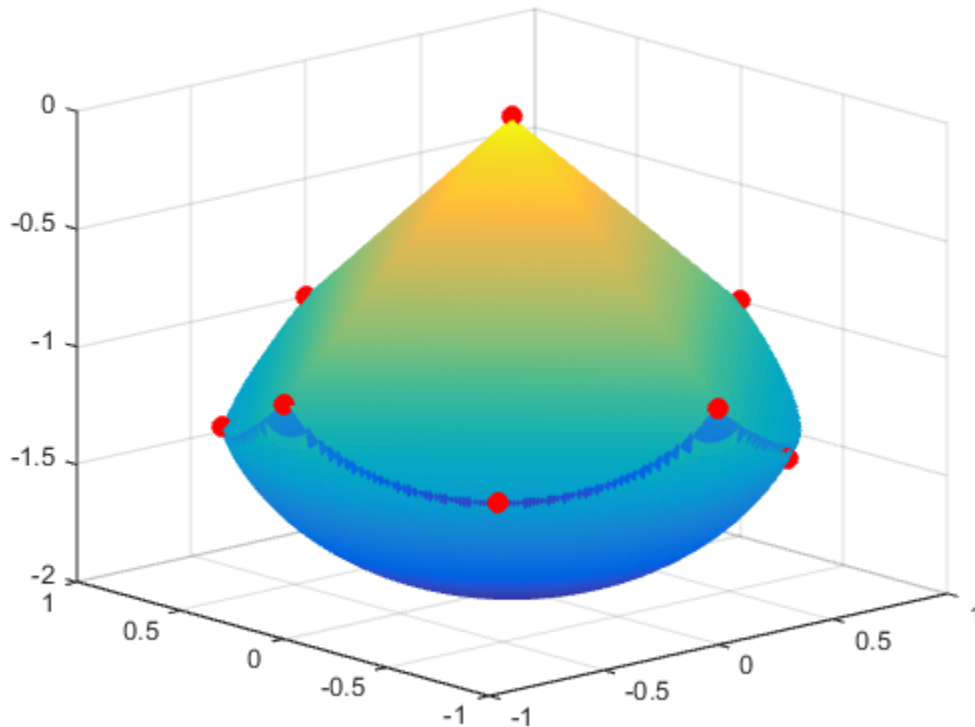
```
ans =
```

```
19
```

```
ans =
```

```
28
```

Even though the initial positions of the electrons were random, the final positions are nearly symmetric:



## Compare to Optimization Without Gradients and Hessians

The use of gradients and Hessians makes the optimization run faster and more accurately. To compare with the same optimization using no gradient or Hessian information, set the options not to use gradients and Hessians:

```
options = optimoptions(@fmincon,'Algorithm','interior-point',...  
    'Display','final');  
[xfinal2 fval2 exitflag2 output2] = fmincon(@demoenergy,Xinitial,...  
    A,b,[],[],[],[],@democonstr,options);
```

The output shows that `fmincon` found an equivalent minimum, but took more iterations and many more function evaluations to do so.

```
xfinal2, fval2, exitflag2, output2.iterations, output2.funcCount
```

```
xfinal2 =
```

```
    0.0000  
    1.0000  
   -1.0000  
    0.6689  
   -0.6644  
   -1.3334  
   -0.6644  
    0.6689  
   -1.3334  
    0.0000  
   -1.0000  
   -1.0000  
    0.6357  
    0.6357  
   -1.4380  
   -0.0317  
   -0.0317  
   -1.9990  
    1.0000  
    0.0000  
   -1.0000  
   -1.0000  
    0.0000  
   -1.0000  
    0.0000  
    0.0000  
   -0.0000  
   -0.6667  
   -0.6667  
   -1.3334
```

```
fval2 =
```

```
    34.1365
```

```
exitflag2 =
```

```
    1
```

```
ans =
```

```
    77
```

```
ans =
```

```
   2435
```

In this run the number of function evaluations (in `output2.funcCount`) is 2435 compared to 28 (in `output.funcCount`) when using gradients and Hessian.

## Clear the Symbolic Variable Assumptions

The symbolic variables in this example have the assumption, in the symbolic engine workspace, that they are real. To clear this assumption from the symbolic engine workspace, it is not sufficient to delete the variables. Clear the variable assumptions by using `syms`:

```
syms x
```

Verify that the assumptions are empty.

```
assumptions(x)
```

```
ans =
```

```
Empty sym: 1-by-0
```

## See Also

### Related Examples

- “Using Symbolic Mathematics with Optimization Toolbox™ Solvers” on page 6-129
- “Constrained Electrostatic Nonlinear Optimization, Problem-Based” on page 7-13

## 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.

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:

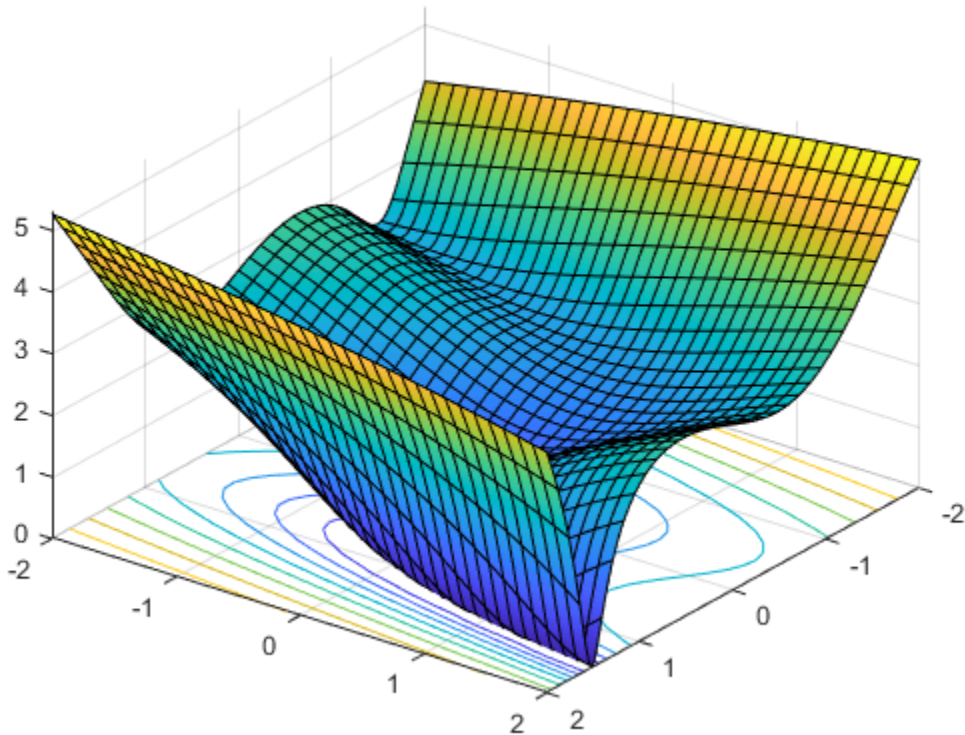
$$f(x_1, x_2) = \log\left(1 + 3(x_2 - (x_1^3 - x_1))^2 + (x_1 - 4/3)^2\right).$$

This function is positive, with a unique minimum value of zero attained at  $x_1 = 4/3$ ,  $x_2 = (4/3)^3 - 4/3 = 1.0370\dots$

We write the independent variables as  $x_1$  and  $x_2$  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((x1 - 4/3)^2 + 3(-x1^3 + x1 + x2)^2 + 1)

fsurf(f,[-2 2], 'ShowContours', 'on')
view(127,38)
```



Compute the gradient and Hessian of f:

```
gradf = jacobian(f,x).' % column gradf
```



$$\text{gradf} = \begin{pmatrix} -\frac{6(3x_1^2 - 1)(-x_1^3 + x_1 + x_2) - 2x_1 + \frac{8}{3}}{\sigma_1} \\ \frac{-6x_1^3 + 6x_1 + 6x_2}{\sigma_1} \end{pmatrix}$$

where

$$\sigma_1 = \left(x_1 - \frac{4}{3}\right)^2 + 3(-x_1^3 + x_1 + x_2)^2 + 1$$

$$\text{hessf} = \text{jacobian}(\text{gradf}, \mathbf{x})$$

$$\text{hessf} = \begin{pmatrix} \frac{6(3x_1^2 - 1)^2 - 36x_1(-x_1^3 + x_1 + x_2) + 2}{\sigma_2} - \frac{\sigma_3^2}{\sigma_2^2} & \sigma_1 \\ \sigma_1 & \frac{6}{\sigma_2} - \frac{(-6x_1^3 + 6x_1 + 6x_2)^2}{\sigma_2^2} \end{pmatrix}$$

where

$$\sigma_1 = \frac{(-6x_1^3 + 6x_1 + 6x_2)\sigma_3}{\sigma_2^2} - \frac{18x_1^2 - 6}{\sigma_2}$$

$$\sigma_2 = \left(x_1 - \frac{4}{3}\right)^2 + 3(-x_1^3 + x_1 + x_2)^2 + 1$$

$$\sigma_3 = 6(3x_1^2 - 1)(-x_1^3 + x_1 + x_2) - 2x_1 + \frac{8}{3}$$

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

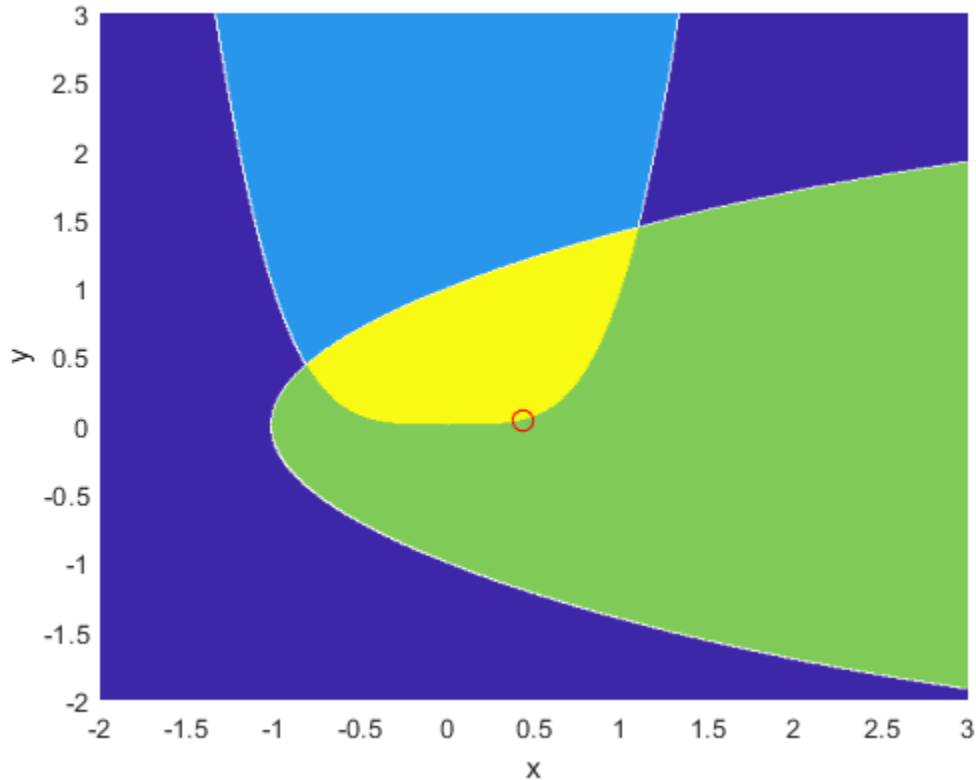
We consider the same objective function and starting point, but now have two nonlinear constraints:

$$5\sinh(x_2/5) \geq x_1^4$$

$$5 \tanh(x_1/5) \geq x_2^2 - 1.$$

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
```

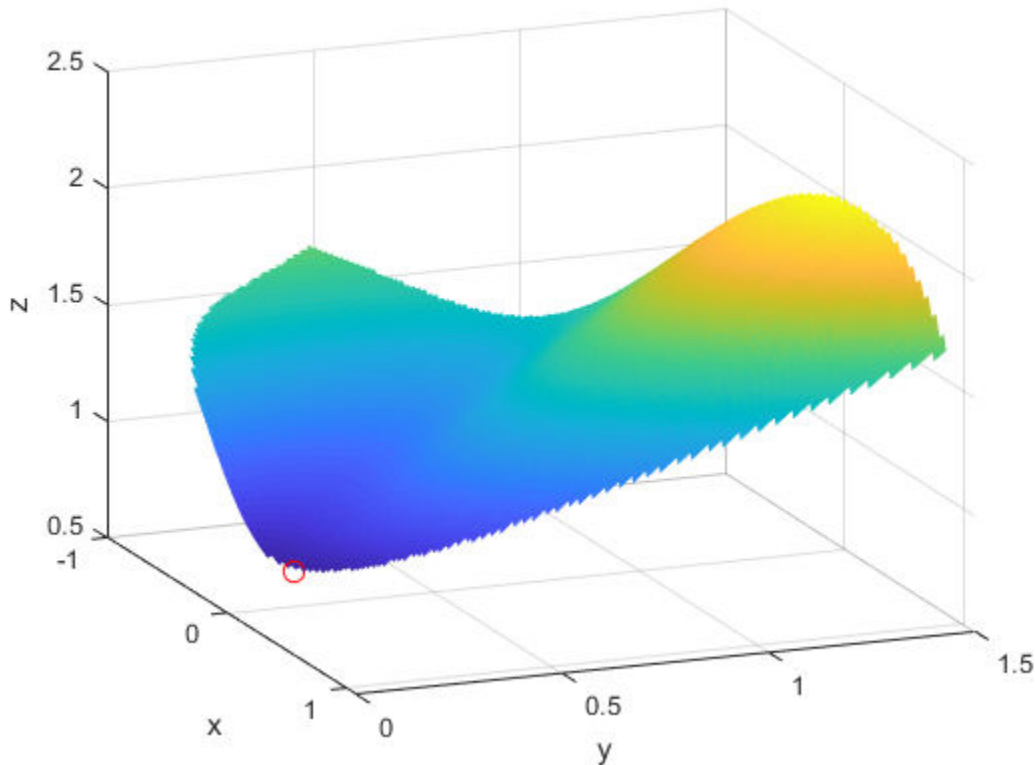


We plotted a small red circle around the optimal point.

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  $c(x) \leq 0$ . 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: '...'
```

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: 6.0309e-07
    algorithm: 'interior-point'
    firstorderopt: 3.8435e-07
    cgiterations: 0
    message: '...'
```

```
fprintf(['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.'
```

```
fprintf(['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

- “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115

## 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

$$\begin{array}{rclcl} 1 & \leq & w_1 & \leq & 100, \\ 1 & \leq & w_2 & \leq & 100. \end{array}$$

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

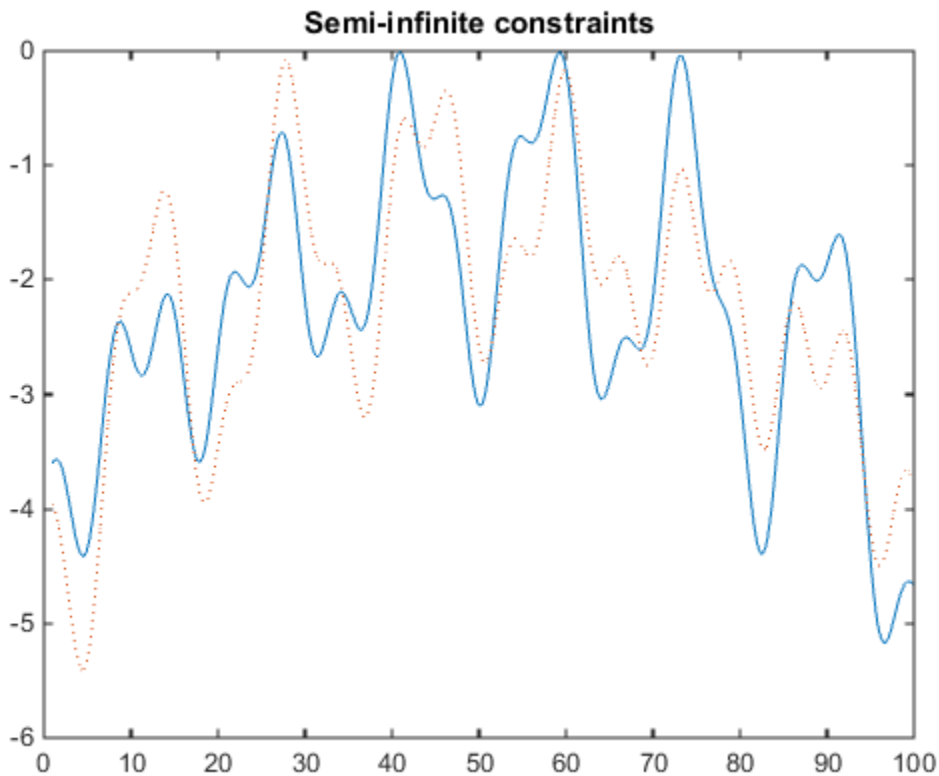
```
x
x =
    0.6675
    0.3012
    0.4022
```

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 6-145

- “Analyzing the Effect of Uncertainty Using Semi-Infinite Programming” on page 6-148
- “Analyzing the Effect of Uncertainty Using Semi-Infinite Programming”

## 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

$$K_1(x, w) = \sin(w_1 x_1) \cos(w_2 x_2) - \frac{1}{1000} (w_1 - 50)^2 - \sin(w_1 x_3) - x_3 + \dots$$

$$\sin(w_2 x_2) \cos(w_1 x_1) - \frac{1}{1000} (w_2 - 50)^2 - \sin(w_2 x_3) - x_3 \leq 1.5,$$

for all values of  $w_1$  and  $w_2$  over the ranges

$$\begin{array}{rclcl} 1 & \leq & w_1 & \leq & 100, \\ 1 & \leq & w_2 & \leq & 100, \end{array}$$

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
```

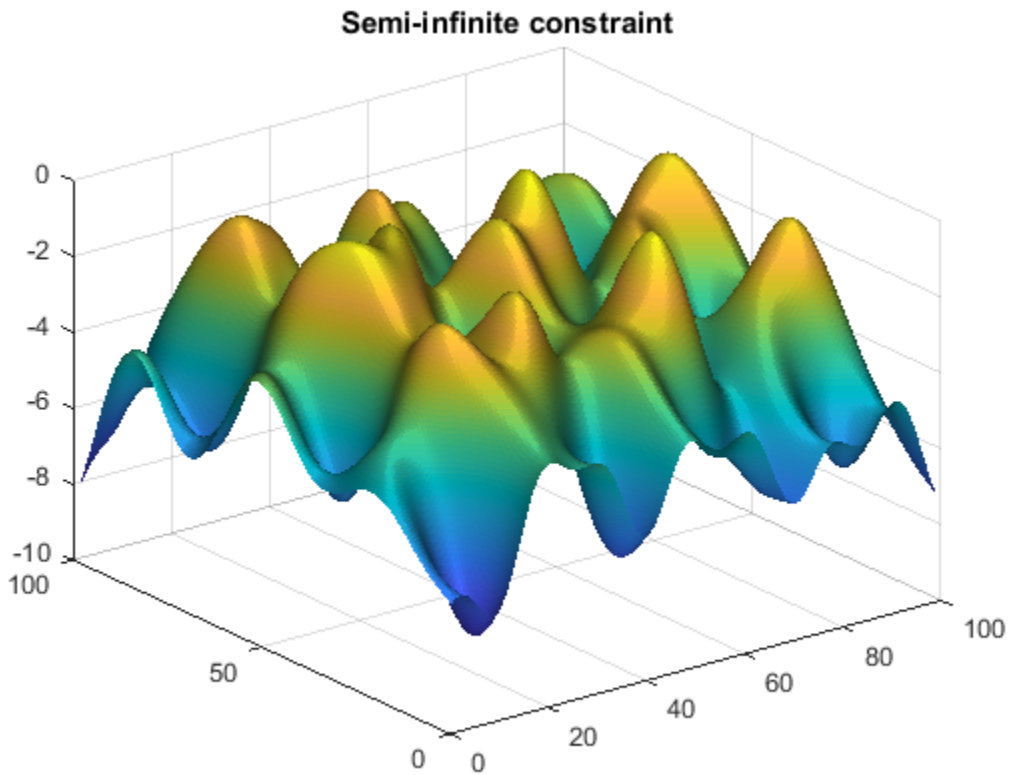
The goal was to minimize the objective  $f(x)$  such that the semi-infinite constraint satisfied  $K_1(x,w) \leq 1.5$ . Evaluating `mycon` at the solution `x` and looking at the maximum element of the matrix `K1` shows the constraint is easily satisfied.

```
[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`.





## See Also

fseminf

## Related Examples

- “One-Dimensional Semi-Infinite Constraints” on page 6-141
- “Analyzing the Effect of Uncertainty Using Semi-Infinite Programming” on page 6-148
- “Analyzing the Effect of Uncertainty Using Semi-Infinite Programming”

## 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 Estadística e Investigación 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.

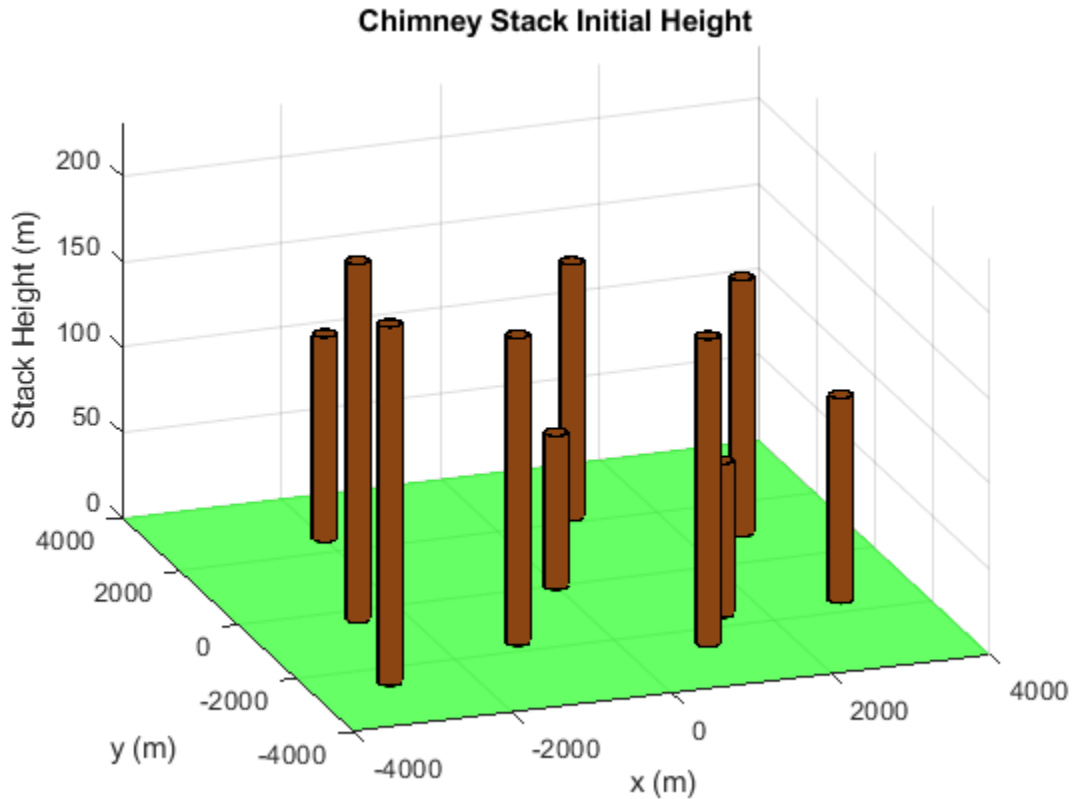
### Minimal Stack Height Problem

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 (SO<sub>2</sub>) over the entire region R (remember that the plot of chimney stacks was over a smaller region). The SO<sub>2</sub> concentration has been calculated with the chimney stacks set to their initial heights.

We can see that the concentration of SO<sub>2</sub> varies over the region of interest. There are two features of the Sulfur Dioxide graph of note:

- SO<sub>2</sub> concentration rises in the top left hand corner of the (x,y) plane
- SO<sub>2</sub> concentration is approximately zero throughout most of the region

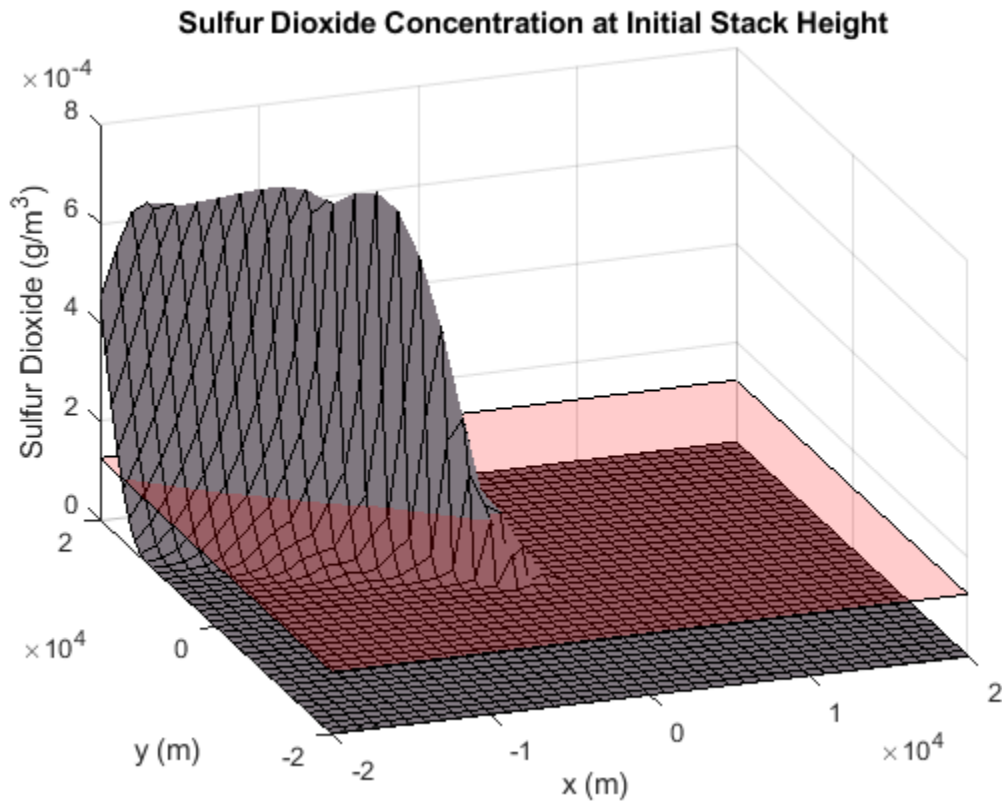
In very simple terms, the first feature is due to the prevailing wind, which is blowing SO<sub>2</sub> toward the top left hand corner of the (x,y) plane in this example. The second factor is due to SO<sub>2</sub> being transported to the ground via diffusion. This is a slower process compared to the prevailing wind and thus SO<sub>2</sub> 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 SO<sub>2</sub> concentration of  $0.000125gm^{-3}$ . 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 SO<sub>2</sub> 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');
```



### 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

$$Ax \leq b \text{ (Linear inequality constraints)}$$

$$Aeq*x = beq \text{ (Linear equality constraints)}$$

$$c(x) \leq 0 \text{ (Nonlinear Inequality Constraints)}$$

$ceq(x) = 0$  (Nonlinear Equality Constraints)

$l \leq x \leq u$  (Bounds)

and

$K_j(x, w) \leq 0$ , where  $w \in I_j$  for  $j = 1, \dots, n_{inf}$  (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  $0.000125\text{gm}^{-3}$  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;
```

```
wly = -20000:s(1,2):20000;
[t1,t2] = meshgrid(wlx,wly);

% 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.
```

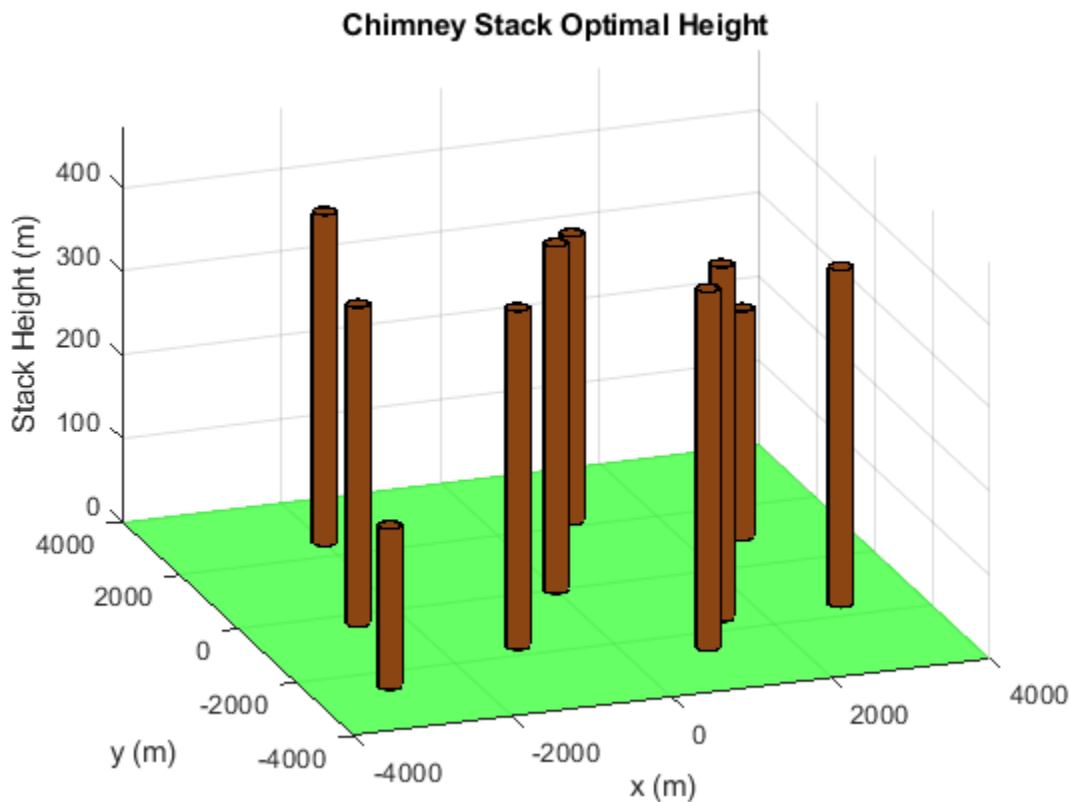
```
fprintf('\nMinimum computed cumulative height of chimney stacks : %7.2f m\n', sumh);
```

```
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');
```



### 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.

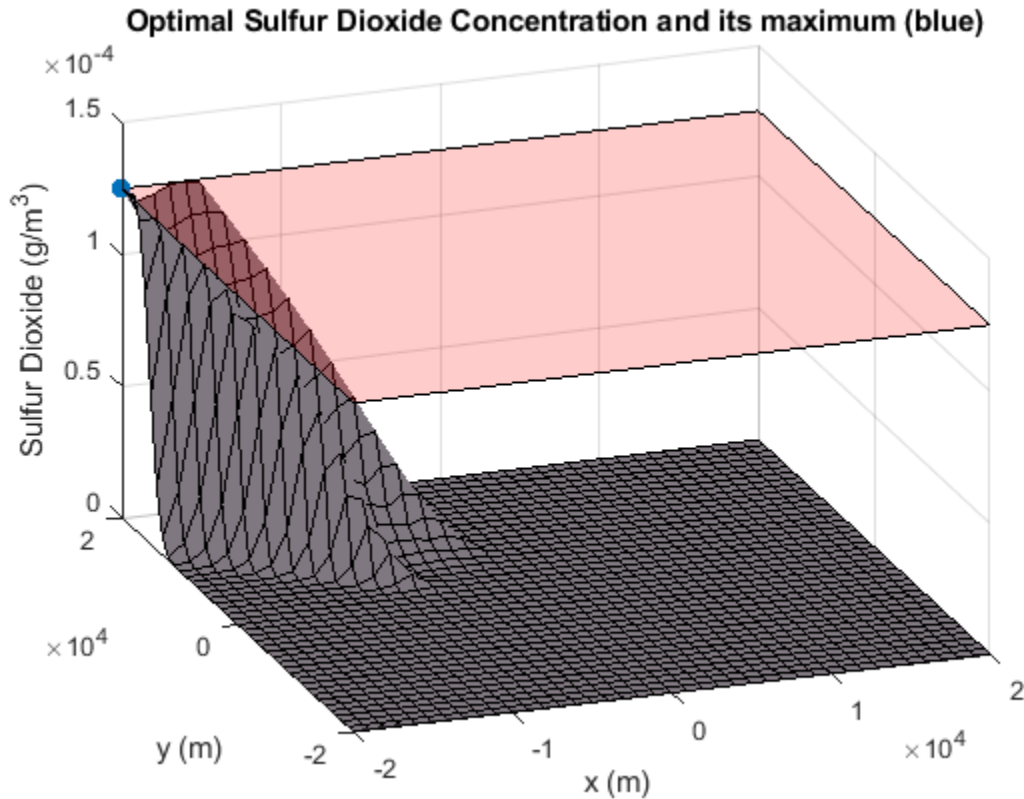
Note that the sulfur dioxide concentration takes its maximum possible value in the upper left corner of the  $(x, y)$  plane, i.e. at  $x = -20000\text{m}$ ,  $y = 20000\text{m}$ . This point is marked by the blue dot in the figure below and verified by calculating the sulfur dioxide concentration at 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);

```



```

S02Max = concSulfurDioxide(-20000, 20000, hsopt, theta0, U0);
fprintf('Sulfur Dioxide Concentration at x = -20000m, y = 20000m : %e g/m³\n', S02Max);

```

Sulfur Dioxide Concentration at x = -20000m, y = 20000m : 1.250000e-04 g/m<sup>3</sup>

### 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 SO<sub>2</sub> concentration at the optimum stack height in the previous problem. In particular, the SO<sub>2</sub> concentration takes its maximum possible value at  $x = -20000\text{m}$ ,  $y = 20000\text{m}$ . To reduce the number of "infinite" variables, we will assume that the SO<sub>2</sub> concentration will also take its maximum value at this point when uncertainty is present. We then require that SO<sub>2</sub> concentration at this point is below  $0.000125\text{gm}^{-3}$  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 0(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 SO<sub>2</sub> 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 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.
```

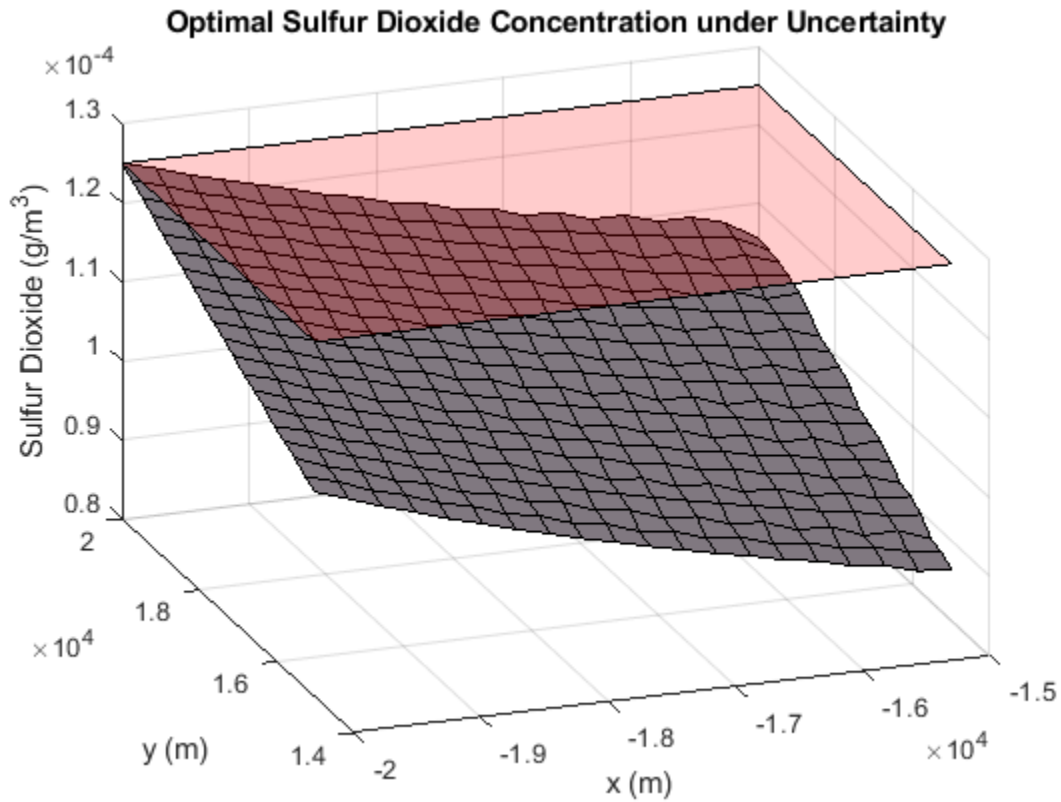
```
fprintf('\nMinimal computed cumulative height of chimney stacks with uncertainty: %7.2f\n');
```

```
Minimal computed cumulative height of chimney stacks with uncertainty: 3812.15 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 SO<sub>2</sub> concentration to remain below the legislated maximum for all wind speed/direction pairs in the specified range.

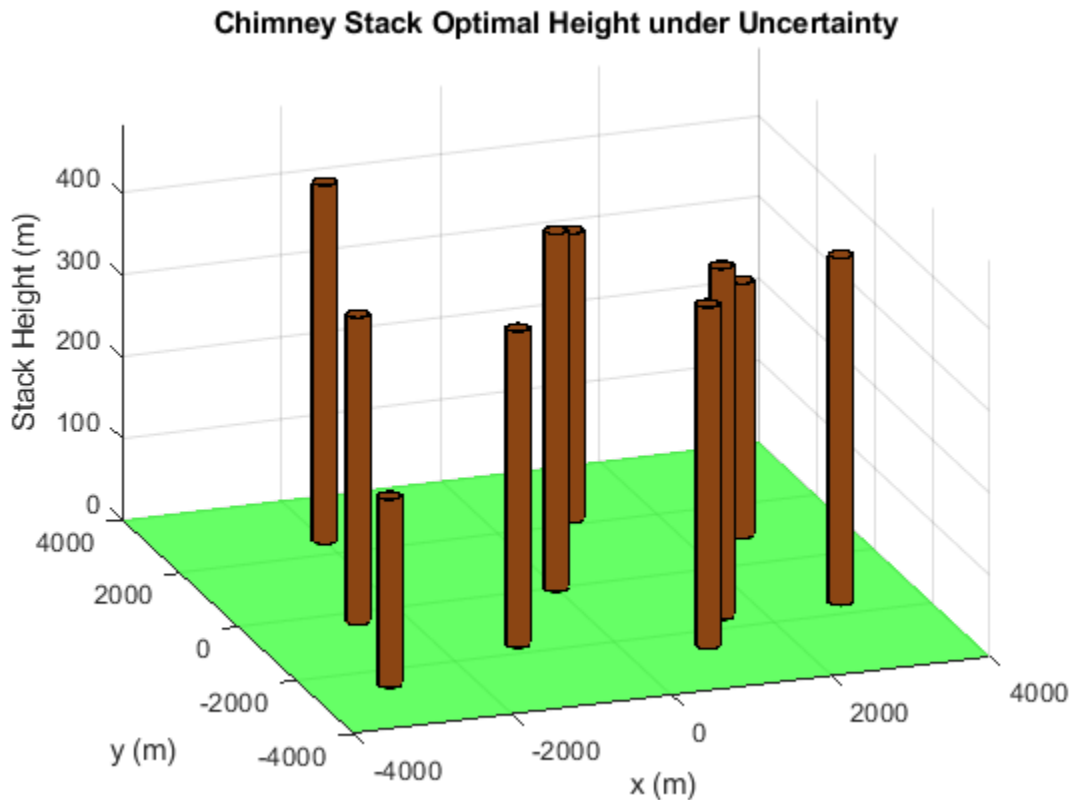
We can check that the sulfur dioxide concentration does not exceed the limit over the region of interest via inspection of a sulfur dioxide plot. For a given (x, y) point, we plot the maximum SO<sub>2</sub> concentration for the wind speed and direction in the stated ranges. Note that we have zoomed 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');
```



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 6-141
- “Two-Dimensional Semi-Infinite Constraint” on page 6-145



# Nonlinear Problem-Based

---

- “Rational Objective Function, Problem-Based” on page 7-2
- “Solve Constrained Nonlinear Optimization, Problem-Based” on page 7-4
- “Convert Nonlinear Function to Optimization Expression” on page 7-8
- “Constrained Electrostatic Nonlinear Optimization, Problem-Based” on page 7-13
- “Problem-Based Nonlinear Minimization with Linear Constraints” on page 7-20
- “Include Derivatives in Problem-Based Workflow” on page 7-24
- “Output Function for Problem-Based Optimization” on page 7-31

## 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, you can create the objective function expression directly from the variables. (In contrast, when your objective function is not a rational function, you must create a MATLAB® function that represents the objective and then convert the function to an expression by using `fcn2optimexpr`.)

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)
```

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: 30  
    stepsize: 1.7081e-06  
    lssteplength: 1  
    firstorderopt: 1.3411e-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 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

$$f(x, y) = e^x(4x^2 + 2y^2 + 4xy + 2y - 1).$$

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');
```

Convert the objective function into an optimization expression by using `fcn2optimexpr`. 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.

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

Create a nonlinear constraint that the solution lies in a tilted ellipse, specified as

$$\frac{xy}{2} + (x + 2)^2 + \frac{(y - 2)^2}{2} \leq 2.$$

You do not need to create and convert the constraint as a separate function because it is a rational function of 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.

```
showproblem(prob)
```

```
OptimizationProblem :
```

```
  minimize :  
    objfunx(x, y)
```

```
  subject to constr:  
    (((x .* y) ./ 2) + (x + 2).^2) + ((y - 2).^2 ./ 2) <= 2
```

Solve the problem.

```
[sol,fval] = solve(prob,x0)
```

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)
```

Local minimum possible. Constraints satisfied.

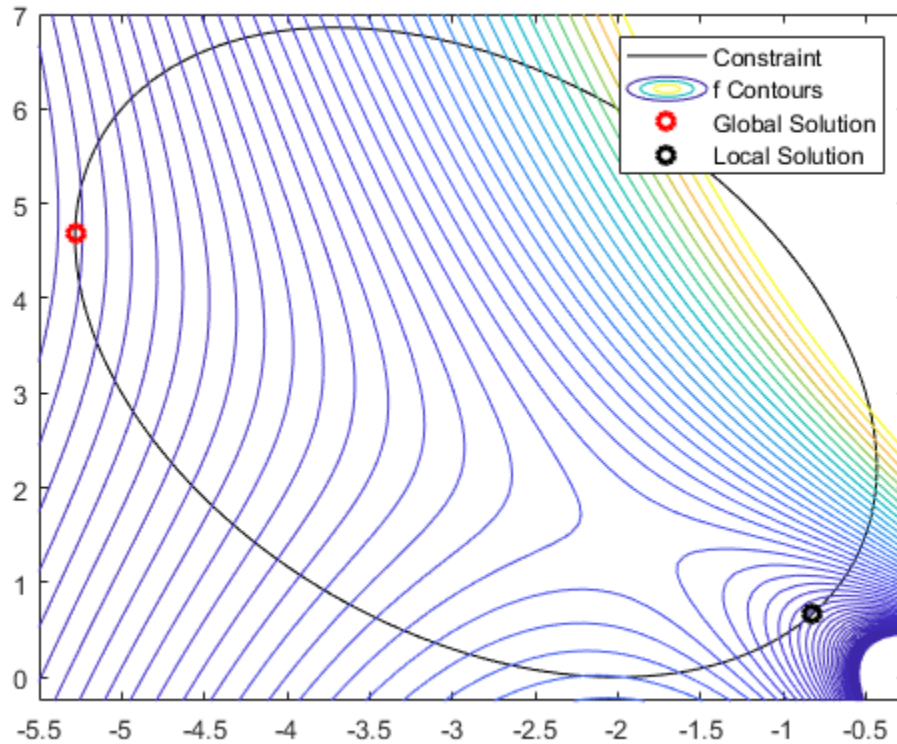
fmincon stopped because the size of the current step is less than the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.

```
sol2 = struct with fields:  
    x: -0.8211  
    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;  
rng = [-5.5 -0.25 -0.25 7];  
fimplicit(g, 'k-')  
axis(rng);  
hold on  
fcontour(f, rng, '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  $[-1, 4]$ .

## See Also

`fcn2optimexpr`

## More About

- “Problem-Based Nonlinear Optimization”

## Convert Nonlinear Function to Optimization Expression

To use a nonlinear function as an objective or nonlinear constraint function in the problem-based approach, convert the function to an optimization expression using `fcn2optimexpr`. This example shows how to convert the function using both a function file and an anonymous function.

### Function File

To use a function file in the problem-based approach, you need to 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;
```

To use this function file 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 `f` and convert it.

```

fun = @(x,y)-exp(-y'*x*y);
funexpr = fcn2optimexpr(fun,u,v,'OutputSize',[1,1])

funexpr =
    Nonlinear OptimizationExpression

    anonymousFunction1(u, v)

where:

    anonymousFunction1 = @(x,y)-exp(-y'*x*y);

```

### Create Objective

To use either expression as an objective function, create an optimization problem.

```

prob = optimproblem;
prob.Objective = f;
% Or, equivalently, prob.Objective = funexpr;

```

### Define Constraints

Define the constraint  $g \leq 0$  in the optimization problem.

```
prob.Constraints.nlcons1 = g <= 0;
```

Also define the constraints that  $u$  is symmetric and that  $\text{mineval} \geq -1/2$ .

```
prob.Constraints.sym = u == u.';
prob.Constraints.mineval = mineval >= -1/2;
```

View the problem.

```
showproblem(prob)
```

```
OptimizationProblem :

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 = randn(3);
x0.u = x0.u + x0.u.';
x0.v = 2*randn(3,1);
[sol,fval,exitflag,output] = solve(prob,x0)

```

Solver stopped prematurely.

`fmincon` stopped because it exceeded the function evaluation limit,  
`options.MaxFunctionEvaluations = 3.000000e+03`.

```

sol = struct with fields:
  u: [3x3 double]
  v: [3x1 double]

```

```
fval = -403.4288
```

```

exitflag =
  SolverLimitExceeded

```

```

output = struct with fields:
  iterations: 203
  funcCount: 3012
  constrviolation: 0
  stepsize: 8.5222e-05
  algorithm: 'interior-point'
  firstorderopt: 0.0980
  cgiterations: 222
  message: '...'
  solver: 'fmincon'

```

View the solution.

```
disp(sol.u)
```

```
    0.6521    0.7607   -0.3914  
    0.7607    0.0026   -0.2581  
   -0.3914   -0.2581   -0.3666
```

```
disp(sol.v)
```

```
    2.0000  
   -2.0000  
    2.0000
```

The solution matrix  $u$  is symmetric. The solution vector  $v$  has all entries at bound constraints.

## See Also

`fcn2optimexpr`

## More About

- “Problem-Based Optimization Setup”

## Constrained Electrostatic Nonlinear Optimization, Problem-Based

Consider the electrostatics problem of placing 10 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].

For an equivalent solver-based example using Symbolic Math Toolbox™, see “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

### Problem Geometry

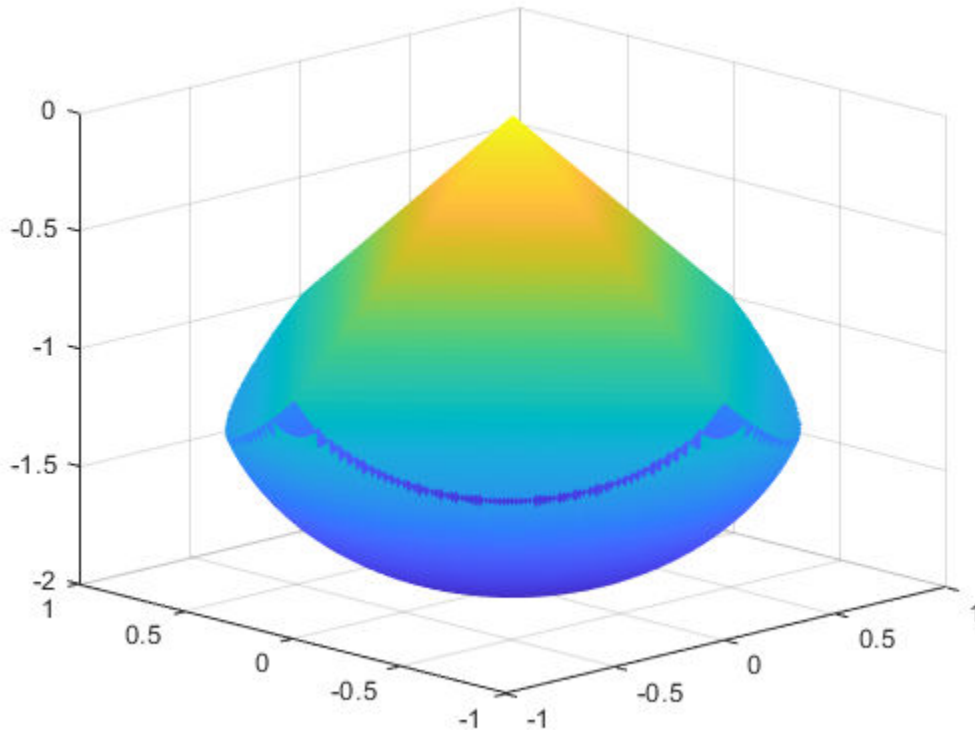
This example involves a conducting body defined by the following inequalities. For each electron with coordinates  $(x, y, z)$ ,

$$z \leq -|x| - |y|$$

$$x^2 + y^2 + (z + 1)^2 \leq 1.$$

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, since we'll plot more 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.

### Define Problem Variables

The problem has ten 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 = 10;  
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 `showconstr`.

```
showconstr(elecprob.Constraints.spherec)

((x.^2 + y.^2) + (z + 1).^2) <= arg_RHS

where:

    arg2 = 1;
    arg1 = arg2([1 1 1 1 1 1 1 1 1]);
    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`. For a different approach, write the absolute value constraint as four linear inequalities. Each constraint command returns a vector of ten 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.

```
energy = optimexpr(1);
for ii = 1:(N-1)
```

```
    for jj = (ii+1):N
        tempe = (x(ii) - x(jj))^2 + (y(ii) - y(jj))^2 + (z(ii) - z(jj))^2;
        energy = energy + tempe^(-1/2);
    end
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)
```

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: [10x1 double]
  y: [10x1 double]
  z: [10x1 double]
```

```
fval = 34.1365
```

```
exitflag =
  OptimalSolution
```

```
output = struct with fields:
  iterations: 94
```

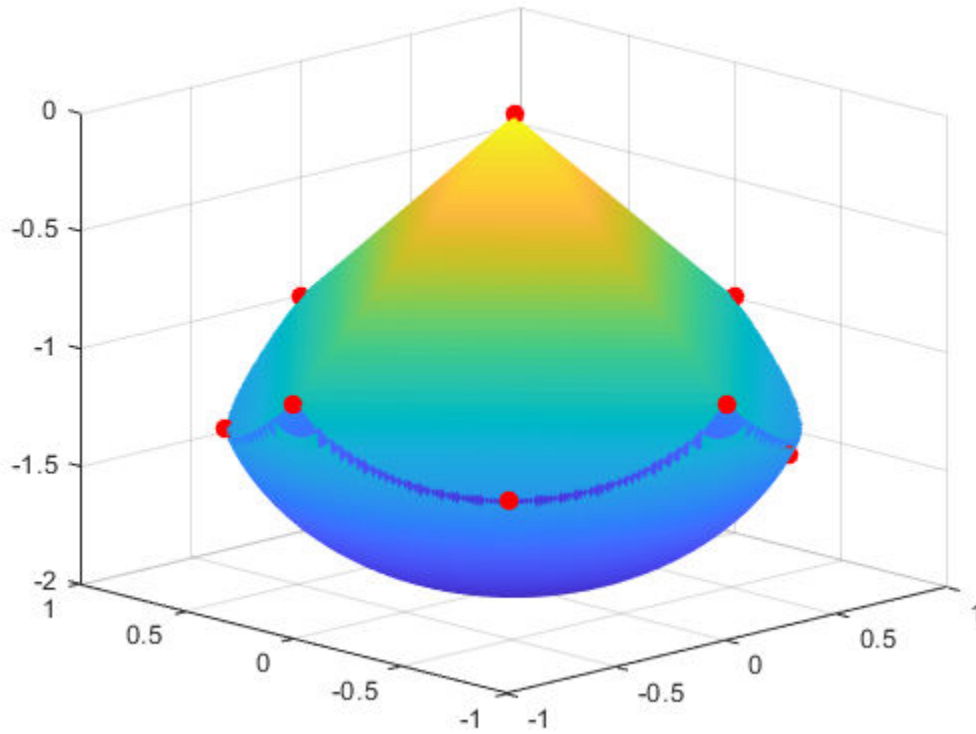


```
    funcCount: 2959
  constrviolation: 0
    stepsize: 4.1829e-07
  algorithm: 'interior-point'
firstorderopt: 1.2800e-06
  cgiterations: 0
  message: '...'
  solver: 'fmincon'
```

### **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. The electrons are mainly on 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

- "Symbolic Math Toolbox Calculates Gradients and Hessians" on page 6-115
- "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” on page 6-107 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

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

subject to a set of linear equality constraints  $A_{eq} * x = b_{eq}$ . 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. Convert the function to an optimization expression using `fcn2optimexpr`.

```
prob.Objective = fcn2optimexpr(@brownfgh,x,'OutputSize',[1,1]);
```

### Include Constraints

To obtain the `Aeq` and `beq` matrices in your workspace, execute this command.

```
load browneg
```

Include the linear constraints in the problem.

```
prob.Constraints = Aeq*x == beq;
```

## Review and Solve Problem

Review the problem objective.

```
showexpr(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.

```
% showconstr(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)
```

```
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: 2.6246e-13
    stepsize: 1.9303
    algorithm: 'interior-point'
    firstorderopt: 2.6432
    cgiterations: 0
```

```
message: '...'  
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)
```

```
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: 33066  
  constrviolation: 2.1316e-14  
  stepsize: 1.1197e-05  
  algorithm: 'interior-point'  
  firstorderopt: 4.5457e-06  
  cgiterations: 2  
  message: '...'  
  solver: 'fmincon'
```

### **Compare with Solver-Based Solution**

To solve the problem using the solver-based approach as shown in “Minimization with Linear Equality Constraints” on page 6-107, 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” on page 6-107 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 “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

## See Also

`fcn2optimexpr`

## More About

- “Minimization with Linear Equality Constraints” on page 6-107
- “Problem-Based Optimization Setup”

## Include Derivatives in Problem-Based Workflow

### In this section...

“Why Include Derivatives?” on page 7-24

“Create Optimization Problem” on page 7-24

“Convert Problem to Solver-Based Form” on page 7-25

“Calculate Derivatives and Keep Track of Variables” on page 7-25

“Edit the Objective and Constraint Files” on page 7-26

“Run Problem Using Two Methods” on page 7-27

### Why Include Derivatives?

The example shows how to include derivative information in nonlinear problem-based optimization. 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.
- Analytic gradients can be more accurate than finite difference estimates.
- Including a Hessian can lead to faster convergence, meaning fewer iterations.
- 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.

Despite these advantages, the problem-based approach currently does not use derivative information. To use derivatives in problem-based optimization, convert your problem using `prob2struct`, and edit the resulting objective and constraint functions.

### Create Optimization Problem

Create a problem-based nonlinear optimization. With 2-D control variables  $x$  and  $y$ , use the objective function

$$\text{fun1} = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

$$\text{fun2} = \exp\left(-\sum (x_i - y_i)^2\right) \exp(-\exp(-y_1)) \text{sech}(y_2)$$

$$\text{objective} = \text{fun1} + \text{fun2}.$$



Include the constraint that the sum of squares of  $x$  and  $y$  is no more than 4.

`fun2` is not a rational function of its arguments. Therefore, to include it in an optimization problem, you must convert it to an optimization expression using `fcn2optimexpr`.

```
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 + fcn2optimexpr(fun2,x,y);
prob.Constraints.cons = sum(x.^2 + y.^2) <= 4;
```

## Convert Problem to Solver-Based Form

To include derivatives, convert the problem to a structure using `prob2struct`.

```
problem = prob2struct(prob);
```

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'.

## Calculate Derivatives and Keep Track of Variables

Calculate the derivatives associated with the objective and nonlinear constraint functions. If you have a Symbolic Math Toolbox license, you can use the `gradient` or `jacobian` function to help compute the derivatives. See “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

The solver-based approach has one control variable. Each optimization variable ( $x$  or  $y$ , 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, 'generatedObjective.m'. The beginning of the file contains these lines of code:

```
%% Variable indices.
idx_x = [1 2];
idx_y = [3 4];

%% Map solver-based variables to problem-based.
```

```
x = inputVariables(idx_x);
x = x(:);
y = inputVariables(idx_y);
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
```

Once you know the mapping, you can use standard calculus to find the following expressions for the gradient `grad` of the objective function `objective = fun1 + fun2` with respect to the control variable `[x(:);y(:)]`.

$$\text{grad} = \begin{bmatrix} 2x_1 - 400x_1(x_2 - x_1^2) + \sigma_1 - 2 \\ 200x_2 - 200x_1^2 + \sigma_2 \\ -\sigma_1 - d\exp(-y_1) \\ -\sigma_2 + d \tanh(y_2) \end{bmatrix},$$

where

$$c = \exp(-(x_1 - y_1)^2 - (x_2 - y_2)^2)$$

$$d = c\exp(-\exp(-y_1))\text{sech}(y_2)$$

$$\sigma_1 = 2(x_1 - y_1)d$$

$$\sigma_2 = 2(x_2 - y_2)d.$$

## Edit the Objective and Constraint Files

To include the calculated gradients in the objective function file, edit 'generatedObjective.m' as follows.

```

%% Insert gradient calculation here.
% If you include a gradient, notify the solver by setting the
% SpecifyObjectiveGradient option to true.
if nargout > 1
    c = exp(-sum((x - y).^2));
    d = c*exp(-exp(-y(1)))*sech(y(2));
    sigma1 = 2*(x(1) - y(1))*d;
    sigma2 = 2*(x(2) - y(2))*d;
    grad = zeros(4,1);
    grad(1) = 2*x(1) - 400*x(1)*(x(2) - x(1)^2) + sigma1 - 2;
    grad(2) = 200*x(2) - 200*x(1)^2 + sigma2;
    grad(3) = -sigma1 - d*exp(-y(1));
    grad(4) = -sigma2 + d*tanh(y(2));
end

```

Recall that the nonlinear constraint is  $x(1)^2 + x(2)^2 + y(1)^2 + y(2)^2 \leq 4$ . Clearly, the gradient of this constraint function is  $2*[x;y]$ . To include the calculated gradients of the nonlinear constraint, edit 'generatedConstraints.m' as follows.

```

%% 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 Using Two Methods

Run the problem using both the problem-based and solver-based methods to see the differences. To run the 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
```

Compare this solution with the one obtained from `solve`, which uses no derivative information.

```
init.x = x0(1:2);  
init.y = x0(3:4);  
[xproblem,fvalproblem,exitflagproblem,outputproblem] = solve(prob,init)
```

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: 48  
  funcCount: 276  
  constrviolation: 0  
  stepsize: 7.9340e-07  
  algorithm: 'interior-point'  
  firstorderopt: 6.5496e-07  
  cgiterations: 9  
  message: '↵Local minimum found that satisfies the constraints.↵↵Optimization terminated 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.'  
  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, 48 without). However, the solution using gradients requires only 77 function evaluations, compared to 276 for the solution without gradients.

### **See Also**

`prob2struct` | `varindex`

### **More About**

- “Including Gradients and Hessians” on page 2-25

## 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” on page 3-39.

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  $y$ :

$$f = \exp(x)(4x^2 + 2y^2 + 4xy + 2y + 1).$$

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;
```

The objective function is in the `objfun.m` file.

```
type objfun

function f = objfun(x,y)
%OBJFUN Objective function.
% Documentation example.

% Copyright 2018 The MathWorks, Inc.

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

This expression is not a rational function of its variables. Therefore, to include it in the objective, first convert the function to an optimization expression by using `fcn2optimexpr`.

```
prob.Objective = fcn2optimexpr(@objfun,x,y);
```

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 Syntax” on page 15-37.

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  $x = \text{in}(1)$  and  $y = \text{in}(2)$ .

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
```

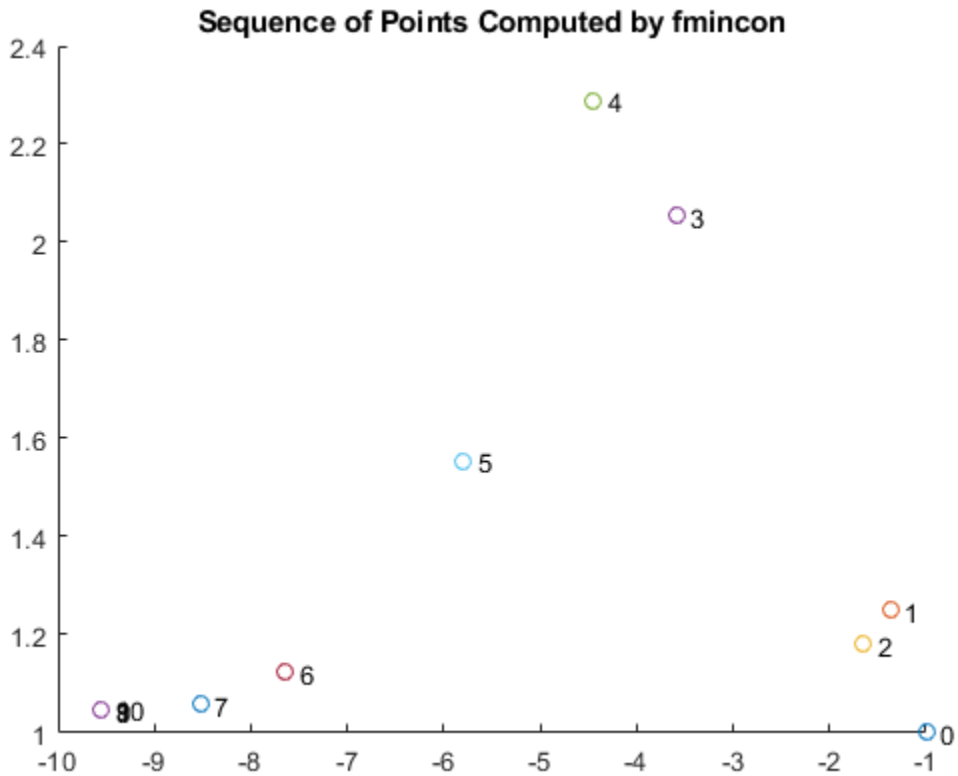
Include the output function in the optimization by setting the `OutputFcn` option. Also, set the `Algorithm` option to use the 'sqp' algorithm instead of the default 'interior-point' algorithm. Pass `idx` to the output function as an extra parameter in the last input. See “Passing Extra Parameters” on page 2-70.

```
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)
```



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: 34  
    algorithm: 'sqp'  
    message: '...'  
    constrviolation: 0  
    stepsize: 1.4788e-07  
    lssteplength: 1  
    firstorderopt: 8.0068e-10  
    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
```

## See Also

varindex

## More About

- “Output Functions” on page 3-39
- “Problem-Based Optimization Setup”



# Multiobjective Algorithms and Examples

---

- “Multiobjective Optimization Algorithms” on page 8-2
- “Compare fminimax and fminunc” on page 8-8
- “Using fminimax with a Simulink Model” on page 8-11
- “Signal Processing Using fgoalattain” on page 8-15
- “Generate and Plot a Pareto Front” on page 8-19
- “Multi-Objective Goal Attainment Optimization” on page 8-23
- “Minimax Optimization” on page 8-31

## 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

$$\frac{F_i(x) - F_i^*}{w_i} \quad (8-1)$$

This minimization is supposed to be accomplished while satisfying all types of constraints:  $c(x) \leq 0$ ,  $ceq(x) = 0$ ,  $A \cdot x \leq b$ ,  $Aeq \cdot x = beq$ , and  $l \leq x \leq u$ .

If you set all weights equal to 1 (or any other positive constant), the goal attainment problem is the same as the unscaled goal attainment problem. If the  $F_i^*$  are positive, and you set all weights as  $w_i = F_i^*$ , the goal attainment problem becomes minimizing the relative difference between the functions  $F_i(x)$  and the goals  $F_i^*$ .

In other words, the goal attainment problem is to minimize a slack variable  $\gamma$ , 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:

$$\min_{x, \gamma}$$

such that  $F(x) - w \cdot \gamma \leq F^*$ ,  $c(x) \leq 0$ ,  $ceq(x) = 0$ ,  $A \cdot x \leq b$ ,  $Aeq \cdot x = beq$ , and  $l \leq x \leq u$ .



- $f_{\min\max}$  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  $c(x) \leq 0$ ,  $ceq(x) = 0$ ,  $A \cdot x \leq b$ ,  $Aeq \cdot x = beq$ , and  $l \leq x \leq u$ .

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

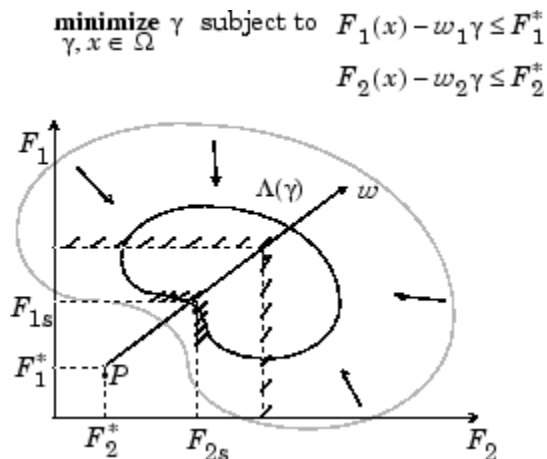
This section describes the goal attainment method of Gembicki [3]. This method uses a set of design goals,  $F^* = \{F_1^*, F_2^*, \dots, F_m^*\}$ , associated with a set of objectives,  $F(x) = \{F_1(x), F_2(x), \dots, F_m(x)\}$ . The problem formulation allows the objectives to be under- or overachieved, 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,  $w = \{w_1, w_2, \dots, w_m\}$ , and is expressed as a standard optimization problem using the formulation

$$\begin{aligned} &\text{minimize } \gamma \\ &\gamma \in \mathfrak{R}, x \in \Omega \end{aligned} \tag{8-2}$$

such that  $F_i(x) - w_i \gamma \leq F_i^*$ ,  $i = 1, \dots, m$ .

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  $\gamma$  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

$$\text{minimize } \max_i \{\Lambda_i\}, \quad x \in \mathbb{R}^n \quad (8-3)$$

where

$$\Lambda_i = \frac{F_i(x) - F_i^*}{w_i}, \quad i = 1, \dots, m.$$

Following the argument of Brayton et al. [1] for minimax optimization using SQP, using the merit function of “Equation 6-46” for the goal attainment problem of “Equation 8-3” gives

$$\psi(x, \gamma) = \gamma + \sum_{i=1}^m r_i \cdot \max\{0, F_i(x) - w_i\gamma - F_i^*\}. \quad (8-4)$$

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  $\gamma$ , 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.,

$$\psi(x) = \max_i \Lambda_i. \quad (8-5)$$

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 6-47”, giving the following:

$$\psi(x) = \sum_{i=1}^m \begin{cases} r_i \cdot \max\{0, F_i(x) - w_i\gamma - F_i^*\} & \text{if } w_i = 0 \\ \max_i \Lambda_i, \quad i = 1, \dots, m & \text{otherwise.} \end{cases} \quad (8-6)$$

Another feature that can be exploited in SQP is the objective function  $\gamma$ . 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  $\gamma$ . 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  $\gamma$ .

These changes make the Hessian,  $H$ , indefinite. Therefore  $H$  is set to have zeros in the rows and columns associated with  $\gamma$ , 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 6-32.

The preceding modifications have been implemented in `fgoalattain` and have been found to make the method more robust. However, because of the rapid convergence of the SQP method, the requirement that the merit function strictly decrease sometimes requires more function evaluations than an implementation of SQP using the merit function of “Equation 6-46”.

### Minimizing the Maximum Objective

`fminimax` uses a goal attainment method. It takes goals of 0, and weights of 1. With this formulation, the goal attainment problem becomes

$$\min_{i,x} \max \left( \frac{f_i(x) - goal_i}{weight_i} \right) = \min_{i,x} f_i(x),$$

which is the minimax problem.

Parenthetically, you might expect `fminimax` to turn the multiobjective function into a single objective. The function

$$f(x) = \max(F_1(x), \dots, F_j(x))$$

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

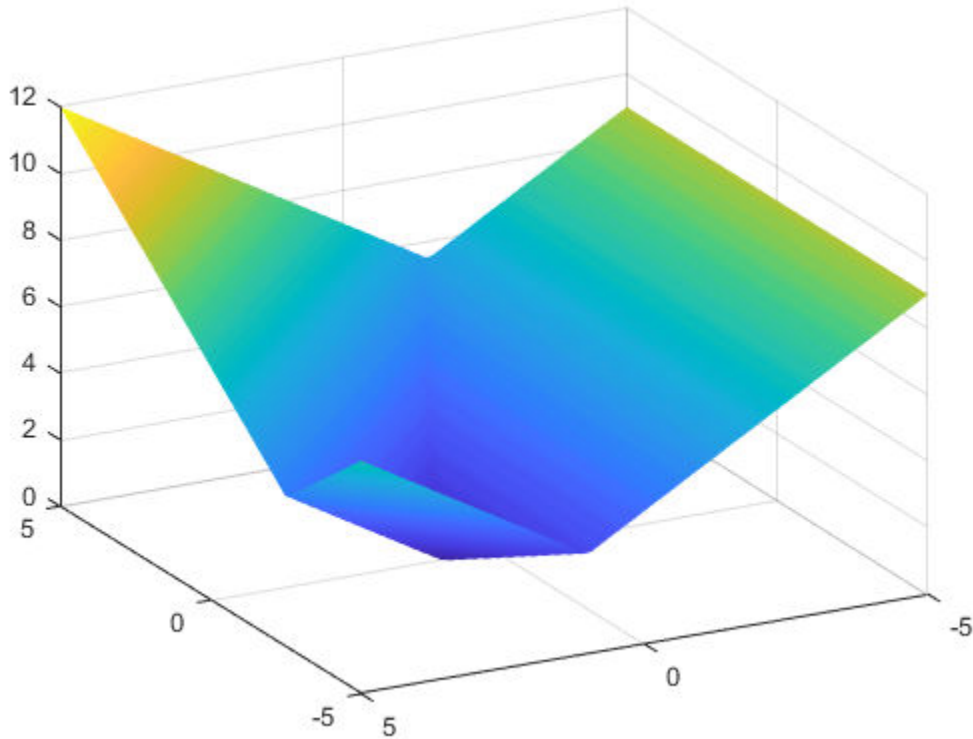
A minimax problem minimizes the maximum of a set of objective functions. Why not minimize this maximum function, which is a scalar function? The answer is that the maximum is not smooth, and Optimization Toolbox™ solvers such as fminunc require smoothness.

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 ob
```

```
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

Another approach to optimizing the control parameters in the Simulink model shown in “Plant with Actuator Saturation” on page 12-21 is to use the `fminimax` function. In this case, rather than minimizing the error between the output and the input signal, you minimize the maximum value of the output at any time  $t$  between 0 and 100.

The code for this example, shown below, is contained in the function `runtrackmm`, in which the objective function is simply the output `yout` returned by the `sim` command. But minimizing the maximum output at all time steps might force the output to be far below unity for some time steps. To keep the output above 0.95 after the first 20 seconds, the constraint function `trackmmcon` contains the constraint `yout >= 0.95` from  $t=20$  to  $t=100$ . Because constraints must be in the form  $g \leq 0$ , the constraint in the function is `g = -yout(20:100)+.95`.

Both `trackmmobj` and `trackmmcon` use the result `yout` from `sim`, calculated from the current PID values. To avoid calling the simulation twice, `runtrackmm` has nested functions so that the value of `yout` is shared between the objective and constraint functions. The simulation is called only when the current point changes.

The following is the code for `runtrackmm`:

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

optsim % initialize Simulink(R)
pid0 = [0.63 0.0504 1.9688];
% a1, a2, yout are shared with TRACKMMOBJ and TRACKMMCON
a1 = 3; a2 = 43; % Initialize plant variables in model
yout = []; % Give yout an initial value
pold = []; % tracks last pid
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.
    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
```

Copy the code for `runtrackmm` to a file named `runtrackmm.m`, placed in a folder on your MATLAB path.

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

```
[Kp,Ki,Kd] = runtrackmm
Done initializing optsim.
```

Iter	F-count	Objective value	Max constraint	Line search steplength	Directional derivative	Procedure
0	5	0	1.11982			
1	11	1.184	0.07978	1	0.482	
2	17	1.012	0.04285	1	-0.236	
3	23	0.9995	0.007058	1	-0.0186	Hessian modified twice
4	29	0.9997	9.707e-07	1	0.00716	Hessian modified

```
Local minimum possible. Constraints satisfied.
```

```
fminimax stopped because the size of the current search direction is less than twice the value of the step size tolerance and constraints are
```

satisfied to within the value of the constraint tolerance.

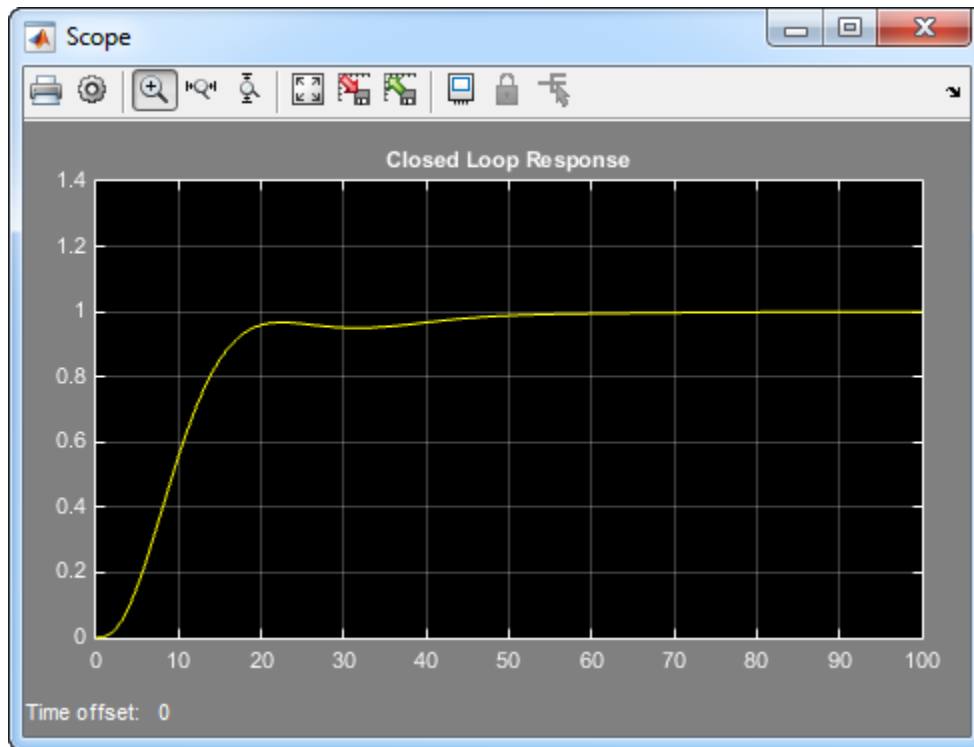
$K_p =$   
0.5910

$K_i =$   
0.0606

$K_d =$   
5.5383

The last value in the **Objective value** column of the output shows that the maximum value for all the time steps is **0.9997**. The closed loop response with this result is shown in the figure “Closed-Loop Response Using fminimax” on page 8-13.

This solution differs from the solution obtained in “lsqnonlin with a Simulink Model” on page 12-21 because you are solving different problem formulations.



**Closed-Loop Response Using fminimax**

## **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

$$\begin{aligned} H(f) &= \sum_{n=0}^{2M} h(n)e^{-j2\pi fn} \\ &= A(f)e^{-j2\pi fM}, \end{aligned} \tag{8-7}$$

$$A(f) = \sum_{n=0}^{M-1} a(n)\cos(2\pi fn),$$

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

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

This information is stored in the variable `goal` passed to `fgoalattain`. The length of `goal` is the same as the length returned by the function `filtmin`. So that the goals are equally satisfied, usually `weight` would be set to `abs(goal)`. However, since some of the goals are zero, the effect of using `weight=abs(goal)` will force the objectives with `weight 0` to be satisfied as hard constraints, and the objectives with `weight 1` possibly to be underattained (see “Goal Attainment Method” on page 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;

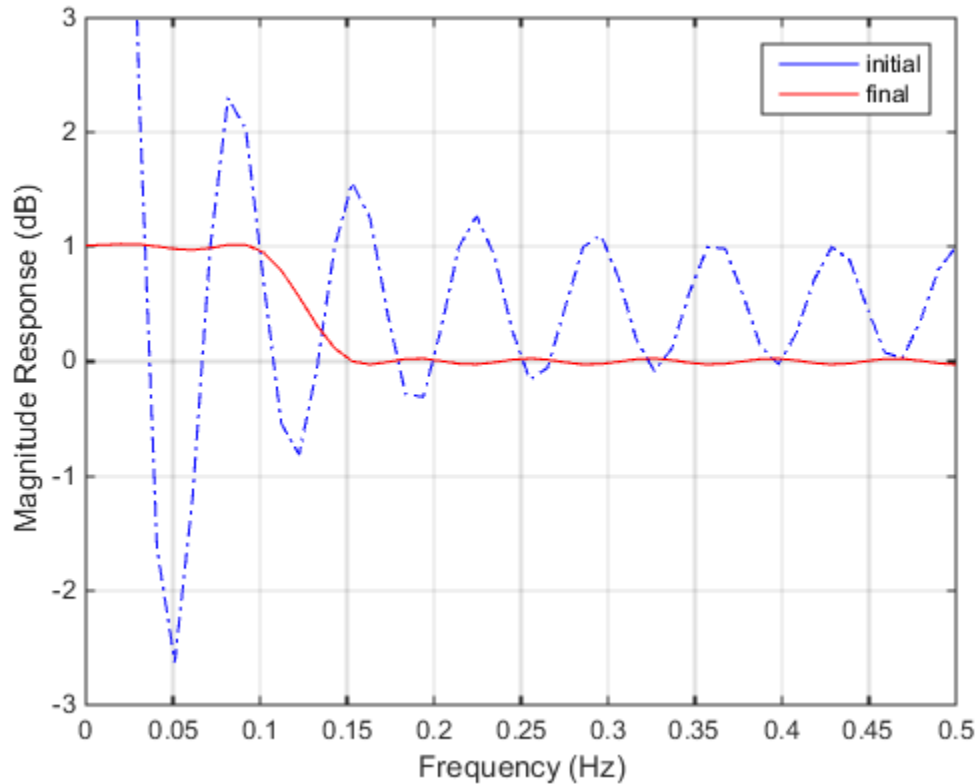
% Set up the goal attainment problem
w1 = linspace(0,0.1,incr) ;
w2 = linspace(0.15,0.5,incr);
w0 = [w1 w2];
goal = [1.0*ones(1,length(w1)) zeros(1,length(w2))];
weight = ones(size(goal));

% 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

8-17). 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 8-23

- “Minimax Optimization” on page 8-31



## Generate and Plot a Pareto Front

This example shows how to generate and plot a Pareto front for a 2-D multiobjective function using `fgoalattain`.

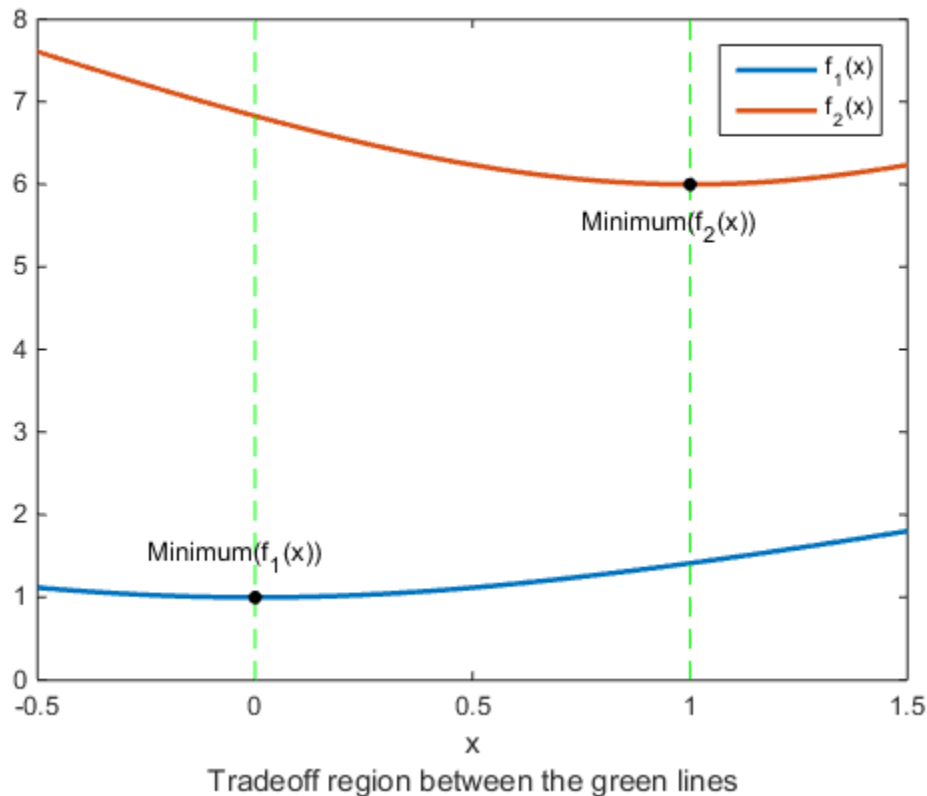
The two objectives in this example are shifted and scaled versions of the convex function  $\sqrt{1+x^2}$ .

```
function f = simple_mult(x)

f(:,1) = sqrt(1+x.^2);
f(:,2) = 4 + 2*sqrt(1+(x-1).^2);
```

Both components are increasing as  $x$  decreases below 0 or increases above 1. In between 0 and 1,  $f_1(x)$  is increasing and  $f_2(x)$  is decreasing, so there is a tradeoff region.

```
t = linspace(-0.5,1.5);
F = simple_mult(t);
plot(t,F,'LineWidth',2)
hold on
plot([0,0],[0,8],'g--');
plot([1,1],[0,8],'g--');
plot([0,1],[1,6],'k.','MarkerSize',15);
text(-0.25,1.5,'Minimum(f_1(x))')
text(.75,5.5,'Minimum(f_2(x))')
hold off
legend('f_1(x)','f_2(x)')
xlabel({'x';'Tradeoff region between the green lines'})
```



To find the Pareto front, first find the unconstrained minima of the two functions. In this case, you can see by inspection that the minimum of  $f_1(x)$  is 1, and the minimum of  $f_2(x)$  is 6, but in general you might need to use an optimization routine.

In general, write a function that returns a particular component of the multiobjective function.

```
function z = pickindex(x,k)
z = simple_mult(x); % evaluate both objectives
z = z(k); % return objective k
```

Then find the minimum of each component using an optimization solver. You can use `fminbnd` in this case, or `fminunc` for higher-dimensional problems.

```

k = 1;
[min1,minfn1] = fminbnd(@(x)pickindex(x,k),-1,2);
k = 2;
[min2,minfn2] = fminbnd(@(x)pickindex(x,k),-1,2);

```

Set goals that are the unconstrained optima for each component. You can simultaneously achieve these goals only if the multiobjective functions do not interfere with each other, meaning there is no tradeoff.

```
goal = [minfn1,minfn2];
```

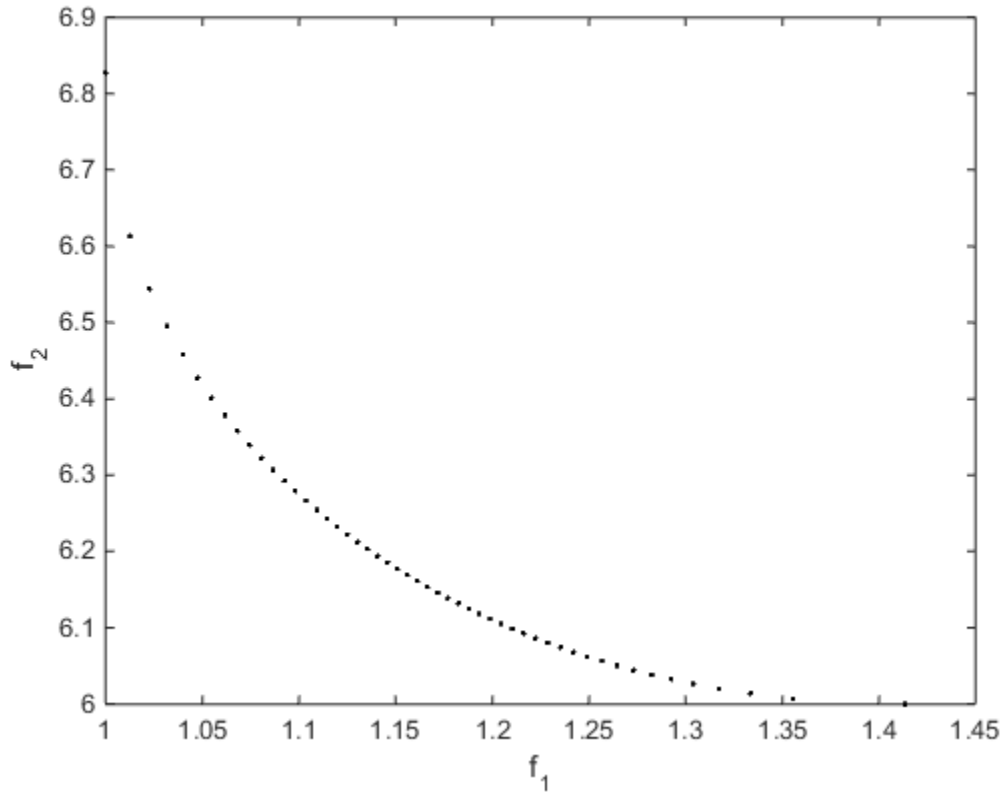
To calculate the Pareto front, take weight vectors  $[a,1-a]$  for  $a$  from 0 through 1. Solve the goal attainment problem, setting the weights to the various values.

```

nf = 2; % number of objective functions
N = 50; % number of points for plotting
onen = 1/N;
x = zeros(N+1,1);
f = zeros(N+1,nf);
fun = @simple_mult;
x0 = 0.5;
options = optimoptions('fgoalattain','Display','off');
for r = 0:N
    t = onen*r; % 0 through 1
    weight = [t,1-t];
    [x(r+1,:),f(r+1,:)] = fgoalattain(fun,x0,goal,weight,...
        [],[],[],[],[],[],[],[],options);
end

figure
plot(f(:,1),f(:,2),'k. ');
xlabel('f_1')
ylabel('f_2')

```



You can see the tradeoff between the two objectives.

## See Also

`fgoalattain`

## More About

- “Multi-Objective Goal Attainment Optimization” on page 8-23

## 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

$$A = \begin{bmatrix} -0.5 & 0 & 0; & 0 & -2 & 10; & 0 & 1 & -2 \end{bmatrix};$$

$$B = \begin{bmatrix} 1 & 0; & -2 & 2; & 0 & 1 \end{bmatrix};$$

$$C = \begin{bmatrix} 1 & 0 & 0; & 0 & 0 & 1 \end{bmatrix};$$

### 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  $(A + B*K*C)$  are smaller than  $[-5, -3, -1]$ . (This is called pole placement in the control literature.)
2.  $\text{abs}(K) \leq 4$  (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 modi
4	34	-0.2236	0.06654	1	-0.234	Hessian modi
5	41	-0.3568	0.007894	1	-0.0812	
6	48	-0.3645	0.000145	1	-0.164	Hessian modi
7	55	-0.3645	0	1	-0.00515	Hessian modi
8	62	-0.3675	0.0001546	1	-0.00812	Hessian modi
9	69	-0.3889	0.008328	1	-0.00751	Hessian modi
10	76	-0.3862	0	1	0.00568	
11	83	-0.3863	8.058e-13	1	-0.998	Hessian modi

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  $x(0) = [1;1;1]$ .

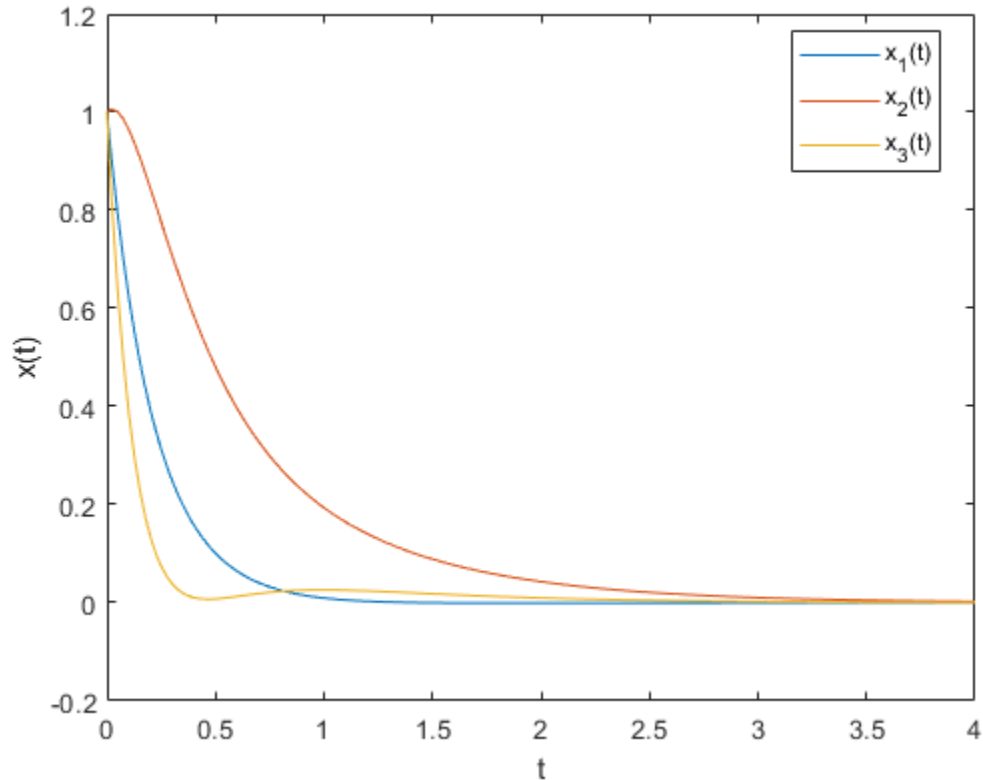
First solve the differential equation:

```
[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.GoalsExactAchieve` 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.GoalsExactAchieve = 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);
```

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	0.1528	-0.009105	1	-0.22	Hessian modi
4	34	0.02684	0.03722	1	-0.166	Hessian modi
5	41	-1.041e-17	0.005702	1	-0.116	Hessian modi
6	48	-1.163e-18	9.705e-06	1	7.98e-16	Hessian modi
7	55	5.209e-21	4.957e-11	1	6.12e-14	Hessian modi

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 = 2x2

```
-1.5954    1.2040
-0.4201   -2.9046
```

This time the eigenvalues of the closed loop system, which are also held in output fval, are as follows:

eigfun(K)

ans = 3x1

```
-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 = 5.2086e-21
```

### **Evolution of New System Via Solution to ODE**

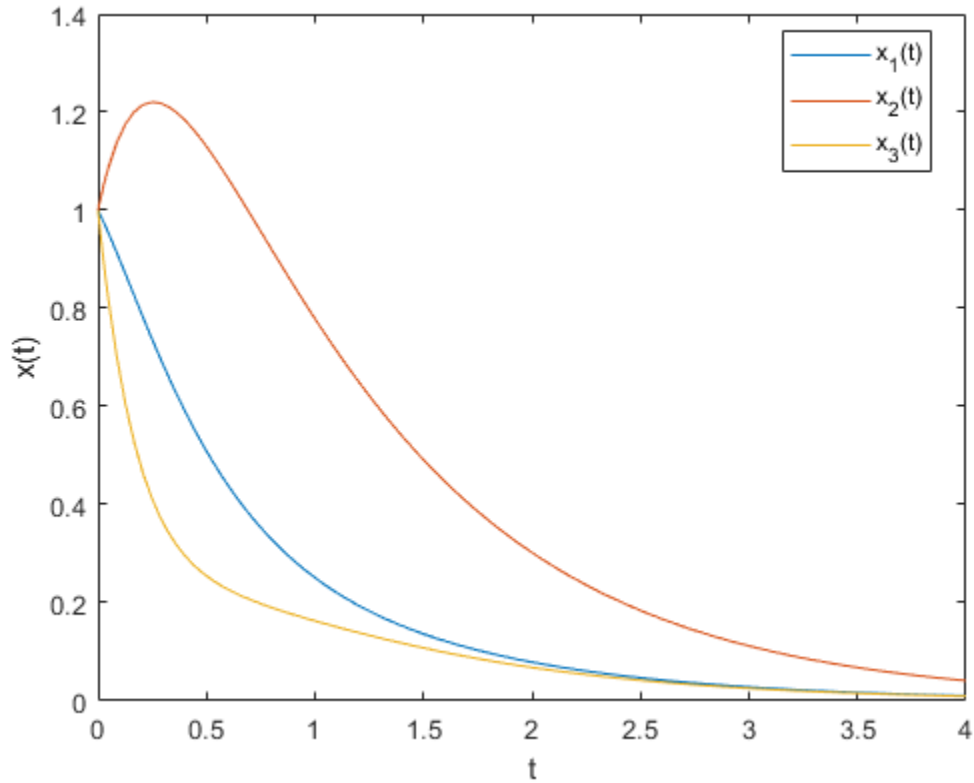
Here is how the system  $x(t)$  evolves from time 0 to time 4, using the new calculated feedback matrix  $K$ , starting from the point  $x(0) = [1;1;1]$ .

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 12-21

## 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.

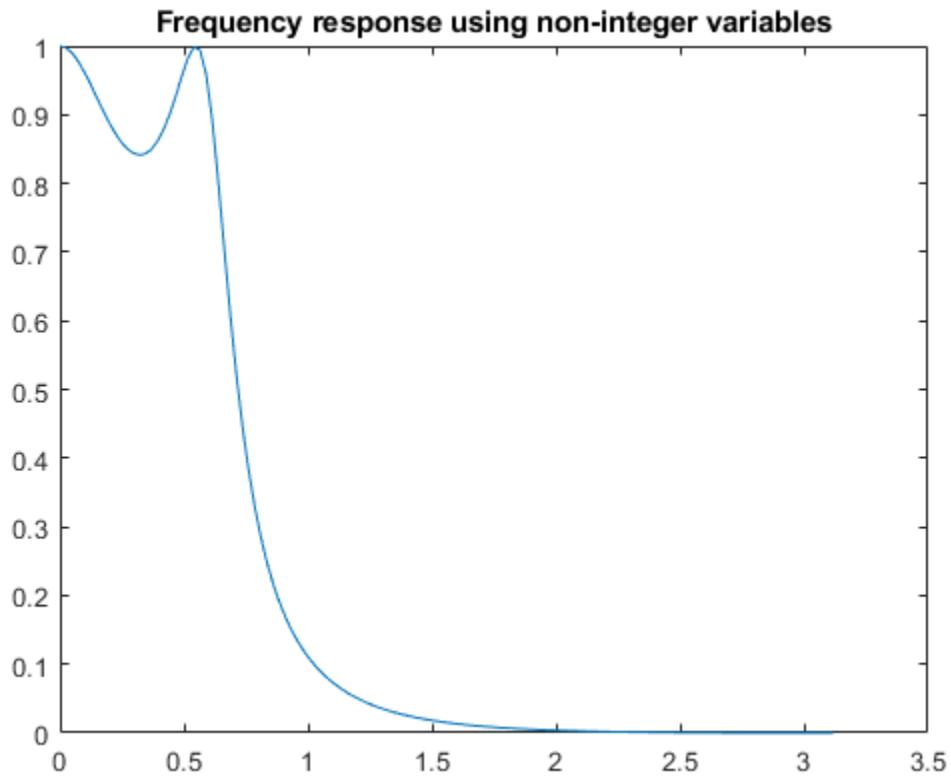
```
nbits = 8;           % How many bits have we to realize filter
maxbin = 2^nbits-1; % Maximum number expressable in nbits bits
n = 4;              % Number of coefficients (order of filter plus 1)
Wn = 0.2;          % Cutoff frequency for filter
Rp = 1.5;          % Decibels of ripple in the passband
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')
```



```
x = [b1,a1];           % The design variables
```

### Set Bounds for Filter Coefficients

We now set bounds on the maximum and minimum values:

```
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.

```

[m, mix] = max(abs(x));
factor = maxbin/m;
x = factor * x;    % Rescale other filter coefficients
xorig = x;

xmask = 1:2*n;
% 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:

```

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:

```

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`:

```

[x, xmask] = elimone(x, xmask, h, w, n, maxbin)

```

```
x = 1x8
    0.5441    1.6323    1.6323    0.5441    57.1653 -127.0000    108.0000   -33.8267
```

```
xmask = 1x6
```

```
    1    2    3    4    5    8
```

```
niters = length(xmask);
disp(sprintf('Performing %g stages of optimization.\n\n', niters));
```

```
Performing 6 stages of optimization.
```

```
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
```

Iter	F-count	Objective value	Max constraint	Line search steplength	Directional 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
```

Iter	F-count	Objective value	Max constraint	Line search steplength	Directional derivative	Procedure
0	7	0	0.0414182			



1	15	0.01649	0.0002558	1	0.261	
2	23	0.01544	6.124e-07	1	-0.0282	Hessian modi

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

Iter	F-count	Objective value	Max constraint	Line search steplength	Directional derivative	Procedure
0	6	0	0.0716961			
1	13	0.05943	3.143e-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

Iter	F-count	Objective value	Max constraint	Line search steplength	Directional derivative	Procedure
0	5	0	0.129938			
1	11	0.04278	7.922e-11	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

Iter	F-count	Objective value	Max constraint	Line search steplength	Directional derivative	Procedure
0	4	0	0.0901749			
1	9	0.03867	-2.598e-11	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

Iter	F-count	Objective value	Max constraint	Line search steplength	Directional derivative	Procedure
0	3	0	0.11283			
1	7	0.05033	-1.249e-16	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:

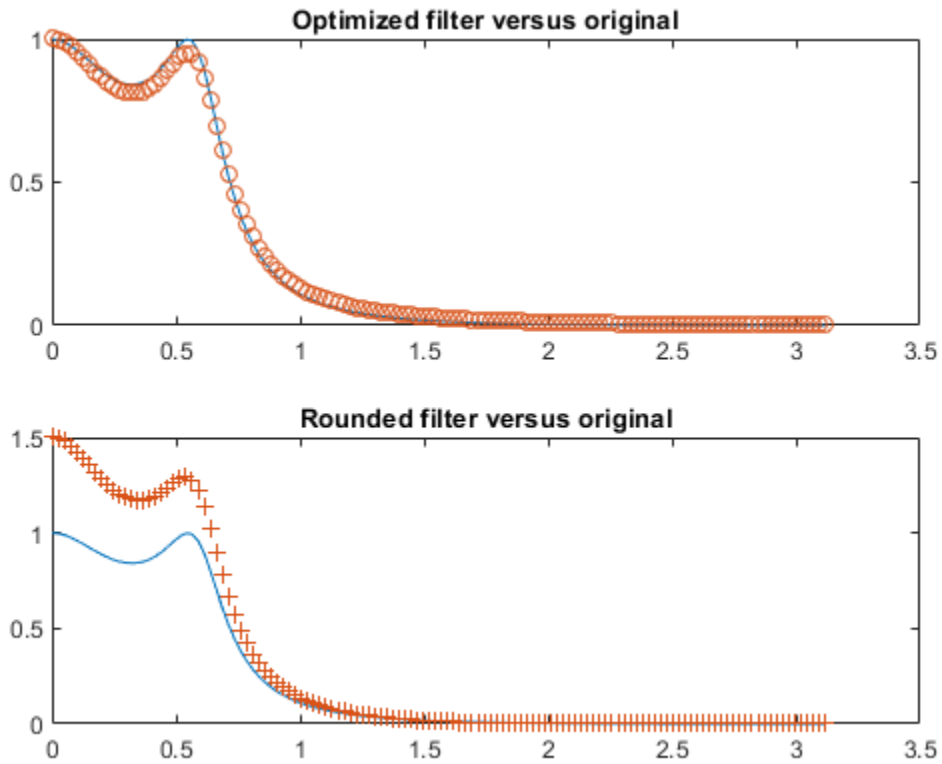
```
subplot(211)
bo = x(1:n);
ao = x(n+1:2*n);
h2 = abs(freqz(bo,ao,128));
plot(w,h,w,h2,'o')
title('Optimized filter versus original')
```

```
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 12-21

# Linear Programming and Mixed-Integer Linear Programming

---

- “Linear Programming Algorithms” on page 9-2
- “Typical Linear Programming Problem” on page 9-16
- “Maximize Long-Term Investments Using Linear Programming: Solver-Based” on page 9-18
- “Mixed-Integer Linear Programming Algorithms” on page 9-33
- “Tuning Integer Linear Programming” on page 9-45
- “Mixed-Integer Linear Programming Basics: Solver-Based” on page 9-48
- “Factory, Warehouse, Sales Allocation Model: Solver-Based” on page 9-52
- “Traveling Salesman Problem: Solver-Based” on page 9-64
- “Optimal Dispatch of Power Generators: Solver-Based” on page 9-73
- “Mixed-Integer Quadratic Programming Portfolio Optimization: Solver-Based” on page 9-86
- “Solve Sudoku Puzzles Via Integer Programming: Solver-Based” on page 9-95
- “Office Assignments by Binary Integer Programming: Solver-Based” on page 9-104
- “Cutting Stock Problem: Solver-Based” on page 9-113
- “Factory, Warehouse, Sales Allocation Model: Problem-Based” on page 9-119
- “Traveling Salesman Problem: Problem-Based” on page 9-129
- “Optimal Dispatch of Power Generators: Problem-Based” on page 9-138
- “Office Assignments by Binary Integer Programming: Problem-Based” on page 9-149
- “Mixed-Integer Quadratic Programming Portfolio Optimization: Problem-Based” on page 9-156
- “Cutting Stock Problem: Problem-Based” on page 9-165
- “Solve Sudoku Puzzles Via Integer Programming: Problem-Based” on page 9-171

## Linear Programming Algorithms

<b>In this section...</b>
---------------------------

<p>“Linear Programming Definition” on page 9-2</p> <p>“Interior-Point linprog Algorithm” on page 9-2</p> <p>“Interior-Point-Legacy Linear Programming” on page 9-8</p> <p>“Dual-Simplex Algorithm” on page 9-11</p>
---

### 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:

$$\begin{array}{rcl}
 A \cdot x & \leq & b \\
 A_{eq} \cdot x & = & b_{eq} \\
 l & \leq & x \leq & u.
 \end{array}$$

### 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 begins by attempting to simplify the problem by removing redundancies and simplifying constraints. The tasks performed during the presolve step include:

- Check if any variables have equal upper and lower bounds. If so, check for feasibility, and then fix and remove the variables.
- Check if any linear inequality constraint involves just one variable. If so, check for feasibility, and change the linear constraint to a bound.
- Check if any linear equality constraint involves just one variable. If so, check for feasibility, and then fix and remove the variable.
- Check if any linear constraint matrix has zero rows. If so, check for feasibility, and delete the rows.
- Check if the bounds and linear constraints are consistent.
- Check if any variables appear only as linear terms in the objective function and do not appear in any linear constraint. If so, check for feasibility and boundedness, and fix the variables at their appropriate bounds.
- Change any linear inequality constraints to linear equality constraints by adding slack variables.

If algorithm detects an infeasible or unbounded problem, it halts and issues an appropriate exit message.

The algorithm might arrive at a single feasible point, which represents the solution.

If the algorithm does not detect an infeasible or unbounded problem in the presolve step, it continues, if necessary, with the other steps. At the end, the algorithm reconstructs the original problem, undoing any presolve transformations. This final step is the postsolve step.

For simplicity, if the problem is not solved in the presolve step, the algorithm shifts all finite lower bounds to zero.

## Generate Initial Point

To set the initial point,  $x_0$ , the algorithm does the following.

- 1 Initialize  $x_0$  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  $x_0(i) = u/2$ .
- 3 For components that have only one bound, modify the component if necessary to lie strictly inside the bound.
- 4 To put  $x_0$  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 6-37, the `interior-point-convex` algorithm tries to find a point where the Karush-Kuhn-Tucker (KKT) on page 3-13 conditions hold. To describe these equations for the linear programming problem, consider the standard form of the linear programming problem after preprocessing:

$$\min_x f^T x \text{ subject to } \begin{cases} \bar{A}x = \bar{b} \\ x + t = u \\ x, t \geq 0. \end{cases}$$

- Assume for now that all variables have at least one finite bound. By shifting and negating components, if necessary, this assumption means that all  $x$  components have a lower bound of 0.
- $\bar{A}$  is the extended linear matrix that includes both linear inequalities and linear equalities.  $\bar{b}$  is the corresponding linear equality vector.  $\bar{A}$  also includes terms for extending the vector  $x$  with slack variables  $s$  that turn inequality constraints to equality constraints:

$$\bar{A}x = \begin{pmatrix} A_{eq} & 0 \\ A & I \end{pmatrix} \begin{pmatrix} x_0 \\ s \end{pmatrix},$$

where  $x_0$  means the original  $x$  vector.

- $t$  is the vector of slacks that convert upper bounds to equalities.

The Lagrangian for this system involves the following vectors:

- $y$ , Lagrange multipliers associated with the linear equalities
- $v$ , Lagrange multipliers associated with the lower bound (positivity constraint)
- $w$ , Lagrange multipliers associated with the upper bound



The Lagrangian is

$$L = f^T x - y^T (\bar{A}x - \bar{b}) - v^T x - w^T (u - x - t).$$

Therefore, the KKT conditions for this system are

$$f - \bar{A}^T y - v + w = 0$$

$$\bar{A}x = \bar{b}$$

$$x + t = u$$

$$v_i x_i = 0$$

$$w_i t_i = 0$$

$$(x, v, w, t) \geq 0.$$

The algorithm first predicts a step from the Newton-Raphson formula, and then computes a corrector step. The corrector attempts to reduce the residual in the nonlinear complementarity equations  $s_i z_i = 0$ . The Newton-Raphson step is

$$\begin{pmatrix} 0 & -\bar{A}^T & 0 & -I & I \\ \bar{A} & 0 & 0 & 0 & 0 \\ -I & 0 & -I & 0 & 0 \\ V & 0 & 0 & X & 0 \\ 0 & 0 & W & 0 & T \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta t \\ \Delta v \\ \Delta w \end{pmatrix} = - \begin{pmatrix} f - \bar{A}^T y - v + w \\ \bar{A}x - \bar{b} \\ u - x - t \\ VX \\ WT \end{pmatrix} = - \begin{pmatrix} r_d \\ r_p \\ r_{ub} \\ r_{vx} \\ r_{wt} \end{pmatrix}, \quad (9-1)$$

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:

$$\text{Primal infeasibility} = \|r_p\|_1 + \|r_{ub}\|_1$$

$$\text{Dual infeasibility} = \|r_d\|_\infty.$$

To solve “Equation 9-1”, first convert it to the symmetric matrix form

$$\begin{pmatrix} -D & \bar{A}^T \\ \bar{A} & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = - \begin{pmatrix} R \\ r_p \end{pmatrix}, \quad (9-2)$$

where

$$\begin{aligned} D &= X^{-1}V + T^{-1}W \\ R &= -r_d - X^{-1}r_{vx} + T^{-1}r_{wt} + T^{-1}Wr_{ub}. \end{aligned}$$

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  $\Delta v$  and  $\Delta w$ , and then solving for  $\Delta t$ .

“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

$$\bar{A}D^{-1}\bar{A}^T\Delta y = -\bar{A}D^{-1}R - r_p. \quad (9-3)$$

Usually, the most efficient way to find the Newton step is to solve “Equation 9-3” for  $\Delta y$  using Cholesky factorization. Cholesky factorization is possible because the matrix multiplying  $\Delta y$  is obviously symmetric and, in the absence of degeneracies, is positive definite. Afterward, to find the Newton step, back substitute to find  $\Delta x$ ,  $\Delta t$ ,  $\Delta v$ , and  $\Delta w$ . 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 11-6.

### 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

$$\rho = \max(1, \|\bar{A}\|, \|f\|, \|\bar{b}\|).$$

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.

## 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 9-10). After preprocessing, the problem has the form

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x = b \\ 0 \leq x \leq u. \end{cases} \quad (9-4)$$

The upper bounds constraints are implicitly included in the constraint matrix  $A$ . With the addition of primal slack variables  $s$ , "Equation 9-4" becomes

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x = b \\ x + s = u \\ x \geq 0, s \geq 0. \end{cases} \quad (9-5)$$

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

$$\max_b^T y - u^T w \text{ such that } \begin{cases} A^T \cdot y - w + z = f \\ z \geq 0, w \geq 0, \end{cases} \quad (9-6)$$

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

$$F(x, y, z, s, w) = \begin{pmatrix} A \cdot x - b \\ x + s - u \\ A^T \cdot y - w + z - f \\ x_i z_i \\ s_i w_i \end{pmatrix} = 0, \quad (9-7)$$

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

where  $x_i z_i$  and  $s_i w_i$  denote component-wise multiplication.

The quadratic equations  $x_i z_i = 0$  and  $s_i w_i = 0$  are called the *complementarity* conditions for the linear program; the other (linear) equations are called the *feasibility* conditions. The quantity

$$x^T z \qquad \qquad \qquad + \qquad \qquad \qquad s^T w$$

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

$$\Delta v_c = - \left( F^T(v) \right)^{-1} F(v + \Delta v_p) - \mu \hat{e},$$

where  $\mu > 0$  is called the *centering* parameter and must be chosen carefully.  $\hat{e}$  is a zero-one vector with the ones corresponding to the quadratic equations in  $F(v)$ , i.e., the perturbations are only applied to the complementarity conditions, which are all quadratic, but not to the feasibility conditions, which are all linear. The two directions are combined with a step length parameter  $\alpha > 0$  and update  $v$  to obtain the new iterate  $v^+$ :

$$v^+ = v + \alpha(\Delta v_p + \Delta v_c),$$

where the step length parameter  $\alpha$  is chosen so that

$$v^+ \qquad \qquad \qquad = \qquad \qquad \qquad [x^+; y^+; z^+; s^+; w^+]$$

satisfies

$$[x^+; z^+; s^+; w^+] \quad > \quad 0.$$

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 (MATLAB) in the MATLAB `ldl` function reference page.)

The algorithm then loops until the iterates converge. The main stopping criteria is a standard one:

$$\max \left( \frac{\|r_b\|}{\max(1, \|b\|)}, \frac{\|r_f\|}{\max(1, \|f\|)}, \frac{\|r_u\|}{\max(1, \|u\|)}, \frac{|f^T x - b^T y + u^T w|}{\max(1, |f^T x|, |b^T y - u^T w|)} \right) \leq tol,$$

where

$$\begin{aligned} r_b &= Ax - b \\ r_f &= A^T y - w + z - f \\ r_u &= \{x\} + s - u \end{aligned}$$

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 begins by attempting to simplify the problem by removing redundancies and simplifying constraints. The tasks performed during the presolve step include:

- Check if any variables have equal upper and lower bounds. If so, check for feasibility, and then fix and remove the variables.

- Check if any linear inequality constraint involves just one variable. If so, check for feasibility, and change the linear constraint to a bound.
- Check if any linear equality constraint involves just one variable. If so, check for feasibility, and then fix and remove the variable.
- Check if any linear constraint matrix has zero rows. If so, check for feasibility, and delete the rows.
- Check if the bounds and linear constraints are consistent.
- Check if any variables appear only as linear terms in the objective function and do not appear in any linear constraint. If so, check for feasibility and boundedness, and fix the variables at their appropriate bounds.
- Change any linear inequality constraints to linear equality constraints by adding slack variables.

If algorithm detects an infeasible or unbounded problem, it halts and issues an appropriate exit message.

The algorithm might arrive at a single feasible point, which represents the solution.

If the algorithm does not detect an infeasible or unbounded problem in the presolve step, it continues, if necessary, with the other steps. At the end, the algorithm reconstructs the original problem, undoing any presolve transformations. This final step is the postsolve step.

For simplicity, the algorithm shifts all lower bounds to zero.

While these preprocessing steps can do much to speed up the iterative part of the algorithm, if the Lagrange multipliers are required, the preprocessing steps must be undone since the multipliers calculated during the algorithm are for the transformed problem, and not the original. Thus, if the multipliers are *not* requested, this transformation back is not computed, and might save some time computationally.

## 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 9-10. 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”:

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x = b \\ 0 \leq x \leq u. \end{cases}$$

$A$  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

$$x^+ = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0. \end{cases}$$

The measure of primal infeasibility is

$$\text{Primal infeasibility} = \sqrt{((lb-x)^+)^2 + ((x-ub)^+)^2 + ((Ax-b)^+)^2 + |Aeqx - beq|^2}.$$

As explained in "Equation 9-6", the dual problem is to find vectors  $y$  and  $w$ , and a slack variable vector  $z$  that solve

$$\max b^T y - u^T w \text{ such that } \begin{cases} A^T \cdot y - w + z = f \\ z \geq 0, w \geq 0. \end{cases}$$

The measure of dual infeasibility is

$$\text{Dual infeasibility} = \|A^T y + z - w - f\|_2.$$

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 9-14 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

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

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

$$\min_x \sum_j P_j \quad \text{subject to} \quad \begin{cases} A \cdot x \leq b \\ Aeq \cdot x = beq. \end{cases}$$

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 9-14 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  $\{a_1, a_2, \dots, a_n\}$ . Suppose that  $\{a_{i_1}, a_{i_2}, \dots, a_{i_m}\}$  is a basis for the column space of  $A$ , with index set  $B = \{i_1, i_2, \dots, i_m\}$ , and that  $N = \{1, 2, \dots, n\} \setminus B$  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 <http://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 shows the solution of a typical linear programming problem. The problem is

$$\min_x f^T x \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ x \geq 0. \end{cases}$$

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

```
load sc50b
```

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

Use `linprog` to solve the problem:

```
options = optimoptions(@linprog,'Algorithm','dual-simplex','Display','iter');  
[x,fval,exitflag,output] = ...  
    linprog(f,A,b,Aeq,beq,lb,[],options);
```

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

LP preprocessing removed 2 inequalities, 16 equalities, 16 variables, and 26 non-zero elements.

Iter	Time	Fval	Primal Infeas	Dual Infeas
0	0.001	0.000000e+00	0.000000e+00	1.305013e+00
8	0.001	-1.587073e+02	3.760622e+02	0.000000e+00
33	0.001	-7.000000e+01	0.000000e+00	0.000000e+00

Optimal solution found.

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

```
exitflag,fval,output
```

```
exitflag =
```

```
1
```

```
fval =
```

```
    -70
```

```
output =
```

```
  struct with fields:
```

```
    iterations: 33  
  constrviolation: 3.1264e-13  
    message: 'Optimal solution found.'  
    algorithm: 'dual-simplex'  
  firstorderopt: 2.3082e-14
```

## 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  $k=1 \dots 9$

- 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  $\rho_k$  of bond  $k$ .

Visualize this problem by horizontal bars that represent the available purchase times and durations for each bond.

```
% Time period in years
T = 5;
% Number of bonds
N = 4;
% Initial amount of money
Capital_0 = 1000;
% Total number of buying oportunities
nPtotal = N+T;
% Purchase times
PurchaseYears = [1;2;3;4;5;1;5;2;2];
% Bond durations
Maturity = [1;1;1;1;1;4;1;4;3];
% Interest rates
InterestRates = [0;0;0;0;0;2;4;6;6];

plotInvestments(N,PurchaseYears,Maturity,InterestRates)
```

	Year 1	Year 2	Year 3	Year 4	Year 5
B <sub>0</sub> 0%					
B <sub>1</sub> 2%					
B <sub>2</sub> 4%					
B <sub>3</sub> 6%					
B <sub>4</sub> 6%					

### Decision Variables

Represent your decision variables by a vector  $x$ , where  $x(k)$  is the dollar amount of investment in bond  $k$ , for  $k=1 \dots 9$ . Upon maturity, the payout for investment  $x(k)$  is

$$x(k)(1 + \rho_k/100)^{m_k}.$$

Define  $r_k$  as the total return of bond  $k$ :

$$r_k = (1 + \rho_k/100)^{m_k}.$$

`% Total returns`

```
finalReturns = (1+InterestRates/100).^Maturity;
```

### Objective Function

The goal is to choose investments to maximize the amount of money collected at the end of year  $T$ . From the plot, you see that investments are collected at various intermediate years and reinvested. At the end of year  $T$ , the money returned from investments 5, 7, and 8 can be collected and represents your final wealth:

$$\max_x x_5 r_5 + x_7 r_7 + x_8 r_8$$

To place this problem into the form `linprog` solves, turn this maximization problem into a minimization problem using the negative of the coefficients of  $x(j)$ :



$$\min_x f^T x$$

with

$$f = [0; 0; 0; 0; -r_5; 0; -r_7; -r_8; 0]$$

```
f = zeros(nPtotal,1);
f([5,7,8]) = [-finalReturns(5), -finalReturns(7), -finalReturns(8)];
```

**Linear Constraints: Invest No More Than You Have**

Every year, you have a certain amount of money available to purchase bonds. Starting with year 1, you can invest the initial capital in the purchase options  $x_1$  and  $x_6$ , so:

$$x_1 + x_6 = \text{Capital}_0$$

Then for the following years, you collect the returns from maturing bonds, and reinvest them in new available bonds to obtain the system of equations:

$$\begin{aligned} x_2 + x_8 + x_9 &= r_1 x_1 \\ x_3 &= r_2 x_2 \\ x_4 &= r_3 x_3 \\ x_5 + x_7 &= r_4 x_4 + r_6 x_6 + r_9 x_9 \end{aligned}$$

Write these equations in the form  $Aeqx = beq$ , 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)
```

	Year 1	Year 2	Year 3	Year 4	Year 5
B <sub>0</sub> 0%					
B <sub>1</sub> 2%					
B <sub>2</sub> 4%					
B <sub>3</sub> 6%					
B <sub>4</sub> 6%					

### 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  $P_{\max}$  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:

$$x_1 \leq P_{\max} \times \text{Capital}_0$$

$$x_2 \leq P_{\max} \times (\rho_1 x_1 + \rho_6 x_6)$$

$$x_3 \leq P_{\max} \times (\rho_2 x_2 + \rho_6^2 x_6 + \rho_8 x_8 + \rho_9 x_9)$$

$$x_4 \leq P_{\max} \times (\rho_3 x_3 + \rho_6^3 x_6 + \rho_8^2 x_8 + \rho_9^2 x_9)$$

$$x_5 \leq P_{\max} \times (\rho_4 x_4 + \rho_6^4 x_6 + \rho_8^3 x_8 + \rho_9^3 x_9)$$

$$x_6 \leq P_{\max} \times \text{Capital}_0$$

$$x_7 \leq P_{\max} \times (\rho_4 x_4 + \rho_6^4 x_6 + \rho_8^3 x_8 + \rho_9^3 x_9)$$

$$x_8 \leq P_{\max} \times (\rho_1 x_1 + \rho_6 x_6)$$

$$x_9 \leq P_{\max} \times (\rho_1 x_1 + \rho_6 x_6)$$

Place these inequalities in the matrix form  $Ax \leq b$ .

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)
```

	Year 1	Year 2	Year 3	Year 4	Year 5
B <sub>0</sub> 0%					
B <sub>1</sub> 2%					
B <sub>2</sub> 4%					
B <sub>3</sub> 6%					
B <sub>4</sub> 6%					

### 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  $A_{eq}$  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 opportunities
nPtotal = N+T;
% Generate random maturity durations
Maturity = randi([1 T-1],nPtotal,1);
% Bond 1 has a maturity period of 1 year
Maturity(1:T) = 1;
% Generate random yearly interest rate for each bond
InterestRates = randi(6,nPtotal,1);
% Bond 1 has an interest rate of 0 (not invested)
```

```
InterestRates(1:T) = 0;
% Compute the return at the end of the maturity period for each bond:
finalReturns = (1+InterestRates/100).^Maturity;

% Generate random purchase years for each option
PurchaseYears = zeros(nPtotal,1);
% Bond 1 is available for purchase every year
PurchaseYears(1:T)=1:T;
for i=1:N
    % Generate a random year for the bond to mature before the end of
    % the T year period
    PurchaseYears(i+T) = randi([1 T-Maturity(i+T)+1]);
end

% Compute the years where each bond reaches maturity
SaleYears = PurchaseYears + Maturity;

% Initialize f to 0
f = zeros(nPtotal,1);
% Indices of the sale oportunities at the end of year T
SalesTidx = SaleYears==T+1;
% Expected return for the sale oportunities at the end of year T
ReturnsT = finalReturns(SalesTidx);
% Objective function
f(SalesTidx) = -ReturnsT;

% Generate the system of equality constraints.
% For each purchase option, put a coefficient of 1 in the row corresponding
% to the year for the purchase option and the column corresponding to the
% index of the purchase oportunity
xeq1 = PurchaseYears;
yeq1 = (1:nPtotal)';
ceq1 = ones(nPtotal,1);

% For each sale option, put  $-\rho_k$ , where  $\rho_k$  is the interest rate for the
% associated bond that is being sold, in the row corresponding to the
% year for the sale option and the column corresponding to the purchase
% oportunity
xeq2 = SaleYears(~SalesTidx);
yeq2 = find(~SalesTidx);
ceq2 = -finalReturns(~SalesTidx);

% Generate the sparse equality matrix
```

```

Aeq = sparse([xeq1; xeq2], [yeq1; yeq2], [ceq1; ceq2], T, nPtotal);

% Generate the right-hand side
beq = zeros(T,1);
beq(1) = Capital_0;

% Build the system of inequality constraints
% Maximum percentage to invest in any bond
Pmax = 0.4;

% Build the returns for each bond over the maturity period
cumMaturity = [0;cumsum(Maturity)];
xr = zeros(cumMaturity(end-1),1);
yr = zeros(cumMaturity(end-1),1);
cr = zeros(cumMaturity(end-1),1);
for i = 1:nPtotal
    mi = Maturity(i); % maturity of bond i
    pi = PurchaseYears(i); % purchase year of bond i
    idx = cumMaturity(i)+1:cumMaturity(i+1); % index into xr, yr and cr
    xr(idx) = i; % bond index
    yr(idx) = pi+1:pi+mi; % maturing years
    cr(idx) = (1+InterestRates(i)/100).^(1:mi); % returns over the maturity period
end
yearlyReturns = sparse(xr,yr,cr,nPtotal,T+1);

% Build the system of inequality constraints
A = -Pmax*yearlyReturns(:,PurchaseYears)' + speye(nPtotal);

% Left-hand side
b = zeros(nPtotal,1);
b(PurchaseYears==1) = Pmax*Capital_0;

% Add the lower-bound constraints to the problem.
lb = zeros(nPtotal,1);

```

### Solution with No Holding Limit

First, solve the linear programming problem without inequality constraints using the interior-point algorithm.

```

% Solve the problem without inequality constraints
options = optimoptions('linprog','Algorithm','interior-point');
tic
[xsol,fval,exitflag] = linprog(f,[],[],Aeq,beq,lb,[],options);

```

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 0.037342 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 1.156802 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

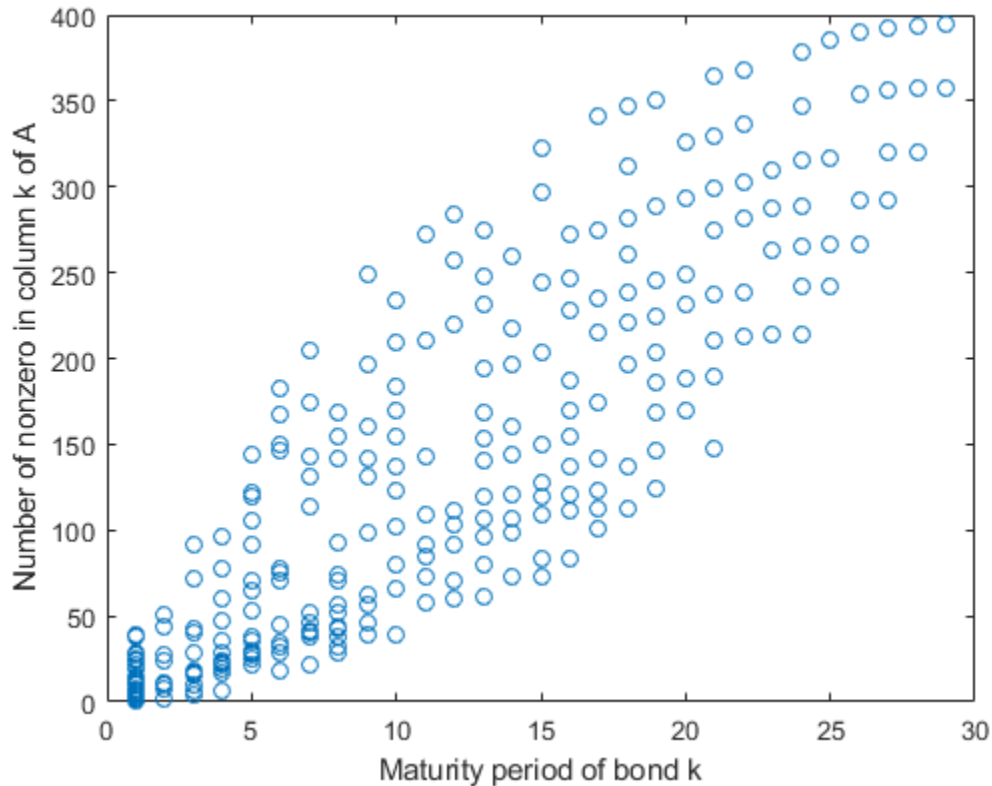
Even though the number of constraints increased by an order of 10, the time for the solver to find a solution increased by an order of 100. This performance discrepancy is partially caused by dense columns in the inequality system shown in matrix A. These



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 0.227974 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

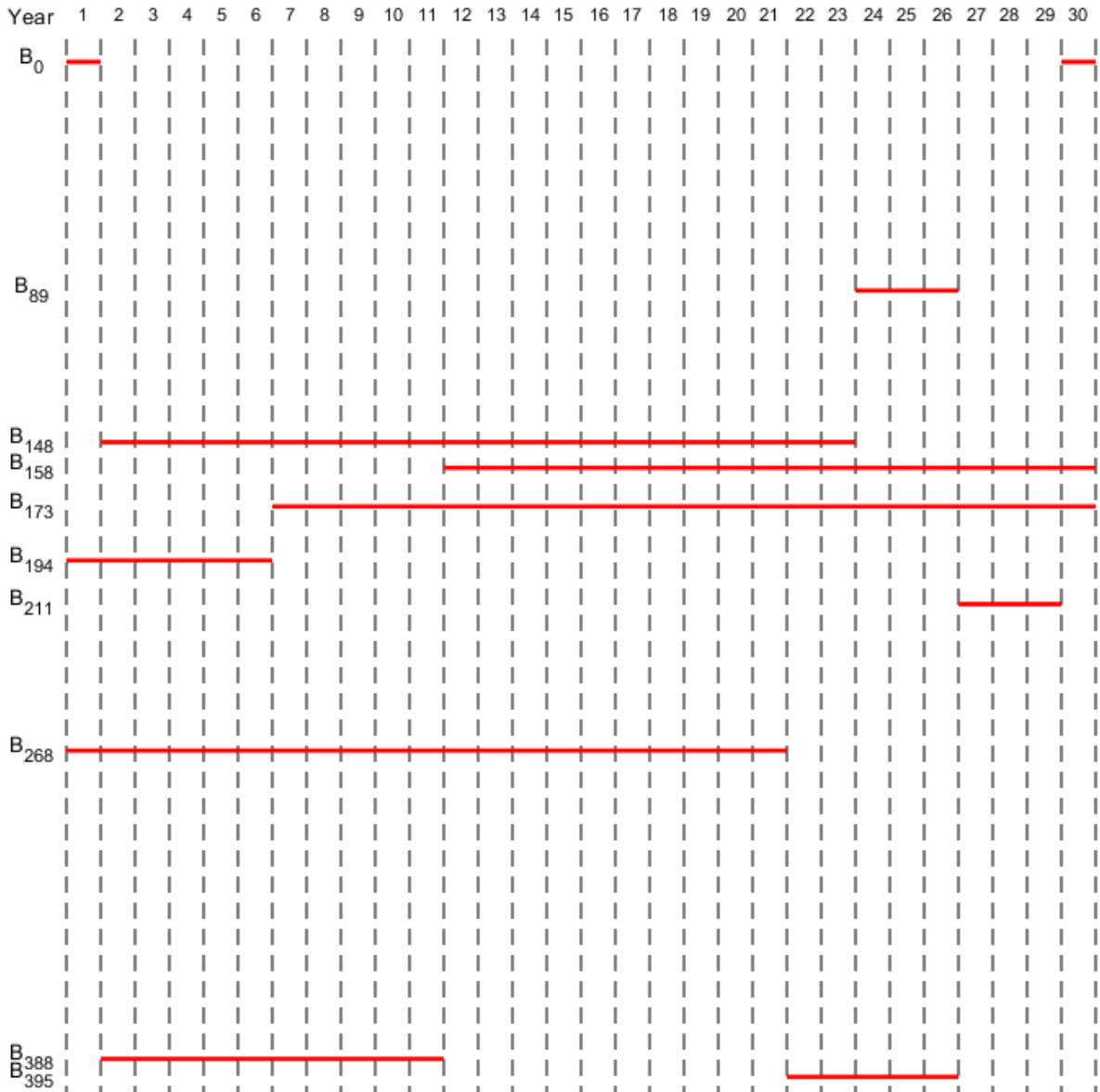
To get a feel for the solution found by `linprog`, compare it to the amount `fmax` that you would get if you could invest all of your starting money in one bond with a 6% interest rate (the maximum interest rate) over the full 30 year period. You can also compute the equivalent interest rate corresponding 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 if you invested all of your starting money in one bond with a 6% interest rate over the 30 year period. Your final wealth corresponds to a 5.57771% interest rate over the 30 year period.

```
plotInvestments(N,PurchaseYears,Maturity,InterestRates,xsol,false)
```



## See Also

9-32

### More About

- “Maximize Long-Term Investments Using Linear Programming: Problem-Based”

## Mixed-Integer Linear Programming Algorithms

### In this section...

“Mixed-Integer Linear Programming Definition” on page 9-33

“intlinprog Algorithm” on page 9-33

### Mixed-Integer Linear Programming Definition

A mixed-integer linear program (MILP) is a problem with

- Linear objective function,  $f^T x$ , where  $f$  is a column vector of constants, and  $x$  is the column vector of unknowns
- Bounds and linear constraints, but no nonlinear constraints (for definitions, see “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

$$\min_x f^T x \text{ subject to } \begin{cases} x(intcon) \text{ are integers} \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub. \end{cases}$$

### intlinprog Algorithm

- “Algorithm Overview” on page 9-34
- “Linear Program Preprocessing” on page 9-34
- “Linear Programming” on page 9-35
- “Mixed-Integer Program Preprocessing” on page 9-35
- “Cut Generation” on page 9-35
- “Heuristics for Finding Feasible Solutions” on page 9-36
- “Branch and Bound” on page 9-40

### 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 9-34.
- 2 Solve an initial relaxed (noninteger) problem using “Linear Programming” on page 9-35.
- 3 Perform “Mixed-Integer Program Preprocessing” on page 9-35 to tighten the LP relaxation of the mixed-integer problem.
- 4 Try “Cut Generation” on page 9-35 to further tighten the LP relaxation of the mixed-integer problem.
- 5 Try to find integer-feasible solutions using heuristics on page 9-36.
- 6 Use a “Branch and Bound” on page 9-40 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 9-33, 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 [7].

### 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 9-33, 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  $A*x \leq b$  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 [9].

### 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.

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'	<code>intlinprog</code> 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'.

Option	Description
'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'.
'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.

Option	Description
'diving'	<p><code>intlinprog</code> 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, <code>intlinprog</code> uses six diving heuristics in this order:</p> <ul style="list-style-type: none"> <li>• Vector length diving</li> <li>• Coefficient diving</li> <li>• Fractional diving</li> <li>• Pseudo cost diving</li> <li>• Line search diving</li> <li>• Guided diving (applies when the solver already found at least one integer-feasible point)</li> </ul> <p>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.</p>
'none'	<p><code>intlinprog</code> does not search for a feasible point. The solver simply takes any feasible point it encounters in its branch-and-bound search.</p>

The main difference between 'intermediate' and 'advanced' is that 'advanced' runs heuristics more frequently during branch-and-bound iterations.

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 15-48.

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  $f^T x$ . The first upper bound is any feasible solution, and the first lower bound is the solution to the relaxed problem. For a discussion of the upper bound, see “Heuristics for Finding Feasible Solutions” on page 9-36.

As explained in “Linear Programming” on page 9-35, any solution to the linear programming relaxed problem has a lower objective function value than the solution to the MILP. Also, any feasible point  $x_{\text{feas}}$  satisfies

$$f^T x_{\text{feas}} \geq f^T x,$$

because  $f^T x$  is the minimum among all feasible points.

In this context, a node is an LP with the same objective function, bounds, and linear constraints as the original problem, but without integer constraints, and with particular changes to the linear constraints or bounds. The root node is the original problem with no integer constraints and no changes to the linear constraints or bounds, meaning the root node is the initial relaxed LP.

From the starting bounds, the branch-and-bound method constructs new subproblems by branching from the root node. The branching step is taken heuristically, according to one of several rules. Each rule is based on the idea of splitting a problem by restricting one variable to be less than or equal to an integer  $J$ , or greater than or equal to  $J+1$ . These two subproblems arise when an entry in  $x_{\text{LP}}$ , corresponding to an integer specified in `intcon`, is not an integer. Here,  $x_{\text{LP}}$  is the solution to a relaxed problem. Take  $J$  as the floor of the variable (rounded down), and  $J+1$  as the ceiling (rounded up). The resulting two problems have solutions that are larger than or equal to  $f^T x_{\text{LP}}$ , because they have more restrictions. Therefore, this procedure potentially raises the lower bound.

The performance of the branch-and-bound method depends on the rule for choosing which variable to split (the branching rule). The algorithm uses these rules, which you can set in the `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:

$$\begin{aligned} p_i^- &= x(i) - \lfloor x(i) \rfloor \\ p_i^+ &= 1 - p_i^- \end{aligned}$$

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

$$\Delta_i^- = f^T x_i^- - f^T x.$$

Similarly,  $\Delta_i^+$  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  $k_2$  times for each variable, where  $k_2$  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

$$\begin{array}{rclcl} p_i^- & = & x(i) & - & [x(i)] \\ p_i^+ & = & 1 & - & p_i^- \end{array}$$

- 'simplebestproj' — Choose the node with the best projection.

### Best Projection

Let  $x_B$  denote the best integer-feasible point found so far,  $x_R$  denote 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 [8] and Wolsey [10].

## 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] Mészáros C., and Suhl, U. H. *Advanced preprocessing techniques for linear and quadratic programming*. OR Spectrum, 25(4), pp. 575-595, 2003.
- [8] Nemhauser, G. L. and Wolsey, L. A. *Integer and Combinatorial Optimization*. Wiley-Interscience, New York, 1999.
- [9] Savelsbergh, M. W. P. *Preprocessing and Probing Techniques for Mixed Integer Programming Problems*. ORSA J. Computing, Vol. 6, No. 4, pp. 445-454, 1994.
- [10] 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 9-45

“Some “Integer” Solutions Are Not Integers” on page 9-46

“Large Components Not Integer Valued” on page 9-46

“Large Coefficients Disallowed” on page 9-47

### 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 program. The example is not complex, but it shows typical steps in formulating a problem in the syntax for `intlinprog`.

For the problem-based approach to this problem, see “Mixed-Integer Linear Programming Basics: Problem-Based” on page 10-45.

### 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  $25 \text{ tons} \times 5\% = 1.25 \text{ tons}$  of carbon and 1.25 tons of molybdenum. The objective is to minimize the cost for blending the steel.

This problem is taken from Carl-Henrik Westerberg, Bengt Bjorklund, and Eskil Hultman, “An Application of Mixed Integer Programming in a Swedish Steel Mill.” *Interfaces* February 1977 Vol. 7, No. 2 pp. 39–43, whose abstract is at <http://interfaces.journal.informs.org/content/7/2/39.abstract>.

Four ingots of steel are available for purchase. Only one of each ingot is available.

Ingot	Weight in Tons	%Carbon	%Molybdenum	Cost/Ton
1	5	5	3	\$350
2	3	4	3	\$330
3	4	5	4	\$310
4	6	3	4	\$280

Three grades of alloy steel are available for purchase, and one grade of scrap steel. Alloy and scrap steels can be purchased in fractional amounts.

Alloy	%Carbon	%Molybdenum	Cost/Ton
1	8	6	\$500
2	7	7	\$450
3	6	8	\$400

Alloy	%Carbon	%Molybdenum	Cost/Ton
Scrap	3	9	\$100

To formulate the problem, first decide on the control variables. Take variable  $x(1) = 1$  to mean you purchase ingot **1**, and  $x(1) = 0$  to mean you do not purchase the ingot. Similarly, variables  $x(2)$  through  $x(4)$  are binary variables indicating that you purchase ingots **2** through **4**.

Variables  $x(5)$  through  $x(7)$  are the quantities in tons of alloys **1**, **2**, and **3** you purchase, and  $x(8)$  is the quantity of scrap steel you purchase.

### MATLAB formulation

Formulate the problem by specifying the inputs for `intlinprog`. The relevant `intlinprog` syntax is as follows.

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

Create the inputs for `intlinprog` from first (`f`) through last (`ub`).

`f` is the vector of cost coefficients. The coefficients representing the costs of ingots are the ingot weights times their cost per ton.

```
f = [350*5,330*3,310*4,280*6,500,450,400,100];
```

The integer variables are the first four.

```
intcon = 1:4;
```

---

**Tip** To specify binary variables, set the variables to be integers in `intcon`, and give them a lower bound of 0 and an upper bound of 1.

---

There are no linear inequality constraints, so `A` and `b` are empty matrices (`[]`).

There are three equality constraints. The first is that the total weight is 25 tons.

$$5*x(1) + 3*x(2) + 4*x(3) + 6*x(4) + x(5) + x(6) + x(7) + x(8) = 25.$$

The second constraint is that the weight of carbon is 5% of 25 tons, or 1.25 tons.

$$5*0.05*x(1) + 3*0.04*x(2) + 4*0.05*x(3) + 6*0.03*x(4)$$

$$+ 0.08*x(5) + 0.07*x(6) + 0.06*x(7) + 0.03*x(8) = 1.25.$$

The third constraint is that the weight of molybdenum is 1.25 tons.

$$\begin{aligned} 5*0.03*x(1) + 3*0.03*x(2) + 4*0.04*x(3) + 6*0.04*x(4) \\ + 0.06*x(5) + 0.07*x(6) + 0.08*x(7) + 0.09*x(8) = 1.25. \end{aligned}$$

In matrix form,  $Aeq*x = beq$ , where

```
Aeq = [5,3,4,6,1,1,1,1;
       5*0.05,3*0.04,4*0.05,6*0.03,0.08,0.07,0.06,0.03;
       5*0.03,3*0.03,4*0.04,6*0.04,0.06,0.07,0.08,0.09];
beq = [25;1.25;1.25];
```

Each variable is bounded below by zero. The integer variables are bounded above by one.

```
lb = zeros(8,1);
ub = ones(8,1);
ub(5:end) = Inf; % No upper bound on noninteger variables
```

### Solve the problem

Now that you have all the inputs, call the solver.

```
[x,fval] = intlinprog(f,intcon,[],[],Aeq,beq,lb,ub);
```

View the solution.

```
x,fval
```

```
x =
```

```
1.0000
1.0000
0
1.0000
7.2500
0
0.2500
3.5000
```

```
fval =
```

```
8.4950e+03
```

The optimal purchase costs \$8,495. Buy ingots **1**, **2**, and **4**, but not **3**, and buy 7.25 tons of alloy **1**, 0.25 ton of alloy **3**, and 3.5 tons of scrap steel.

Set `intcon = []` to see the effect of solving the problem without integer constraints. The solution is different, and is not sensible, because you cannot purchase a fraction of an ingot.

## Factory, Warehouse, Sales Allocation Model: 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”.

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

$$\sum_w x(p, f, w) \leq pcap(f, p) \text{ (capacity of factory).}$$

$$\sum_f x(p, f, w) = \sum_s (d(s, p) \cdot y(s, w)) \text{ (demand is met).}$$

$$\sum_p \sum_s \frac{d(s, p)}{turn(p)} \cdot y(s, w) \leq wcap(w) \text{ (capacity of warehouse).}$$

$$\sum_w y(s, w) = 1 \text{ (each sales outlet associates to one warehouse).}$$

$$x(p, f, w) \geq 0 \text{ (nonnegative production).}$$

$$y(s, w) \in \{0, 1\} \text{ (binary } y).$$

The variables  $x$  and  $y$  appear in the objective and constraint functions linearly. Because  $y$  is restricted to integer values, the problem is a mixed-integer linear program (MILP).

### Generate a Random Problem: Facility Locations

Set the values of the  $N$ ,  $f$ ,  $w$ , and  $s$  parameters, and generate the facility locations.

```
rng(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

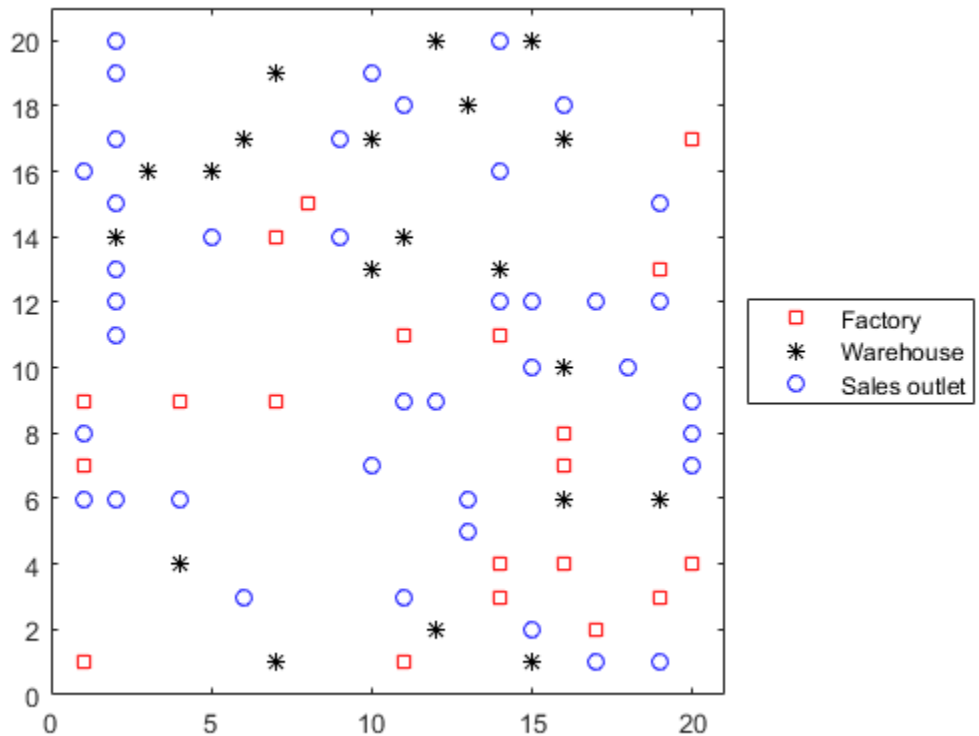
F = floor(f*N2); % number of factories
W = floor(w*N2); % number of warehouses
S = floor(s*N2); % number of sales outlets

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  $P \cdot F \cdot W + S \cdot W$  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  $P \times F$  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;

```

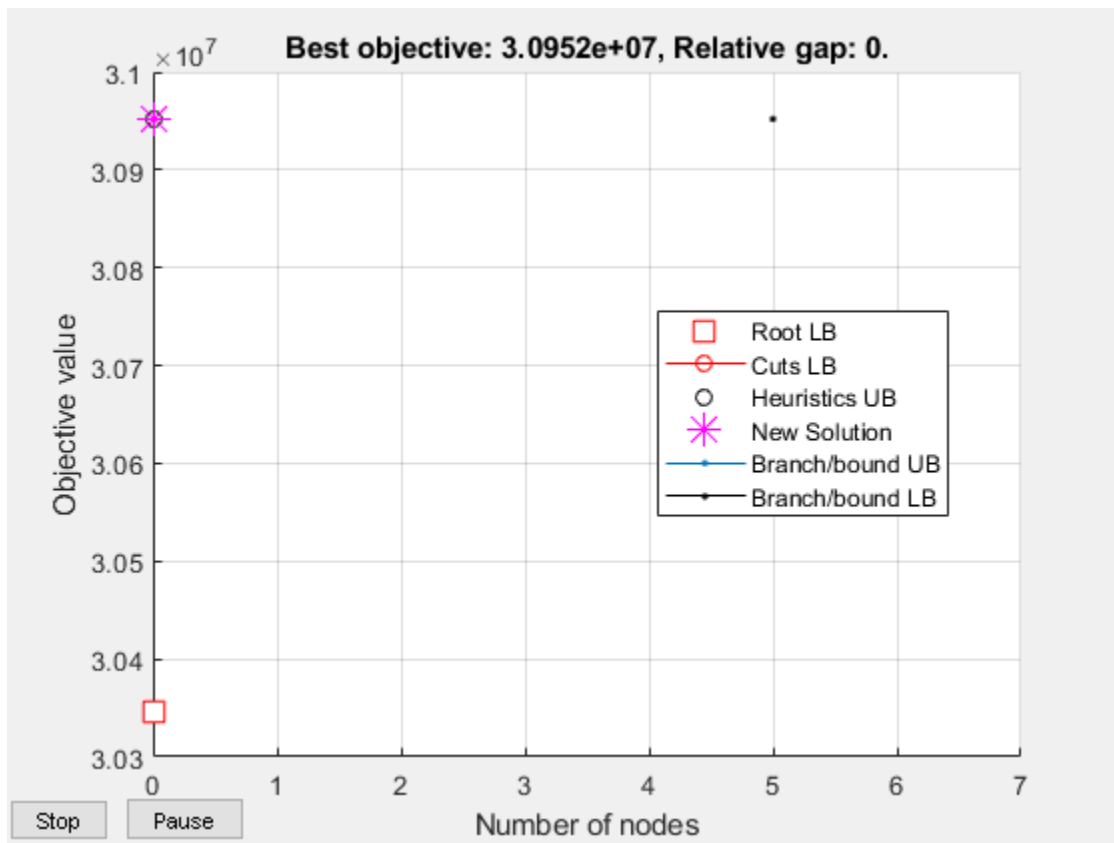
Turn off iterative display so that you don't get hundreds of lines of output. Include a plot function to monitor the solution progress.

```
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 = 9.0949e-13
```

```
infeas2 = norm(Aeq*solution - beq,Inf)
```

```
infeas2 = 5.8549e-12
```

Check that the integer components are really integers, or are close enough that it is reasonable to round them. To understand why these variables might not be exactly integers, see the documentation.

```
diffint = norm(solution(intcon) - round(solution(intcon)),Inf)
```

```
diffint = 6.9944e-15
```

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 = 9.0949e-13
```

```
infeas2 = norm(Aeq*solution - beq,Inf)
```

```
infeas2 = 5.8549e-12
```

```
diffrounding = norm(fval - obj(:)'*solution,Inf)
```

```
diffrounding = 2.6077e-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
```

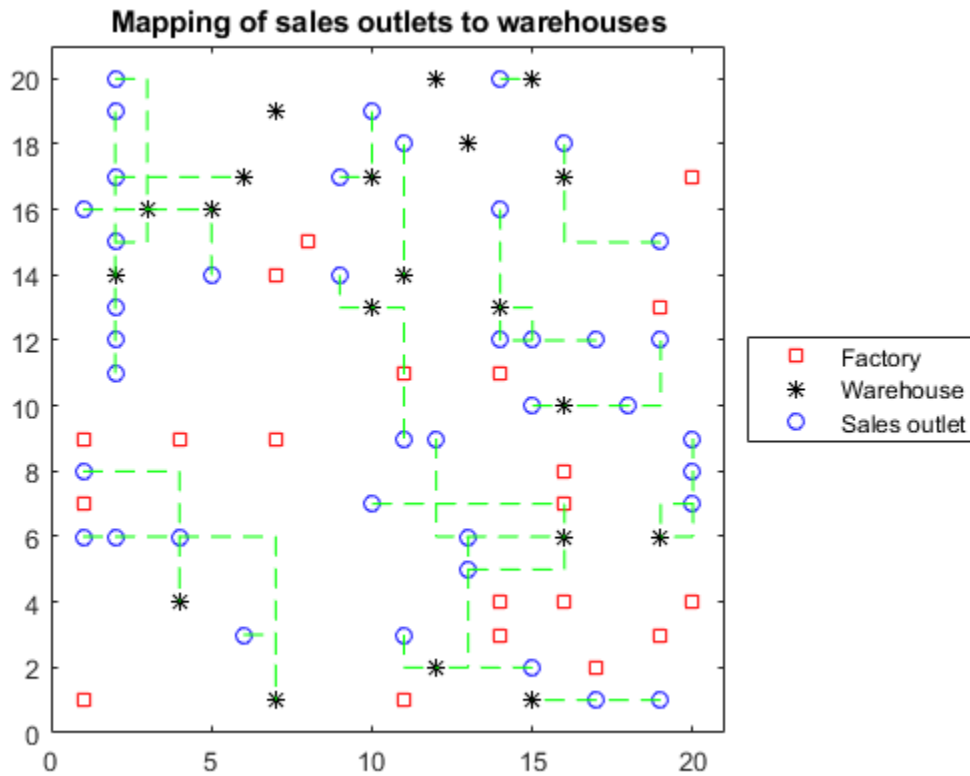
```
outlets = 1x20
```

```
    3    0    3    2    2    2    3    2    3    1    1    0    0    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”

## 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.

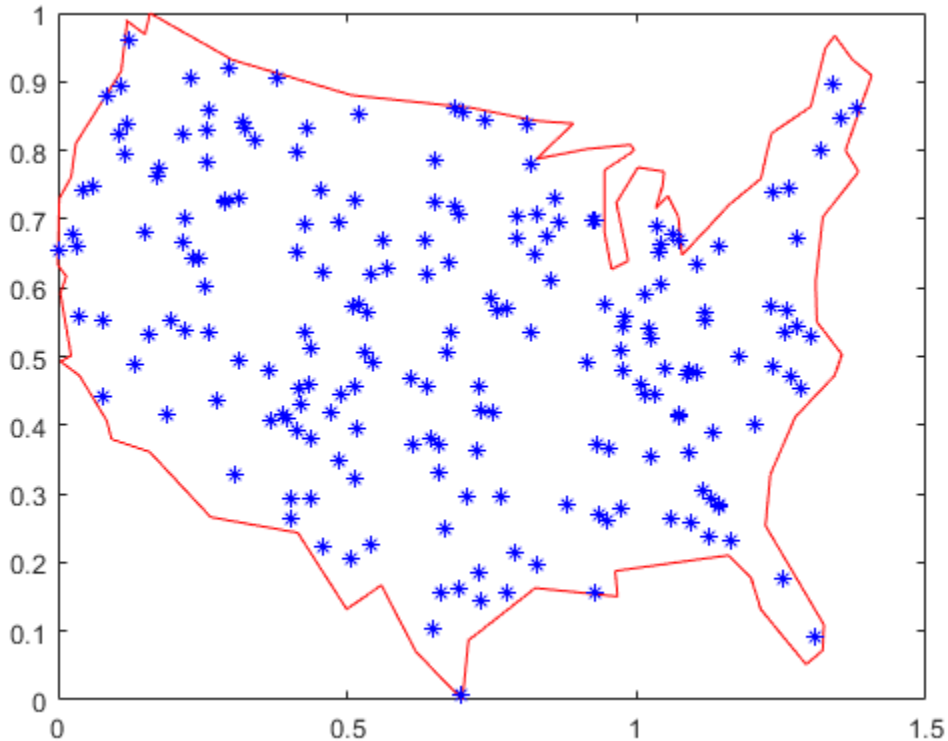
For the problem-based approach, see “Traveling Salesman Problem: Problem-Based”.

### Draw the Map and Stops

Generate random stops inside a crude polygonal representation of the continental U.S.

figure;

```
load('usborder.mat','x','y','xx','yy');
rng(3,'twister') % makes a plot with stops in Maine & Florida, and is reproducible
nStops = 200; % you can use any number, but the problem size scales as N^2
stopsLon = zeros(nStops,1); % allocate x-coordinates of nStops
stopsLat = stopsLon; % allocate y-coordinates
n = 1;
while (n <= nStops)
    xp = rand*1.5;
    yp = rand;
    if inpolygon(xp,yp,x,y) % test if inside the border
        stopsLon(n) = xp;
        stopsLat(n) = yp;
        n = n+1;
    end
end
plot(x,y,'Color','red'); % draw the outside border
hold on
% Add the stops to the map
plot(stopsLon,stopsLat,'*b')
```



### 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.

### Calculate Distances Between Points

Because there are 200 stops, there are 19,900 trips, meaning 19,900 binary variables (# variables = 200 choose 2).

Generate all the trips, meaning all pairs of stops.

```
idxs = nchoosek(1:nStops,2);
```

Calculate all the trip distances, assuming that the earth is flat in order to use the Pythagorean rule.

```
dist = hypot(stopsLat(idxs(:,1)) - stopsLat(idxs(:,2)), ...  
            stopsLon(idxs(:,1)) - stopsLon(idxs(:,2)));  
lendist = length(dist);
```

With this definition of the `dist` vector, the length of a tour is

```
dist'*x_tsp
```

where `x_tsp` is the binary solution vector. This is the distance of a tour that you try to minimize.

### Equality Constraints

The problem has two types of equality constraints. The first enforces that there must be 200 trips total. The second enforces that each stop must have two trips attached to it (there must be a trip to each stop and a trip departing each stop).

Specify the first type of equality constraint, that you must have `nStops` trips, in the form `Aeq*x_tsp = beq`.

```
Aeq = spones(1:length(idxs)); % Adds up the number of trips  
beq = nStops;
```

To specify the second type of equality constraint, that there needs to be two trips attached to each stop, extend the `Aeq` matrix as sparse.

```
Aeq = [Aeq; spalloc(nStops, length(idxs), nStops*(nStops-1))]; % allocate a sparse matrix  
for ii = 1:nStops  
    whichIdxs = (idxs == ii); % find the trips that include stop ii  
    whichIdxs = sparse(sum(whichIdxs,2)); % include trips where ii is at either end  
    Aeq(ii+1,:) = whichIdxs'; % include in the constraint matrix  
end  
beq = [beq; 2*ones(nStops,1)];
```

## Binary Bounds

All decision variables are binary. Now, set the `intcon` argument to the number of decision variables, put a lower bound of 0 on each, and an upper bound of 1.

```
intcon = 1:lendist;  
lb = zeros(lendist,1);  
ub = ones(lendist,1);
```

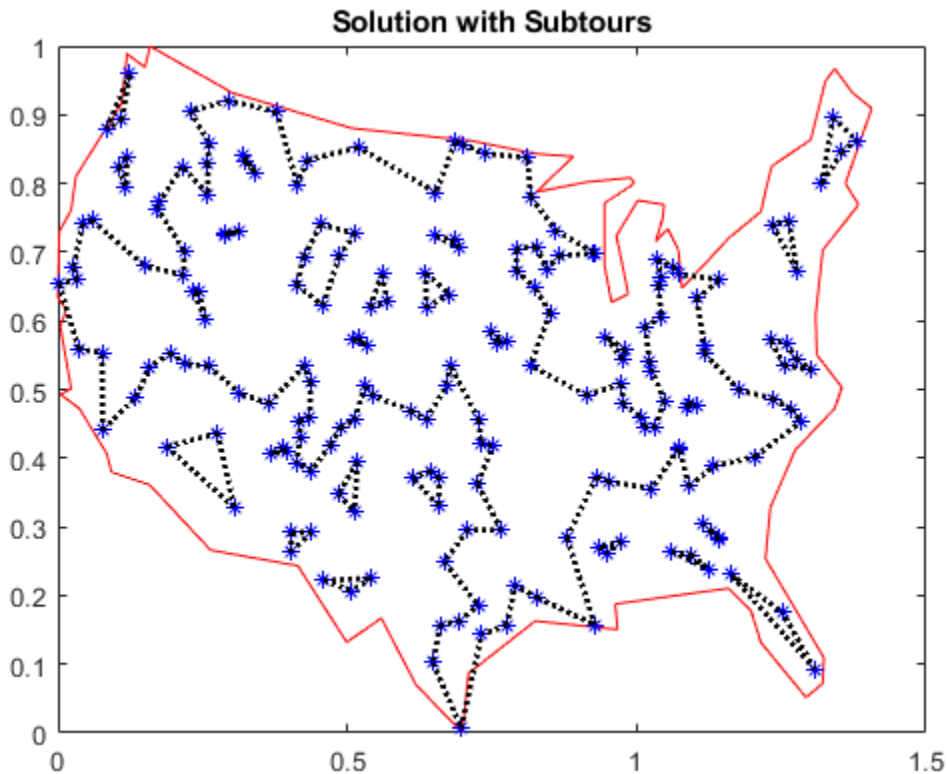
## Optimize Using `intlinprog`

The problem is ready 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);
```

## Visualize the Solution

```
segments = find(x_tsp); % Get indices of lines on optimal path  
lh = zeros(nStops,1); % Use to store handles to lines on plot  
lh = updateSalesmanPlot(lh,x_tsp,idxs,stopsLon,stopsLat);  
title('Solution with Subtours');
```



As can be seen on the map, the solution has several subtours. The constraints specified so far do not prevent these subtours from happening. In order to prevent any possible subtour from happening, you would need an incredibly large number of inequality constraints.

### Subtour Constraints

Because you can't add all of the subtour constraints, take an iterative approach. Detect the subtours in the current solution, then add inequality constraints to prevent those particular subtours from happening. By doing this, you find a suitable tour in a few iterations.



Eliminate subtours with inequality constraints. An example of how this works is if you have five points in a subtour, then you have five lines connecting those points to create the subtour. Eliminate this subtour by implementing an inequality constraint to say there must be less than or equal to four lines between these five points.

Even more, find all lines between these five points, and constrain the solution not to have more than four of these lines present. This is a correct constraint because if five or more of the lines existed in a solution, then the solution would have a subtour (a graph with  $n$  nodes and  $n$  edges always contains a cycle).

The `detectSubtours` function analyzes the solution and returns a cell array of vectors. Each vector in the cell array contains the stops involved in that particular subtour.

```
tours = detectSubtours(x_tsp,idxs);
numtours = length(tours); % number of subtours
fprintf('# of subtours: %d\n',numtours);
```

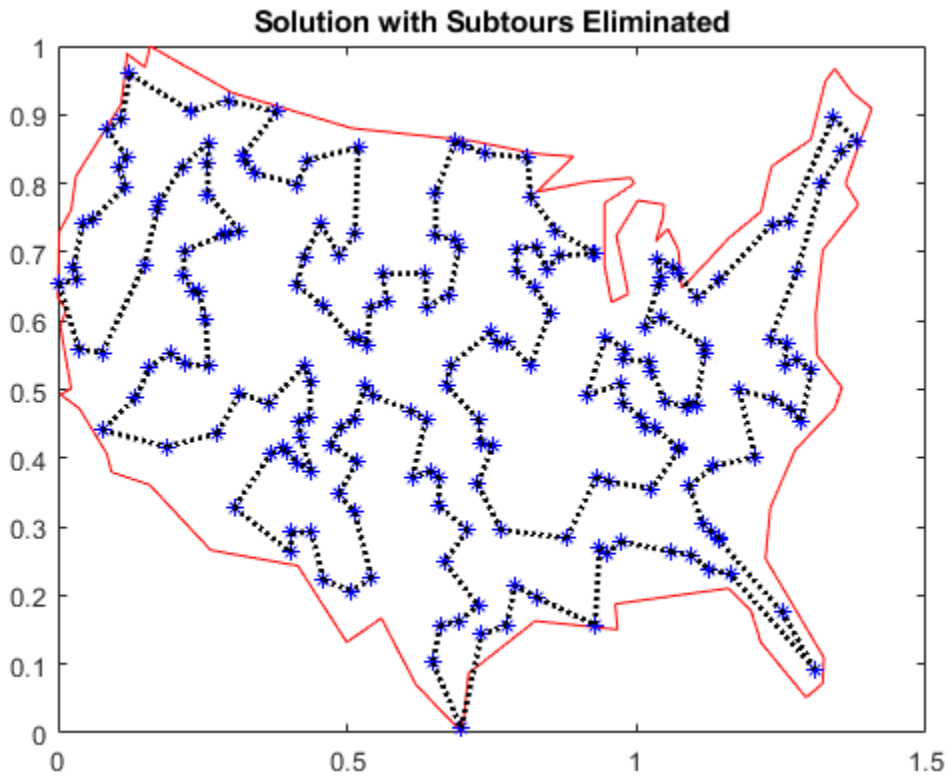
```
# of subtours: 27
```

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 = tours{ii}; % Extract the current subtour
        % The next lines find all of the variables associated with the
        % particular subtour, then add an inequality constraint to prohibit
        % that subtour and all subtours that use those stops.
        variations = nchoosek(1:length(subTourIdx), 2);
        for jj = 1:length(variations)
            whichVar = (sum(idxs == subTourIdx(variations(jj, 1)), 2)) & ...
                (sum(idxs == subTourIdx(variations(jj, 2)), 2));
            A(rowIdx, whichVar) = 1;
        end
        b(rowIdx) = length(subTourIdx) - 1; % One less trip than subtour stops
    end
end

% Try to optimize again
```

```
[x_tsp,costopt,exitflag,output] = intlinprog(dist,intcon,A,b,Aeq,beq,lb,ub,opts);  
  
% Visualize result  
lh = updateSalesmanPlot(lh,x_tsp,idxs,stopsLon,stopsLat);  
  
% How many subtours this time?  
tours = detectSubtours(x_tsp,idxs);  
numtours = length(tours); % number of subtours  
fprintf('# of subtours: %d\n',numtours);  
end  
  
# 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 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.absolute_gap)
```

```
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”

## 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.

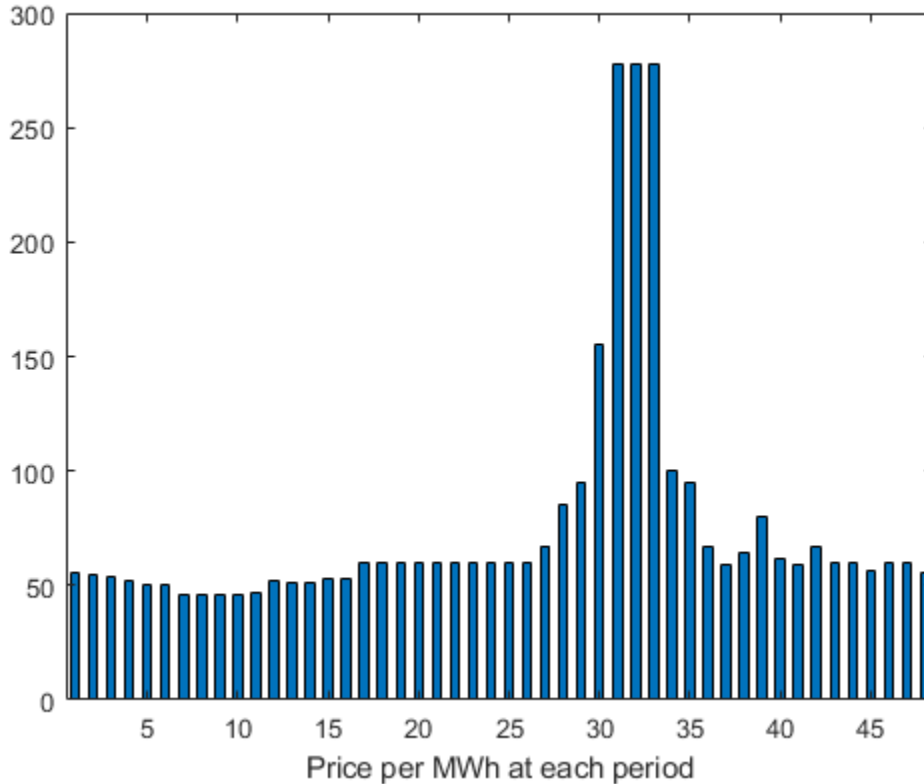
For the problem-based approach to this problem, see “Optimal Dispatch of Power Generators: Problem-Based”.

### 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/About-AEMO/Legal-Notices/Copyright-Permissions>.

```
load dispatchPrice; % Get poolPrice, which is the revenue per MWh
bar(poolPrice,.5)
xlim([.5,48.5])
xlabel('Price per MWh at each period')
```



There is a cost to start a generator after it has been off. The other constraint is a maximum fuel usage for the day. The maximum fuel constraint is because you buy your fuel a day ahead of time, so can use only what you just bought.

**Problem Notation and Parameters**

You can formulate the scheduling problem as a binary integer programming problem as follows. Define indexes  $i$ ,  $j$ , and  $k$ , and a binary scheduling vector  $y$  as:

- $nPeriods$  = the number of time periods, 48 in this case.
- $i$  = a time period,  $1 \leq i \leq 48$ .
- $j$  = a generator index,  $1 \leq j \leq 2$  for this example.

- $y(i, j, k) = 1$  when period  $i$ , generator  $j$  is operating at power level  $k$ . Let low power be  $k = 1$ , and high power be  $k = 2$ . The generator is off when  $\text{sum}_k y(i, j, k) = 0$ .

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  $\text{sum}_k y(i, j, k) = 0$  and  $\text{sum}_k y(i+1, j, k) = 1$ .

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.

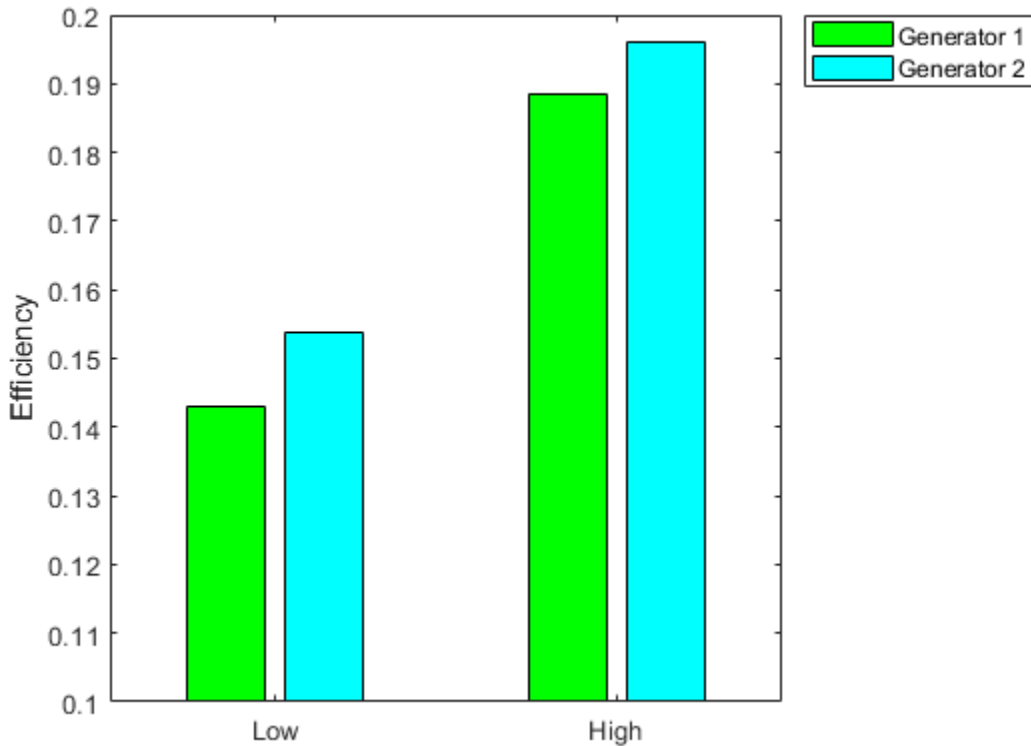
```
fuelPrice = 3;
totalfuel = 3.95e4;
nPeriods = length(poolPrice); % 48 periods
nGens = 2; % Two generators
gen = [61,152;50,150]; % Generator 1 low = 61 MW, high = 152 MW
fuel = [427,806;325,765]; % Fuel consumption for generator 2 is low = 325, high = 765
startCost = 1e4; % Cost to start a generator 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  $n\text{Periods}$ -by- $n\text{Gens}$ -by-2 array, and  $z$  is an  $n\text{Periods}$ -by- $n\text{Gens}$  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  $A*x \leq b$ , the number of columns in the  $A$  matrix must be the same as the length of  $x$ , which is the same as the length of  $lb$ . To create rows of  $A$  of the appropriate size, create zero matrices of the sizes of the  $y$  and  $z$  matrices.

```
cleary = zeros(nPeriods,nGens,2);
clearz = zeros(nPeriods,nGens);
```

To ensure that the power level has no more than one component equal to 1, set a linear inequality constraint:

```
x(i,j,1) + x(i,j,2) <= 1
```

```
A = spalloc(nPeriods*nGens,length(lb),2*nPeriods*nGens); % nPeriods*nGens inequalities
counter = 1;
for ii = 1:nPeriods
    for jj = 1:nGens
        temp = cleary;
        temp(ii,jj,:) = 1;
```

```
        addrow = [temp(:);clearz(:)]';  
        A(counter,:) = sparse(addrow);  
        counter = counter + 1;  
    end  
end  
b = ones(nPeriods*nGens,1); % A*x <= b means no more than one of x(i,j,1) and x(i,j,2)
```

The running cost per period is the cost for fuel for that period. For generator  $j$  operating at level  $k$ , the cost is `fuelPrice * fuel(j,k)`.

To ensure that the generators do not use too much fuel, create an inequality constraint on the sum of fuel usage.

```
yFuel = lby; % Initialize fuel usage array  
yFuel(:,1,1) = fuel(1,1); % Fuel use of generator 1 in low setting  
yFuel(:,1,2) = fuel(1,2); % Fuel use of generator 1 in high setting  
yFuel(:,2,1) = fuel(2,1); % Fuel use of generator 2 in low setting  
yFuel(:,2,2) = fuel(2,2); % Fuel use of generator 2 in high setting  
  
addrow = [yFuel(:);clearz(:)]';  
A = [A;sparse(addrow)];  
b = [b;totalfuel]; % A*x <= b means the total fuel usage is <= totalfuel
```

### Set the Generator Startup Indicator Variables

How can you get the solver to set the  $z$  variables automatically to match the active/off periods that the  $y$  variables represent? Recall that the condition to satisfy is  $z(i,j) = 1$  exactly when

$$\text{sum}_k y(i,j,k) = 0 \text{ and } \text{sum}_k y(i+1,j,k) = 1.$$

Notice that

$$\text{sum}_k ( - y(i,j,k) + y(i+1,j,k) ) > 0 \text{ exactly when you want } z(i,j) = 1.$$

Therefore, include the linear inequality constraints

$$\text{sum}_k ( - y(i,j,k) + y(i+1,j,k) ) - z(i,j) <= 0$$

in the problem formulation, and include the  $z$  variables in the objective function cost. By including the  $z$  variables in the objective function, the solver attempts to lower the values of the  $z$  variables, meaning it tries to set them all equal to 0. But for those intervals when a generator turns on, the linear inequality forces the  $z(i,j)$  to equal 1.

Add extra rows to the linear inequality constraint matrix A to represent these new inequalities. Wrap around the time so that interval 1 logically follows interval 48.

```
tempA = spalloc(nPeriods*nGens,length(lb),2*nPeriods*nGens);
counter = 1;
for ii = 1:nPeriods
    for jj = 1:nGens
        temp = clearz;
        tempy = clearz;
        temp(ii,jj,1) = -1;
        temp(ii,jj,2) = -1;
        if ii < nPeriods % Intervals 1 to 47
            temp(ii+1,jj,1) = 1;
            temp(ii+1,jj,2) = 1;
        else % Interval 1 follows interval 48
            temp(1,jj,1) = 1;
            temp(1,jj,2) = 1;
        end
        tempy(ii,jj) = -1;
        temp = [temp(:);tempy(:)]'; % Row vector for inclusion in tempA matrix
        tempA(counter,:) = sparse(temp);
        counter = counter + 1;
    end
end
A = [A;tempA];
b = [b;zeros(nPeriods*nGens,1)]; % A*x <= b sets z(i,j) = 1 at generator startup
```

### Sparsity of Constraints

If you have a large problem, using sparse constraint matrices saves memory, and can save computational time as well. The constraint matrix A is quite sparse:

```
filledfraction = nnz(A)/numel(A)
```

```
filledfraction = 0.0155
```

`intlinprog` accepts sparse linear constraint matrices A and Aeq, but requires their corresponding vector constraints b and beq to be full.

### Define Objective

The objective function includes fuel costs for running the generators, revenue from running the generators, and costs for starting the generators.

```
generatorlevel = lby; % Generation in MW, start with 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

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

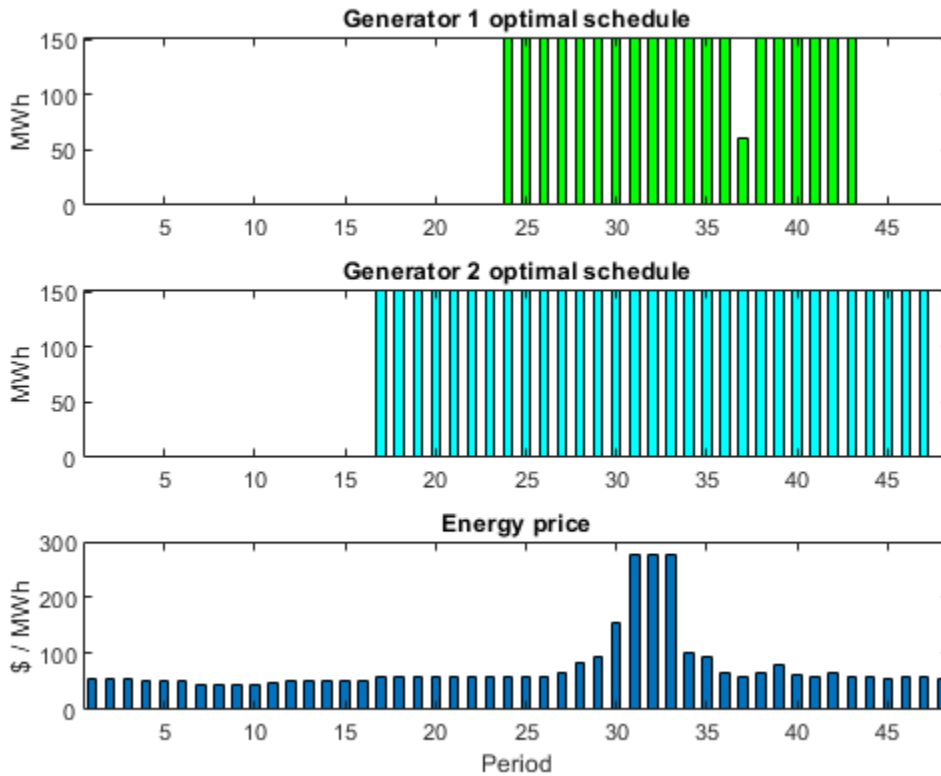
The easiest way to examine the solution is dividing the solution vector  $x$  into its two components,  $y$  and  $z$ .

```
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)
```

```
thepreiod = 2×1
```

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).

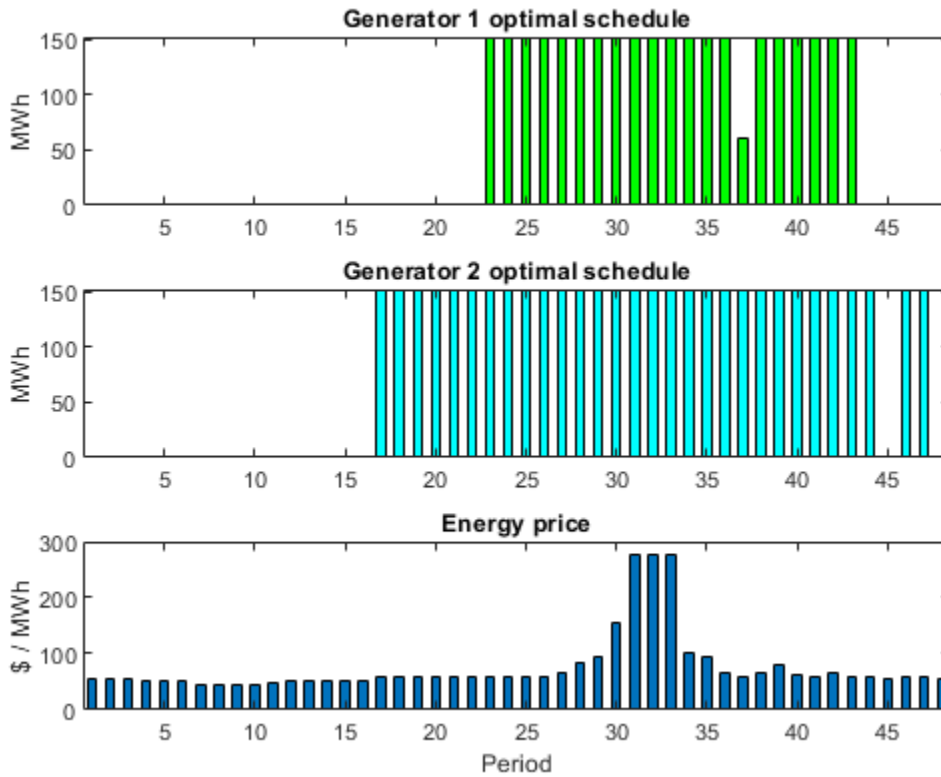
```
ysolutionnew = xnew(1:nPeriods*nGens*2);
zsolutionnew = xnew(nPeriods*nGens*2+1:end);
ysolutionnew = reshape(ysolutionnew,[nPeriods,nGens,2]);
zsolutionnew = reshape(zsolutionnew,[nPeriods,nGens]);

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”

## 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 9-156.

### 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  $\lambda$  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  $M \leq N$ .
- Having at least  $m$  assets in the portfolio, where  $0 < m \leq M$ .
- Having *semicontinuous* constraints, meaning either  $x(i) = 0$ , or  $f_{\min} \leq x(i) \leq f_{\max}$  for some fixed fractions  $f_{\min} > 0$  and  $f_{\max} \geq f_{\min}$ .

You cannot include these constraints in `quadprog`. The difficulty is the discrete nature of the constraints. Furthermore, while the mixed-integer linear programming solver `intlinprog` does handle discrete constraints, it does not address quadratic objective functions.

This example constructs a sequence of MILP problems that satisfy the constraints, and that increasingly approximate the quadratic objective function. While this technique works for this example, it might not apply to different problem or constraint types.

Begin by modeling the constraints.

## Modeling Discrete Constraints

$x$  is the vector of asset allocation fractions, with  $0 \leq x(i) \leq 1$  for each  $i$ . To model the number of assets in the portfolio, you need indicator variables  $v$  such that  $v(i) = 0$  when  $x(i) = 0$ , and  $v(i) = 1$  when  $x(i) > 0$ . To get variables that satisfy this restriction, set the  $v$  vector to be a binary variable, and impose the linear constraints

$$v(i)f_{\min} \leq x(i) \leq v(i)f_{\max}.$$

These inequalities both enforce that  $x(i)$  and  $v(i)$  are zero at exactly the same time, and they also enforce that  $f_{\min} \leq x(i) \leq f_{\max}$  whenever  $x(i) > 0$ .

Also, to enforce the constraints on the number of assets in the portfolio, impose the linear constraints

$$m \leq \sum_i v(i) \leq M.$$

## Objective and Successive Linear Approximations

As first formulated, you try to maximize the objective function. However, all Optimization Toolbox™ solvers minimize. So formulate the problem as minimizing the negative of the objective:

$$\min_x \lambda x^T Q x - r^T x.$$

This objective function is nonlinear. The `intlinprog` MILP solver requires a linear objective function. There is a standard technique to reformulate this problem into one with linear objective and nonlinear constraints. Introduce a slack variable  $z$  to represent the quadratic term.

$$\min_{x,z} \lambda z - r^T x \text{ such that } x^T Q x - z \leq 0, \quad z \geq 0.$$

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  $\delta$  is a variable vector, the first-order Taylor approximation to the constraint is

$$x^T Q x - z = x_0^T Q x_0 + 2x_0^T Q \delta - z + O(|\delta|^2).$$

Replacing  $\delta$  by  $x - x_0$  gives

$$x^T Q x - z = -x_0^T Q x_0 + 2x_0^T Q x - z + O(|x - x_0|^2).$$

For each intermediate solution  $x_k$  you introduce a new linear constraint in  $x$  and  $z$  as the linear part of the expression above:

$$-x_k^T Q x_k + 2x_k^T Q x - z \leq 0.$$

This has the form  $Ax \leq b$ , where  $A = 2x_k^T Q$ , there is a  $-1$  multiplier for the  $z$  term, and  $b = x_k^T Q x_k$ .

This method of adding new linear constraints to the problem is called a cutting plane method. For details, see J. E. Kelley, Jr. "The Cutting-Plane Method for Solving Convex Programs." J. Soc. Indust. Appl. Math. Vol. 8, No. 4, pp. 703-712, December, 1960.

### **MATLAB® Problem Formulation**

To express problems for the `intlinprog` solver, you need to do the following:

- Decide what your variables represent
- Express lower and upper bounds in terms of these variables
- Give linear equality and inequality matrices

Have the first  $N$  variables represent the  $x$  vector, the next  $N$  variables represent the binary  $v$  vector, and the final variable represent the  $z$  slack variable. There are  $2N + 1$  variables in the problem.

Load the data for the problem. This data has 225 expected returns in the vector  $r$  and the covariance of the returns in the 225-by-225 matrix  $Q$ . The data is the same as in the Using Quadratic Programming on Portfolio Optimization Problems example.

```
load port5
r = mean_return;
Q = Correlation .* (stdDev_return * stdDev_return');
```

Set the number of assets as  $N$ .

```
N = length(r);
```

Set indexes for the variables

```
xvars = 1:N;
vvars = N+1:2*N;
zvar = 2*N+1;
```

The lower bounds of all the  $2N+1$  variables in the problem are zero. The upper bounds of the first  $2N$  variables are one, and the last variable has no upper bound.

```
lb = zeros(2*N+1,1);
ub = ones(2*N+1,1);
ub(zvar) = Inf;
```

Set the number of assets in the solution to be between 100 and 150. Incorporate this constraint into the problem in the form, namely

$$m \leq \sum_i v(i) \leq M,$$

by writing two linear constraints of the form  $Ax \leq b$ :

$$\sum_i v(i) \leq M$$

$$\sum_i -v(i) \leq -m.$$

```
M = 150;
m = 100;
A = zeros(1,2*N+1); % Allocate A matrix
A(vvars) = 1; % A*x represents the sum of the v(i)
A = [A;-A];
b = zeros(2,1); % Allocate b vector
b(1) = M;
b(2) = -m;
```

Include semicontinuous constraints. Take the minimal nonzero fraction of assets to be  $0.001$  for each asset type, and the maximal fraction to be  $0.05$ .

```
fmin = 0.001;
fmax = 0.05;
```

Include the inequalities  $x(i) \leq fmax(i) * v(i)$  and  $fmin(i) * v(i) \leq x(i)$  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  $\lambda$  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-10,
    '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  $Ax \leq b$  comes from the linear approximation

$$-x_k^T Q x_k + 2x_k^T Q x - z \leq 0.$$

You see that the new row of  $A = 2x_k^T Q$  and the new element in  $b = x_k^T Q x_k$ , with the  $z$  term represented by a -1 coefficient in  $A$ .

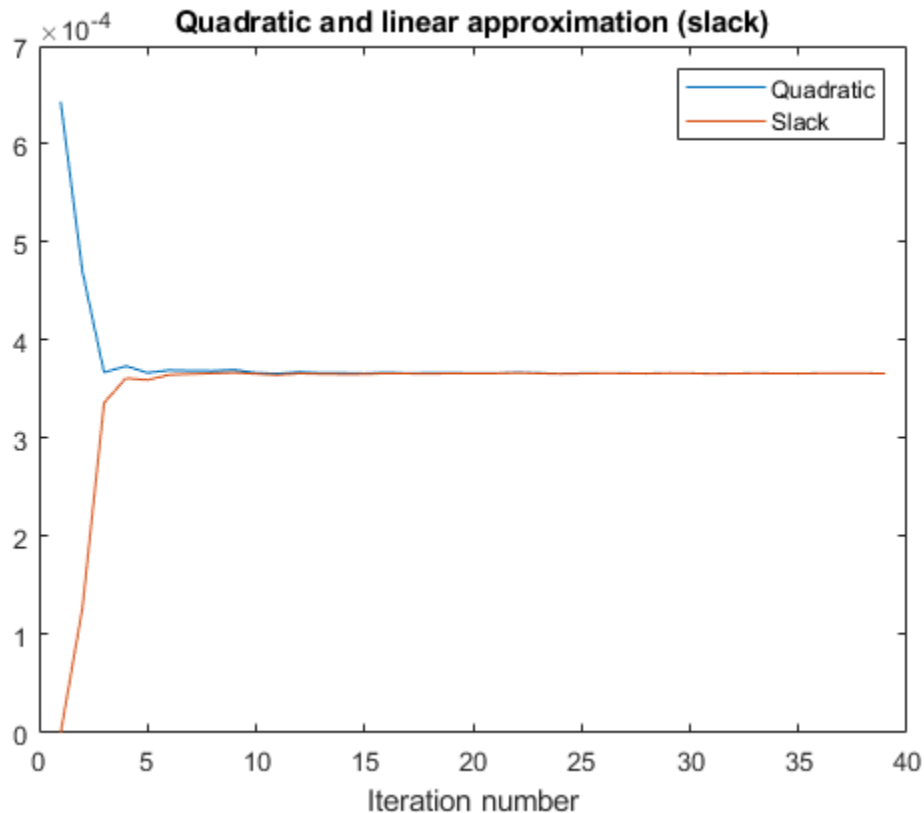
After you find a new solution, use a linear constraint halfway between the old and new solutions. This heuristic way of including linear constraints can be faster than simply taking the new solution. To use the solution instead of the halfway heuristic, comment the "Midway" line below, and uncomment the following one.

```
while abs((zslack - truequadratic)/truequadratic) > thediff % relative error
    newRow = horzcat(2*assets'*Q,zeros(1,N),-1); % Linearized constraint
    rhs = assets'*Q*assets; % right hand side of the linearized constraint
    A = [A;newRow];
    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
```

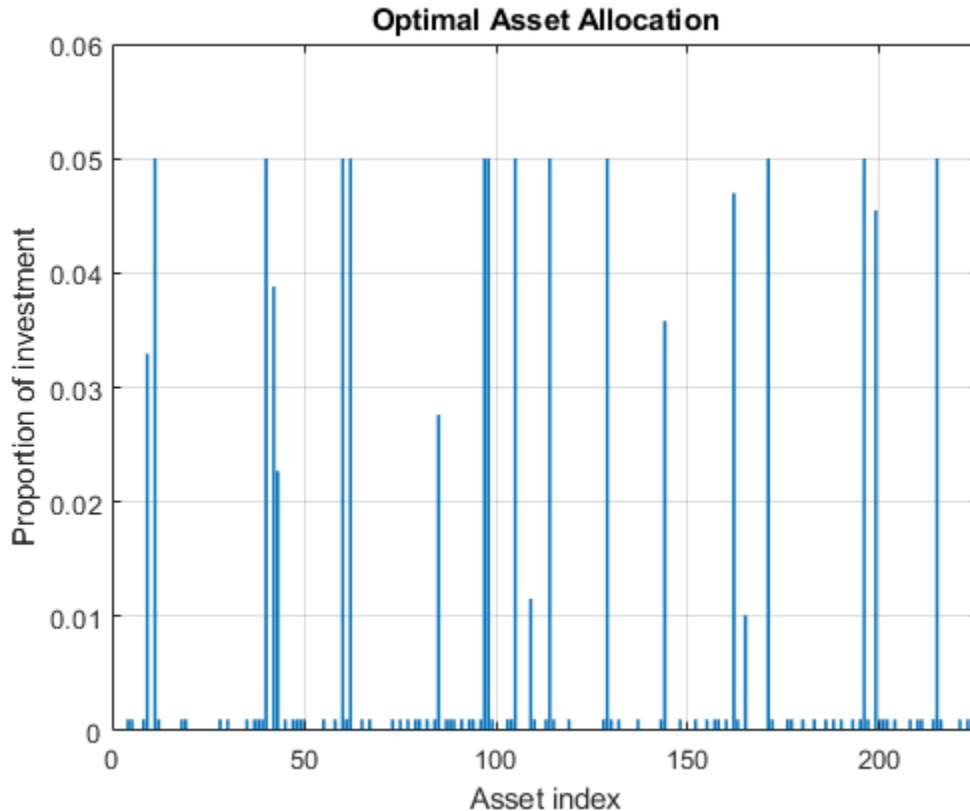
The absolute gap is zero, indicating that the MILP solution is accurate.

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  $f_{\min} = 0.001$  and  $f_{\max} = 0.05$ .

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”

## 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”.

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.
```

	2			3			4	
6								3
		4				5		
			8		6			
8				1				6
			7		5			
		7				6		
4								8
	3			4			2	

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  $x(i, j, 1), \dots, x(i, j, 9)$ . 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  $k \neq m$ .

### 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.
```

```
Cut Generation:    Applied 1 strong CG cut,
                   and 3 zero-half cuts.
                   Lower bound is 29565.000000.
                   Relative gap is 0.00%.
```

```
Optimal solution found.
```

```
Intlinprog stopped at the root node because the objective value is within a gap
tolerance of the optimal value, options.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”

## 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”.

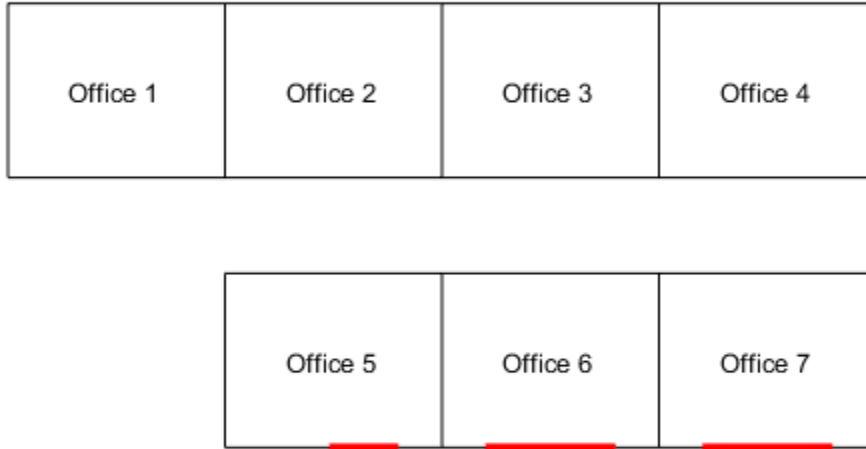
### Office Assignment Problem

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  $i$ th element of a person's preference vector is how highly they value the  $i$ th 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  $\text{sum}(x(8:14))=1$ . Represent these linear constraints in an equality matrix  $A_{eq}$  and vector  $beq$ , where  $A_{eq}x = beq$ , by building the appropriate matrices. The matrix  $A_{eq}$  consists of ones and zeros. For example, the second row of  $A_{eq}$  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 ... 0 0 0
```

There are seven 1s in columns 8 through 14 and 0s elsewhere. Then  $A_{eq}(2,:)x = 1$  is equivalent to  $\text{sum}(x(8:14)) = 1$ .

```
numOffices = 7;
numPeople = 6;
numDim = numOffices * numPeople;
onesvector = ones(1,numOffices);
```

Each row of  $A_{eq}$  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  $Ax \leq b$  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 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  $A(3, :)*x \leq 1$ , 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  $2*\text{numPairs}$  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,

$$x(i) + x(j) \leq 1.$$

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 value, options.AbsoluteGapTolerance = 0 (the default value). The intcon variables are integer within tolerance, 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

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`. `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
```

## See Also

### More About

- “Office Assignments by Binary Integer Programming: Problem-Based”

## 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”.

### 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.



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 over
        for j = 1:size(values)
            if values(j) > 0
                fprintf('Cut %g logs with pattern\n',values(j));
            end
        end
    end
end
```



```

    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
    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 1 logs with pattern
    3 cut(s) of length 12
    Waste of this pattern is 4
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 36 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”

## 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 9-52.

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

$$\sum_w x(p, f, w) \leq pcap(f, p) \text{ (capacity of factory).}$$

$$\sum_f x(p, f, w) = \sum_s (d(s, p) \cdot y(s, w)) \text{ (demand is met).}$$

$$\sum_p \sum_s \frac{d(s, p)}{turn(p)} \cdot y(s, w) \leq wcap(w) \text{ (capacity of warehouse).}$$

$$\sum_w y(s, w) = 1 \text{ (each sales outlet associates to one warehouse).}$$

$$x(p, f, w) \geq 0 \text{ (nonnegative production).}$$

$$y(s, w) \in \{0, 1\} \text{ (binary } y).$$

The variables  $x$  and  $y$  appear in the objective and constraint functions linearly. Because  $y$  is restricted to integer values, the problem is a mixed-integer linear program (MILP).

### Generate a Random Problem: Facility Locations

Set the values of the  $N$ ,  $f$ ,  $w$ , and  $s$  parameters, and generate the facility locations.

```

rng(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

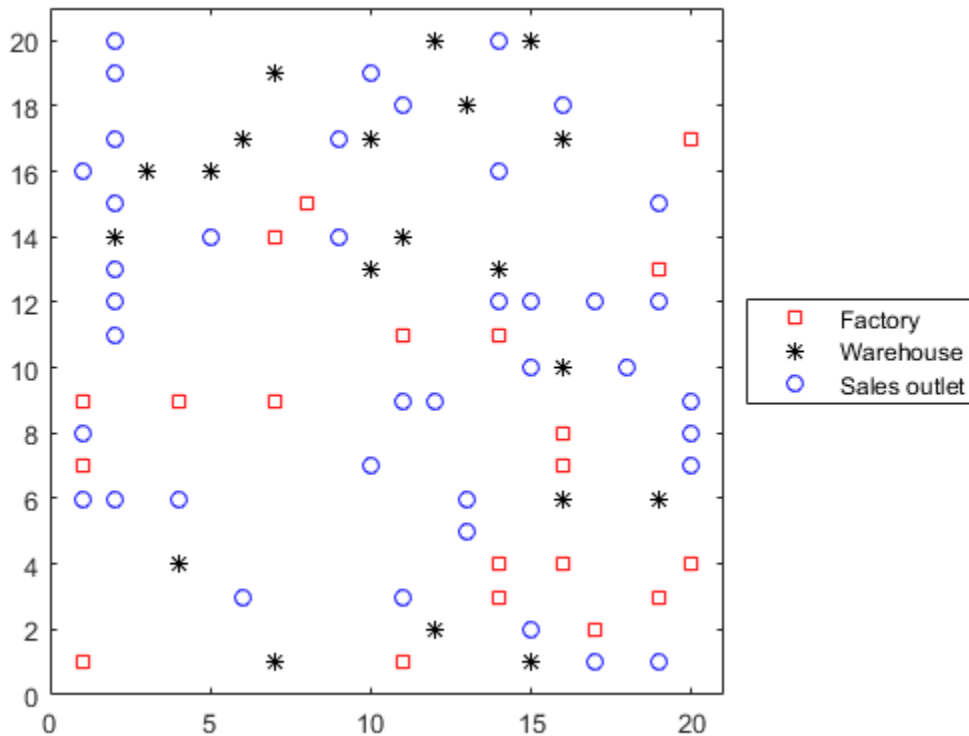
F = floor(f*N2); % number of factories
W = floor(w*N2); % number of warehouses
S = floor(s*N2); % number of sales outlets

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;
```

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;
```

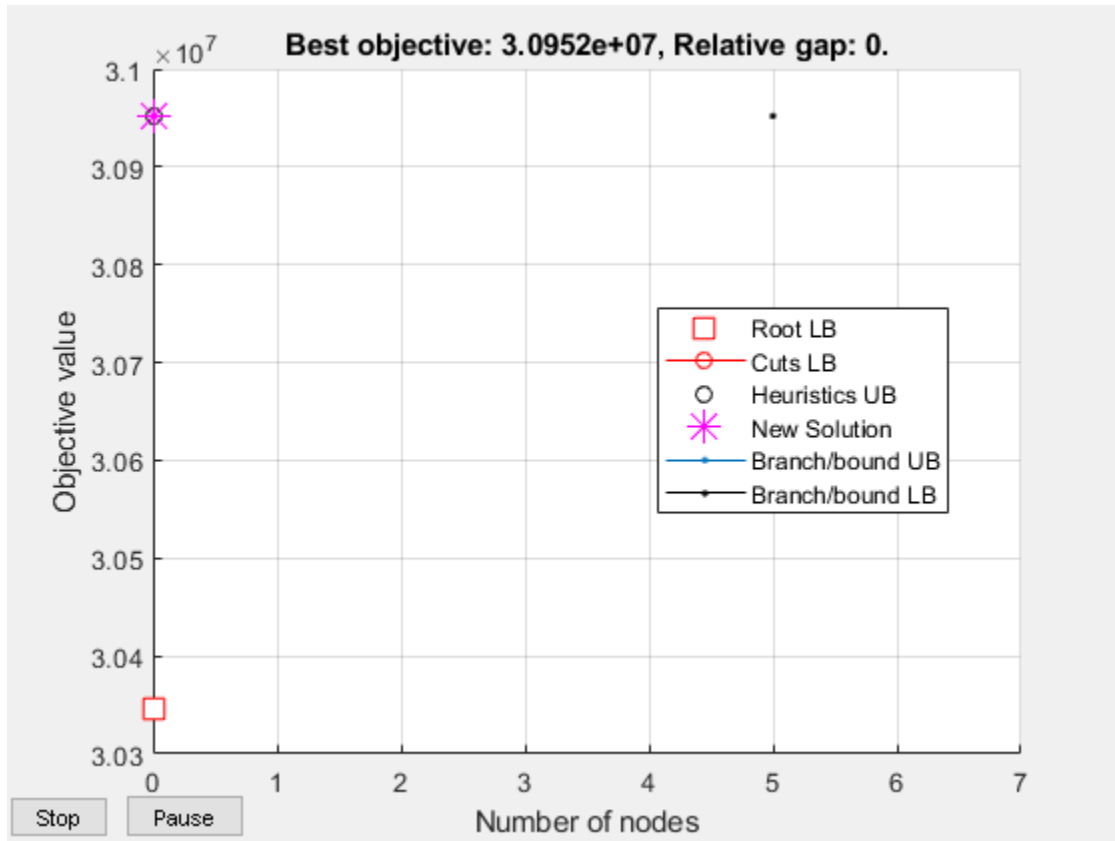
### **Solve the Problem**

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 = 2.2737e-12
infeas2 = max(max(infeasibility(demconstr,sol)))
infeas2 = 2.6830e-11
infeas3 = max(infeasibility(warecap,sol))
infeas3 = 0
infeas4 = max(infeasibility(salesware,sol))
infeas4 = 3.1086e-15
    
```

Round the  $y$  portion of the solution to be exactly integer-valued. To understand why these variables might not be exactly integers, see the documentation.

```
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 = 1x20
```

```

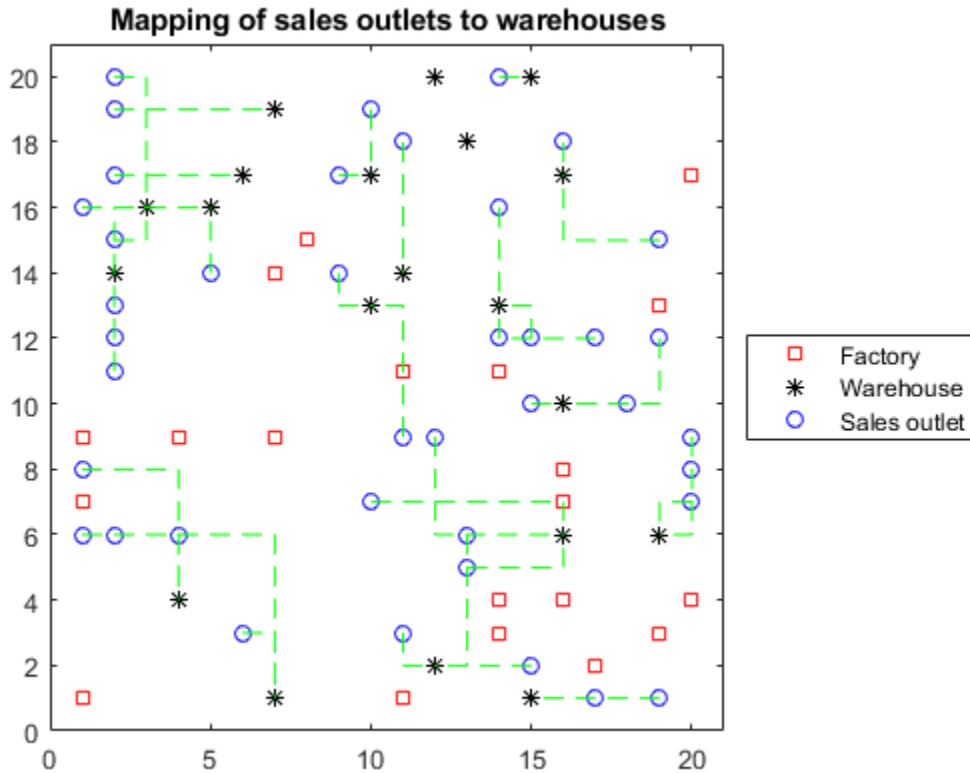
     2     1     3     2     2     2     3     2     3     1     1     0     0     3
    
```

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], 'g--')
    else % Draw x direction first the rest of the time
        plot([xsales,xwarehouse,xwarehouse],[ysales,ysales,ywarehouse], 'g--')
    end
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: Solver-Based" on page 9-52

## 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.

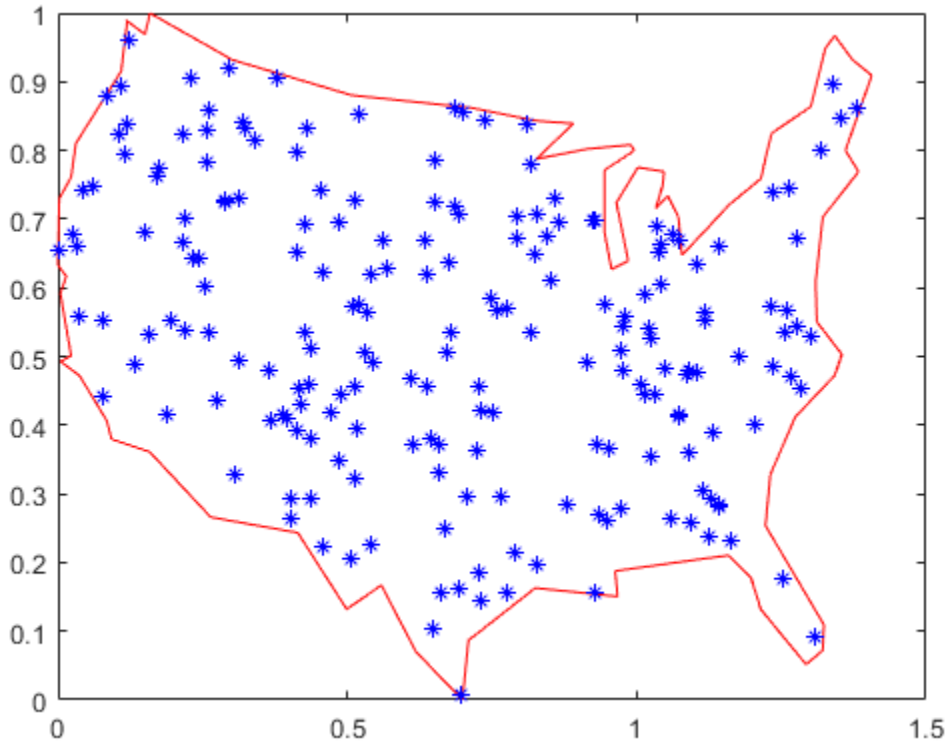
For the solver-based approach to this problem, see “Traveling Salesman Problem: Solver-Based” on page 9-64.

### Draw the Map and Stops

Generate random stops inside a crude polygonal representation of the continental U.S.

```
figure;

load('usborder.mat','x','y','xx','yy');
rng(3,'twister') % makes a plot with stops in Maine & Florida, and is reproducible
nStops = 200; % you can use any number, but the problem size scales as N^2
stopsLon = zeros(nStops,1); % allocate x-coordinates of nStops
stopsLat = stopsLon; % allocate y-coordinates
n = 1;
while (n <= nStops)
    xp = rand*1.5;
    yp = rand;
    if inpolygon(xp,yp,x,y) % test if inside the border
        stopsLon(n) = xp;
        stopsLat(n) = yp;
        n = n+1;
    end
end
plot(x,y,'Color','red'); % draw the outside border
hold on
% Add the stops to the map
plot(stopsLon,stopsLat,'*b')
hold off
```



### Problem Formulation

Formulate the 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.

## 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 Variables and Problem

Create a problem and binary variables.

```
tsp = optimproblem;
trips = optimvar('trips',lendist,1,'Type','integer','LowerBound',0,'UpperBound',1);
```

Include the objective function in the problem.

```
tsp.Objective = dist'*trips;
```

## Equality Constraints

The problem has two types of equality constraints. The first enforces that there must be 200 trips total. The second enforces that each stop must have two trips attached to it (there must be a trip to each stop and a trip departing each stop).

Specify the first type of equality constraint, that you must have `nStops` trips, and include it in the problem.

```
constrips = sum(trips) == nStops;
tsp.Constraints.constrips = constrips;
```

To specify the second type of equality constraint, that there needs to be two trips attached to each stop, find the trips for each stop and add up the number of trips for that stop. Look at trips that both start and end at that stop.

```
constr2trips = optimconstr(nStops,1);
for stops = 1:nStops
    whichIdxs = (idxs == stops);
    whichIdxs = any(whichIdxs,2); % start or end at stops
    constr2trips(stops) = sum(trips(whichIdxs)) == 2;
end
tsp.Constraints.constr2trips = constr2trips;
```

### Solve the Initial Problem

The problem is ready to be solved. To suppress iterative output, turn off the default display.

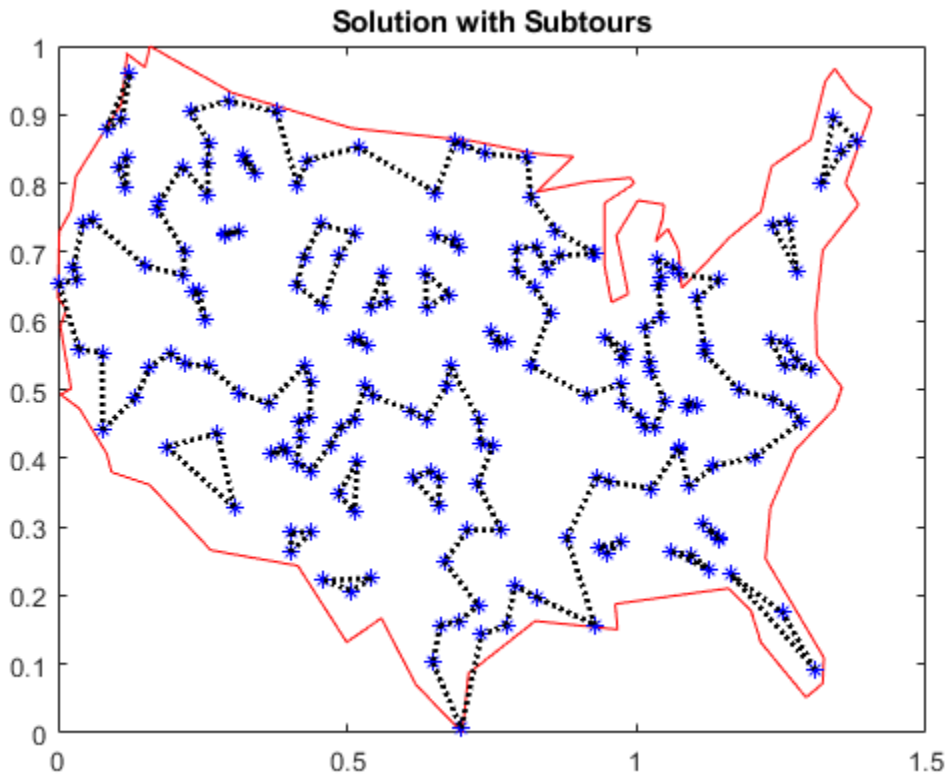
```
opts = optimoptions('intlinprog','Display','off');
tspSol = solve(tsp,'options',opts)
```

```
tspSol = struct with fields:
    trips: [19900x1 double]
```

### Visualize the Solution

```
hold on
segments = find(tspSol.trips); % Get indices of lines on optimal path
lh = zeros(nStops,1); % Use to store handles to lines on plot
lh = updateSalesmanPlot(lh,tspSol.trips,idxs,stopsLon,stopsLat);
title('Solution with Subtours');
```





As can be seen on the map, the solution has several subtours. The constraints specified so far do not prevent these subtours from happening. In order to prevent any possible subtour from happening, you would need an incredibly large number of inequality constraints.

### Subtour Constraints

Because you can't add all of the subtour constraints, take an iterative approach. Detect the subtours in the current solution, then add inequality constraints to prevent those particular subtours from happening. By doing this, you find a suitable tour in a few iterations.

Eliminate subtours with inequality constraints. An example of how this works is if you have five points in a subtour, then you have five lines connecting those points to create the subtour. Eliminate this subtour by implementing an inequality constraint to say there must be less than or equal to four lines between these five points.

Even more, find all lines between these five points, and constrain the solution not to have more than four of these lines present. This is a correct constraint because if five or more of the lines existed in a solution, then the solution would have a subtour (a graph with  $n$  nodes and  $n$  edges always contains a cycle).

The `detectSubtours` function analyzes the solution and returns a cell array of vectors. Each vector in the cell array contains the stops involved in that particular subtour.

```
tours = detectSubtours(tpsol.trips,idxs);
numtours = length(tours); % 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.

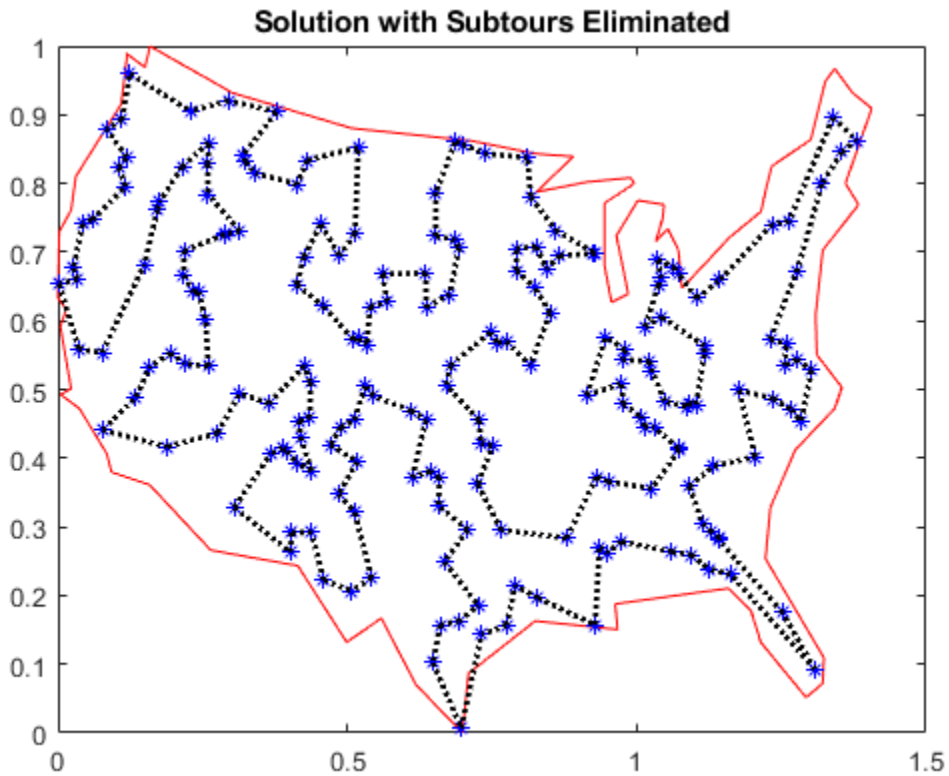
```
% Index of added constraints for subtours
k = 1;
while numtours > 1 % repeat until there is just one subtour
    % Add the subtour constraints
    for ii = 1:numtours
        subTourIdx = tours{ii}; % Extract the current subtour
        % The next lines find all of the variables associated with the
        % particular subtour, then add an inequality constraint to prohibit
        % that subtour and all subtours that use those stops.
        variations = nchoosek(1:length(subTourIdx),2);
        a = false(length(idxs),1);
        for jj = 1:length(variations)
            whichVar = (sum(idxs==subTourIdx(variations(jj,1)),2)) & ...
                (sum(idxs==subTourIdx(variations(jj,2)),2));
            a = a | whichVar;
        end
        tsp.Constraints.(sprintf('subtourconstr%i',k)) = sum(trips(a)) <= length(subTourIdx);
        k = k + 1;
    end
    % Try to optimize again
    [tpsol,fval,exitflag,output] = solve(tsp,'options',opts);
```

```
% Visualize result
lh = updateSalesmanPlot(lh, tpsol.trips, idxs, stopsLon, stopsLat);

% How many subtours this time?
tours = detectSubtours(tpsol.trips, idxs);
numtours = length(tours); % 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 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.absolute_gap)
```

```
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 9-64

## 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.

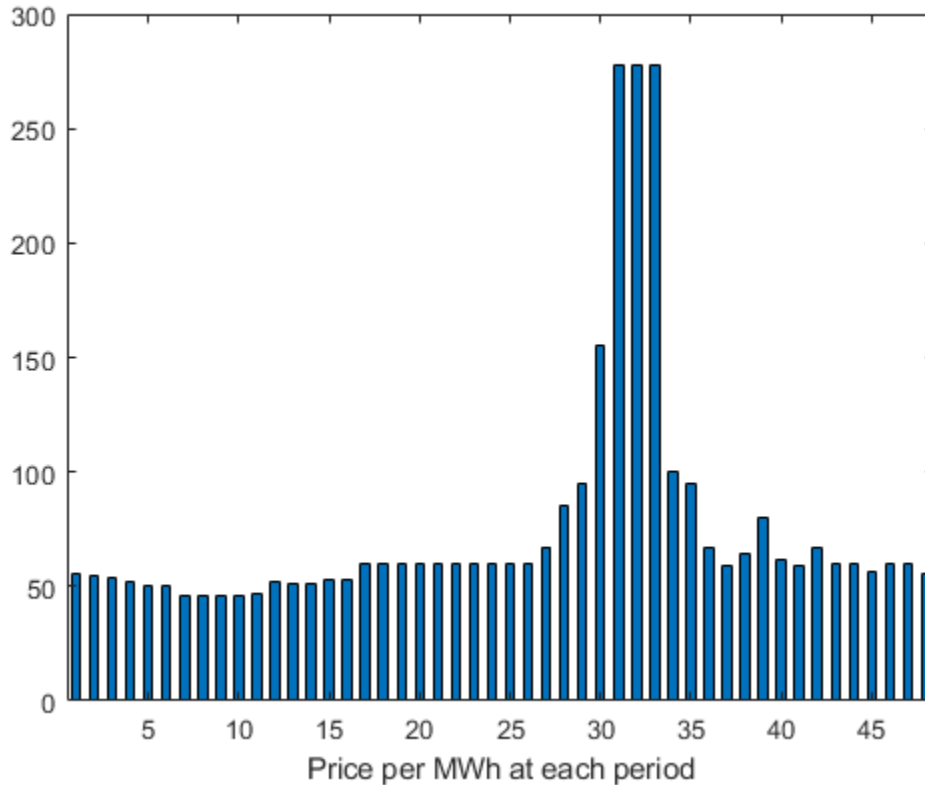
For the solver-based approach to this problem, see “Optimal Dispatch of Power Generators: Solver-Based” on page 9-73.

### 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\\_Notice](https://www.aemo.com.au/Privacy_and_Legal_Notices/Copyright_Permissions_Notice).

```
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. 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:

- $n\text{Periods}$  = the number of time periods, 48 in this case.
- $i$  = a time period,  $1 \leq i \leq 48$ .
- $j$  = a generator index,  $1 \leq j \leq 2$  for this example.

- $y(i, j, k) = 1$  when period  $i$ , generator  $j$  is operating at power level  $k$ . Let low power be  $k = 1$ , and high power be  $k = 2$ . The generator is off when  $\sum_k y(i, j, k) = 0$ .

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
```

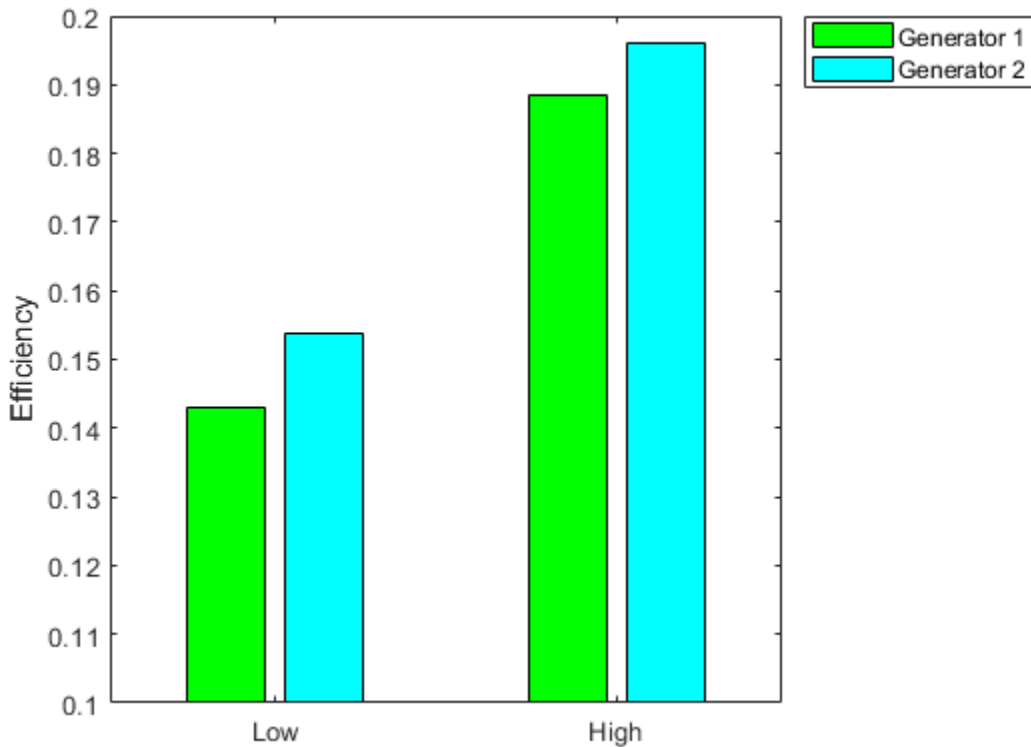
### Generator Efficiency

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  $n$ Periods-by- $n$ Gens-by-2 array, and  $z$  is an  $n$ Periods-by- $n$ Gens 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.

$$\text{sum\_k} ( - y(i,j,k) + y(i+1,j,k) ) - z(i,j) < = 0.$$

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 =  $y.*\text{generatorlevel}.*\text{poolPrice}$ .

```
revenue = optimexpr(size(y));
for ii = 1:nPeriods
    revenue(ii,:,:)= poolPrice(ii)*y(ii,:,:).*generatorlevel(ii,:,:);
end
```

The total fuel cost =  $\text{fuelUsed}*\text{fuelPrice}$ .

```
fuelCost = fuelUsed*fuelPrice;
```

The generator startup cost =  $z*\text{startCost}$ .

```
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);
```

Optimal solution found.

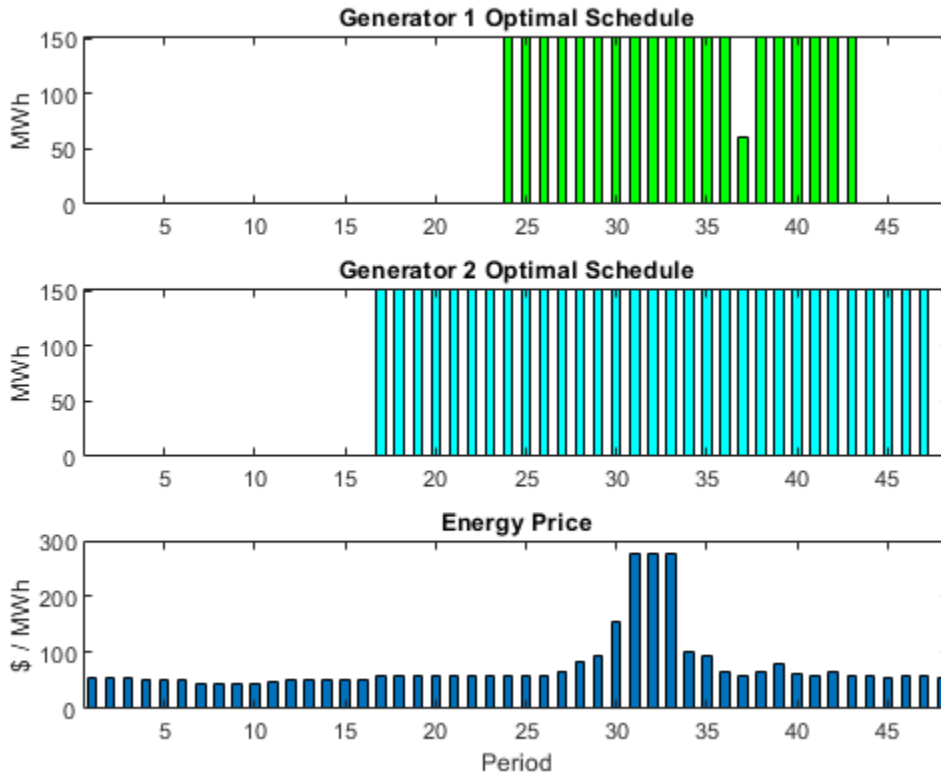
Intlinprog stopped because the objective value is within a gap tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the default value).

### 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
[thepperiod,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);
```

Optimal solution found.

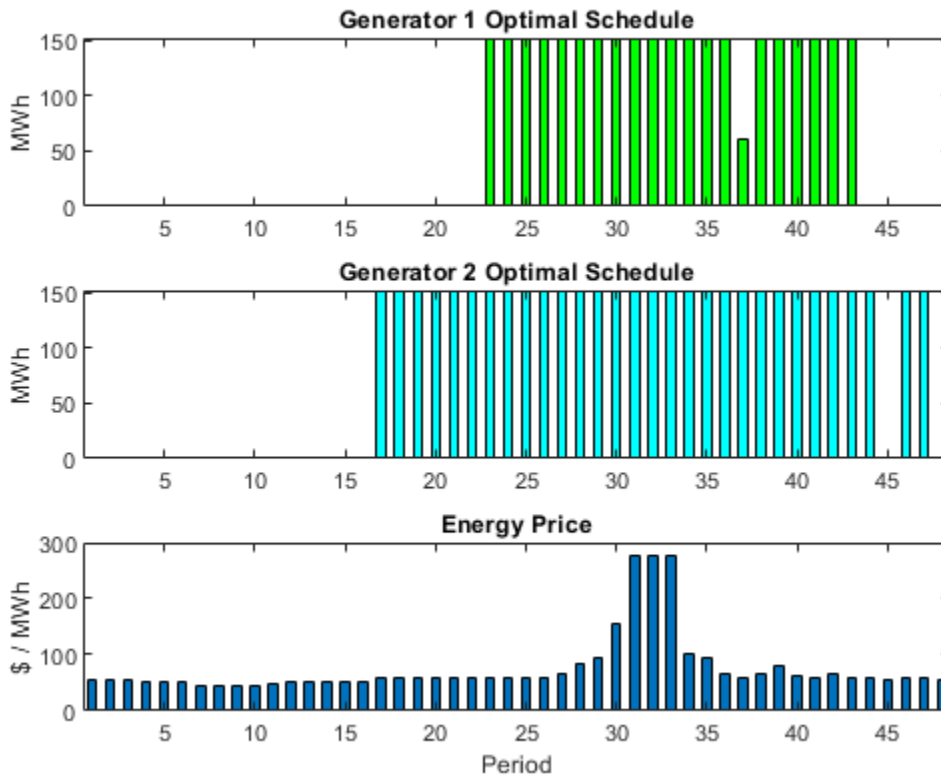
`Intlinprog` stopped because the objective value is within a gap tolerance of the optimal value, `options.AbsoluteGapTolerance = 0` (the default value). The `intcon` variables are integer within tolerance, `options.IntegerTolerance = 1e-05` (the default value).

```
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')

```



```

starttimes = find(round(dispatchsolnew.z) == 1); % Use round for noninteger results
[thepperiod,thegenerator] = ind2sub(size(dispatchsolnew.z),starttimes)

```

```
thepperiod = 3×1
```

```

22
16
45

```

```
thegenerator = 3×1
```

```
1  
2  
2
```

### See Also

#### More About

- “Optimal Dispatch of Power Generators: Solver-Based” on page 9-73



## 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 9-104.

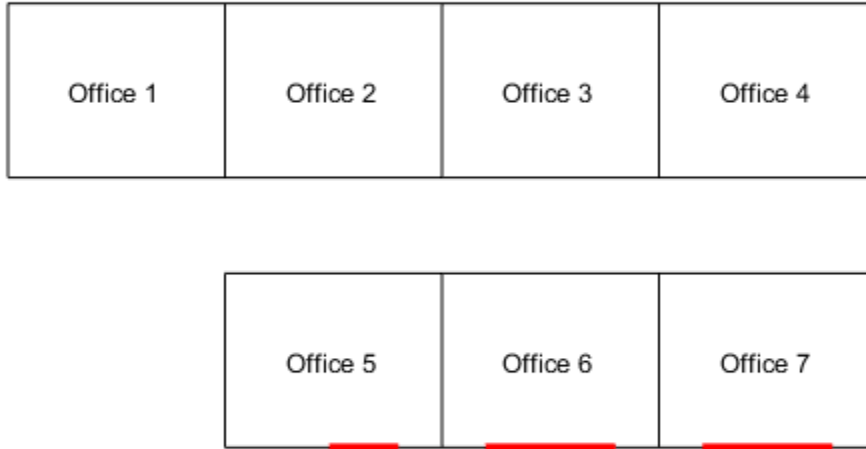
### Office Assignment Problem

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  $i$ th element of a person's preference vector is how highly they value the  $i$ th 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',:)) -
peopleprob.Constraints.constrpt2 = occupy('Tom','Office 2') + sum(occupy('Peter',:)) -
    - occupy('Peter','Office 3') - occupy('Peter','Office 5') <= 1;
peopleprob.Constraints.constrpt3 = occupy('Tom','Office 3') + sum(occupy('Peter',:)) -
    - occupy('Peter','Office 4') - occupy('Peter','Office 6') <= 1;
peopleprob.Constraints.constrpt4 = occupy('Tom','Office 4') + sum(occupy('Peter',:)) -
    - occupy('Peter','Office 7') <= 1;
peopleprob.Constraints.constrpt5 = occupy('Tom','Office 5') + sum(occupy('Peter',:)) -
    - occupy('Peter','Office 6') <= 1;
peopleprob.Constraints.constrpt6 = occupy('Tom','Office 6') + sum(occupy('Peter',:)) -
    - occupy('Peter','Office 5') - occupy('Peter','Office 7') <= 1;
peopleprob.Constraints.constrpt7 = occupy('Tom','Office 7') + sum(occupy('Peter',:)) -
    - 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',:)) -
peopleprob.Constraints.constmr2 = occupy('Marcelo','Office 2') + sum(occupy('Rakesh',:)) -
    - occupy('Rakesh','Office 3') - occupy('Rakesh','Office 5') <= 1;
peopleprob.Constraints.constmr3 = occupy('Marcelo','Office 3') + sum(occupy('Rakesh',:)) -
    - occupy('Rakesh','Office 4') - occupy('Rakesh','Office 6') <= 1;
peopleprob.Constraints.constmr4 = occupy('Marcelo','Office 4') + sum(occupy('Rakesh',:)) -
    - occupy('Rakesh','Office 7') <= 1;
peopleprob.Constraints.constmr5 = occupy('Marcelo','Office 5') + sum(occupy('Rakesh',:)) -
    - occupy('Rakesh','Office 6') <= 1;
peopleprob.Constraints.constmr6 = occupy('Marcelo','Office 6') + sum(occupy('Rakesh',:)) -
    - occupy('Rakesh','Office 5') - occupy('Rakesh','Office 7') <= 1;
```

```
peopleprob.Constraints.constmr7 = occupy('Marcelo','Office 7') + sum(occupy('Rakesh',:
    - 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. `options.AbsoluteGapTolerance = 0` (the default value). The `intcon` variables are integer. `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
    solver: 'intlinprog'
```

## See Also

### More About

- “Office Assignments by Binary Integer Programming: Solver-Based” on page 9-104

## 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 9-86.

### 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  $\lambda$  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  $M \leq N$ .
- Having at least  $m$  assets in the portfolio, where  $0 < m \leq M$ .
- Having *semicontinuous* constraints, meaning either  $x(i) = 0$ , or  $f_{\min} \leq x(i) \leq f_{\max}$  for some fixed fractions  $f_{\min} > 0$  and  $f_{\max} \geq f_{\min}$ .

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  $0 \leq x(i) \leq 1$  for each  $i$ . To model the number of assets in the portfolio, you need indicator variables  $v$  such that  $v(i) = 0$  when  $x(i) = 0$ , and  $v(i) = 1$  when  $x(i) > 0$ . To get variables that satisfy this restriction, set the  $v$  vector to be a binary variable, and impose the linear constraints

$$v(i)f_{\min} \leq x(i) \leq v(i)f_{\max}.$$

These inequalities both enforce that  $x(i)$  and  $v(i)$  are zero at exactly the same time, and they also enforce that  $f_{\min} \leq x(i) \leq f_{\max}$  whenever  $x(i) > 0$ .

Also, to enforce the constraints on the number of assets in the portfolio, impose the linear constraints

$$m \leq \sum_i v(i) \leq M.$$

## Objective and Successive Linear Approximations

As first formulated, you try to maximize the objective function. However, all Optimization Toolbox™ solvers minimize. So formulate the problem as minimizing the negative of the objective:

$$\min_x \lambda x^T Q x - r^T x.$$

This objective function is nonlinear. The 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.

$$\min_{x,z} \lambda z - r^T x \text{ such that } x^T Q x - z \leq 0, \quad z \geq 0.$$

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  $\delta$  is a variable vector, the first-order Taylor approximation to the constraint is

$$x^T Q x - z = x_0^T Q x_0 + 2x_0^T Q \delta - z + O(|\delta|^2).$$

Replacing  $\delta$  by  $x - x_0$  gives

$$x^T Q x - z = -x_0^T Q x_0 + 2x_0^T Q x - z + O(|x - x_0|^2).$$

For each intermediate solution  $x_k$  you introduce a new linear constraint in  $x$  and  $z$  as the linear part of the expression above:

$$-x_k^T Q x_k + 2x_k^T Q x - z \leq 0.$$

This has the form  $Ax \leq b$ , where  $A = 2x_k^T Q$ , there is a  $-1$  multiplier for the  $z$  term, and  $b = x_k^T Q x_k$ .

This method of adding new linear constraints to the problem is called a cutting plane method. For details, see J. E. Kelley, Jr. "The Cutting-Plane Method for Solving Convex Programs." J. Soc. Indust. Appl. Math. Vol. 8, No. 4, pp. 703-712, December, 1960.

### **MATLAB® Problem Formulation**

To express 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  $vvars$  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

$$m \leq \sum_i v(i) \leq M,$$

by writing two linear constraints:

$$\sum_i v(i) \leq M$$

$$\sum_i v(i) \geq m.$$

```
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  $x(i) \leq fmax(i) * v(i)$  and  $fmin(i) * v(i) \leq x(i)$ .

```
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  $\lambda$  to 100.

```
lambda = 100;
```

Define the objective function  $r^T x - \lambda z$  and include it in the problem.

```
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.

```
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.

```
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.

```
history = [truequadratic, zslack];
```

```
options = optimoptions(options, 'LPoptimalityTolerance', 1e-10, 'RelativeGapTolerance', 1e-10,  
                        '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  $Ax \leq b$  comes from the linear approximation

$$-x_k^T Q x_k + 2x_k^T Q x - z \leq 0.$$

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.

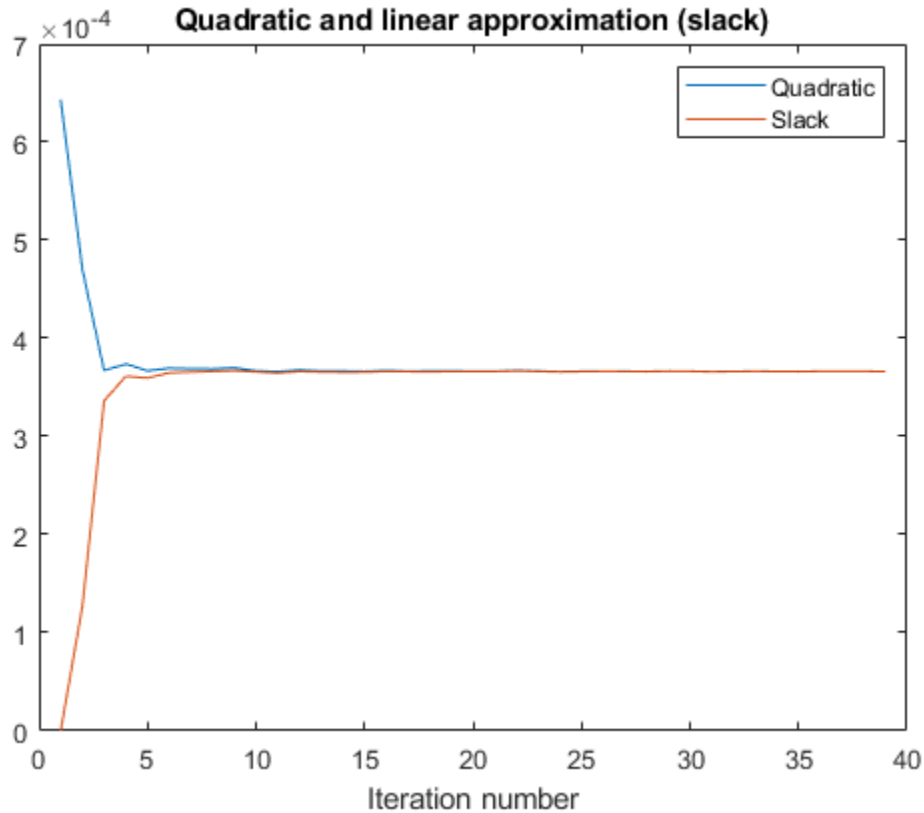
```
while abs((zslack - truequadratic)/truequadratic) > thediff % relative error  
    constr = 2*assets'*Q*xvars - zvar <= assets'*Q*assets;
```

```
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)
```

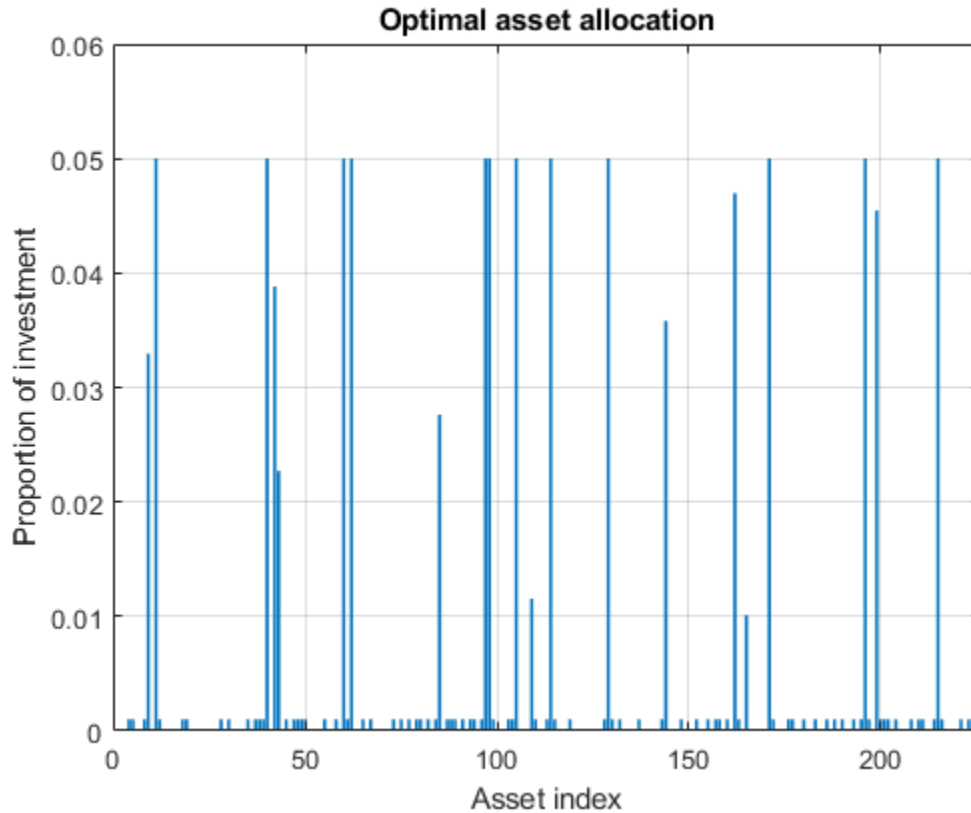
```
0
```

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')
```



You can easily see that all nonzero asset allocations are between the semicontinuous bounds  $f_{\min} = 0.001$  and  $f_{\max} = 0.05$ .

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 9-86



## 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 9-113.

### 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.



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.

```
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
    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
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.

```

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 ov
        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
        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 1 logs with pattern
    3 cut(s) of length 12
    Waste of this pattern is 4
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 36 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 9-113

## 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 9-95.

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.
```

	2			3			4	
6								3
		4				5		
			8		6			
8				1				6
			7		5			
		7				6		
4								8
	3			4			2	

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  $x(i, j, 1), \dots, x(i, j, 9)$ . 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)
```

```
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 9-95



# Problem-Based Optimization

---

- “Problem-Based Workflow” on page 10-2
- “Optimization Expressions” on page 10-4
- “Pass Extra Parameters in Problem-Based Approach” on page 10-10
- “Review or Modify Optimization Problems” on page 10-14
- “Named Index for Optimization Variables” on page 10-22
- “Examine Optimization Solution” on page 10-28
- “Create Efficient Optimization Problems” on page 10-32
- “Separate Optimization Model from Data” on page 10-34
- “Problem-Based Optimization Algorithms” on page 10-36
- “Variables with Duplicate Names Disallowed” on page 10-38
- “Expression Contains Inf or NaN” on page 10-40
- “Supported Operations on Optimization Variables and Expressions” on page 10-42
- “Mixed-Integer Linear Programming Basics: Problem-Based” on page 10-45
- “Create Initial Point for Optimization with Named Index Variables” on page 10-49

## Problem-Based Workflow

---

**Note** There are two different Optimization Toolbox approaches for solving single-objective optimization problems. This section 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 a problem object by using `optimproblem`. A problem object is a container that you fill with objective and constraint expressions. These expressions define the problem, along with 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 a polynomial or rational expression, convert it to an optimization expression by using `fcn2optimexpr`. See “Convert Nonlinear Function to Optimization Expression” on page 7-8.

---

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 in the problem object as expressions in the named variables.



---

**Note** If you have a nonlinear function that is not a polynomial or rational expression, convert it to an optimization expression by using `fcn2optimexpr`. See “Convert Nonlinear Function to Optimization Expression” on page 7-8.

---

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;
```

- Solve the optimization problem by using `solve`.

```
sol = solve(prob);
```

In addition to these basic steps, you can review the problem definition before solving the problem by using `showproblem` and related functions. Set options for `solve` by using `optimoptions`, as explained in “Change Default Solver or Options” on page 10-15.

For a basic mixed-integer linear programming example, see “Mixed-Integer Linear Programming Basics: Problem-Based” on page 10-45 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”.

## See Also

`fcn2optimexpr` | `optimproblem` | `optimvar` | `solve`

## More About

- “Mixed-Integer Linear Programming Basics: Problem-Based” on page 10-45
- *Solve a Mixed-Integer Linear Programming Problem using Optimization Modeling*
- “Optimization Expressions” on page 10-4
- “Review or Modify Optimization Problems” on page 10-14
- “Examine Optimization Solution” on page 10-28

## Optimization Expressions

### In this section...

“What Are Optimization Expressions?” on page 10-4  
 “Expressions for Objective Functions” on page 10-4  
 “Expressions for Constraints” on page 10-6  
 “Optimization Variables Have Handle Behavior” on page 10-8

### 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
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 10-42.

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 10-40.

### 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 a polynomial or rational expression, convert it to an optimization expression by using `fcn2optimexpr`. See “Convert Nonlinear Function to Optimization Expression” on page 7-8.

---

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
showexpr(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,
- 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,
```

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));
showexpr(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 Least-Squares, Problem-Based” on page 12-50.

---

## Expressions for Constraints

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.

```
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 a polynomial or rational expression, convert it to an optimization expression by using `fcn2optimexpr`. See “Convert Nonlinear Function to Optimization Expression” on page 7-8.

---

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;
```

You can also include constraints 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 expressions in a loop, start with an empty constraint expression as returned by `optimconstr`.

```
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
showconstr(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))';
showconstr(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.

---

**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.

```
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;
```

---

## Optimization Variables Have Handle Behavior

- `OptimizationVariable` objects have *handle* copy behavior. See “Handle Object Behavior” (MATLAB) and “Comparison of Handle and Value Classes” (MATLAB). 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` | `optimvar` | `showconstr` | `showexpr` | `showproblem` | `writeconstr` | `writeexpr`

## **More About**

- “Mixed-Integer Linear Programming Basics: Problem-Based” on page 10-45
- “Problem-Based Workflow” on page 10-2
- “Review or Modify Optimization Problems” on page 10-14
- “Named Index for Optimization Variables” on page 10-22

## 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)
```

```
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. Convert `parameterfun` to an optimization expression and set the converted expression as the objective.

```
prob = optimproblem;  
expr = fcn2optimexpr(@parameterfun,x,a,b,c);  
prob.Objective = expr;
```

Solve the problem starting from the point  $x_0$ .  $x = [1/2; 1/2]$ .

```
x0.x = [1/2;1/2];  
[sol,fval] = solve(prob,x0)
```

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
```

Because this objective function is a rational function of  $x$ , you can specify the objective in terms of the optimization variable, without using `fcn2optimexpr`. Either way, you include the extra parameters simply by referring to them in the objective function.

```
prob.Objective = (a - b*x(1)^2 + x(1)^4/3)*x(1)^2 + x(1)*x(2)...  
  + (-c + c*x(2)^2)*x(2)^2;  
[sol,fval] = solve(prob,x0)
```

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
```

## See Also

`fcn2optimexpr`

## **More About**

- “Passing Extra Parameters” on page 2-70

## Review or Modify Optimization Problems

### In this section...

“Review Problem Using `showproblem` or `writeproblem`” on page 10-14

“Change Default Solver or Options” on page 10-15

“Correct a Misspecified Problem” on page 10-17

“Duplicate Variable Name” on page 10-20

### Review Problem Using `showproblem` or `writeproblem`

After you create an optimization problem, you can review its formulation by using `showproblem`. For large problems, use `writeproblem` 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;
```

```
showproblem(prob)
```

```
OptimizationProblem :
    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. 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 OptimizationVariable array with properties:  
  
Array-wide properties:  
    Name: 'x'  
    Type: 'continuous'  
    IndexNames: {} {} {}  
  
Elementwise properties:  
    LowerBound: [9×9 double]  
    UpperBound: [9×9 double]  
  
See variables with showvar.  
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 OptimizationVariable array with properties:  
  
Array-wide properties:  
    Name: 'x'  
    Type: 'integer'  
    IndexNames: {} {} {}  
  
Elementwise properties:  
    LowerBound: [9×9 double]
```



```
UpperBound: [9×9×9 double]
```

See variables with `showvar`.  
See bounds with `showbounds`.

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.

```
writeexpr(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(9, 1, 1) + x(1, 2, 1) + x(2, 2, 1) + x(3, 2, 1) + x(4, 2, 1) + x(5, 2, 1)
...
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.

```
writeconstr(prob.Constraints.cons1, 'cons1.txt')
writeconstr(prob.Constraints.cons2, 'cons2.txt')
writeconstr(prob.Constraints.cons3, 'cons3.txt')
writeconstr(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 10-38.

## See Also

OptimizationConstraint | OptimizationExpression | OptimizationProblem | OptimizationVariable | showbounds | showconstr | showexpr | showproblem | showvar | writebounds | writeconstr | writeexpr | writeproblem | writevar

## **More About**

- “Problem-Based Workflow” on page 10-2
- “Mixed-Integer Linear Programming Basics: Problem-Based” on page 10-45

## Named Index for Optimization Variables

In this section...
"Create Named Indices" on page 10-22
"Use Named Indices" on page 10-23
"View Solution with Index Variables" on page 10-25

### 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 showvar.
```

```
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.

```
showvar(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 array
    {'row1'}    {'importantRow'}    {'row3'}
```

```
x.IndexNames{2}

ans = 1x2 cell array
    {'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;
showconstr(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];
showconstr(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);

```

LP: Optimal objective value is -1027.472366.

Heuristics: Found 1 solution using rounding.

```
Upper bound is -1027.233133.
Relative gap is 0.00%.
```

```
Cut Generation: Applied 1 mir cut, and 2 strong CG cuts.
Lower 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 = 1x2
```

```
2 4
```

```
idxAirports = 1x2
```

```
1 3
```

```
orangeBerries = sol.flow(idxFruit, idxAirports)
```

```
orangeBerries = 2x2
```

```
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 Workflow” on page 10-2
- “Create Initial Point for Optimization with Named Index Variables” on page 10-49

## Examine Optimization Solution

### In this section...

“Obtain Numeric Solution” on page 10-28

“Examine Solution Quality” on page 10-29

“Infeasible Solution” on page 10-29

“Solution Takes Too Long” on page 10-30

### 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)
```

```
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)
```

```
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 10-32.
- *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 9-45.
- *Nonlinear programming solution is slow.* For suggestions, see “Solver Takes Too Long” on page 4-11. 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 9-45
- “Exit Flags and Exit Messages” on page 3-3
- “Output Structures” on page 3-26
- “Lagrange Multiplier Structures” on page 3-27
- “Mixed-Integer Linear Programming Basics: Problem-Based” on page 10-45

## 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 9-45.

Before you start solving the problem, sometimes you can improve the formulation of your problem constraints or objective. Usually, it is faster for the software to create expressions for objective function or constraints in a vectorized fashion rather than in a loop. 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. In this case,

```
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
```

Here, `expr` contains the objective function expression. While this method is straightforward, it can take 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  $x$ .

```
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)));
```

- Loop once over  $b$ .

```
expr = optimexpr;  
for k = 1:10  
    expr = expr + sum(sum(x(:,:,k).*c))*b(k);  
end
```

- Create an expression differently by looping over b and then summing terms after the loop.

```
expr = optimexpr(30,30,10);  
for k = 1:10  
    expr(:,:,k) = x(:,:,k).*c*b(k);  
end  
expr = sum(expr(:));
```

## See Also

### More About

- “Tuning Integer Linear Programming” on page 9-45
- “Separate Optimization Model from Data” on page 10-34

## 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(yr) = sum(costs(yr,:).*usage(yr,:));
    totalcost = totalcost + costPerYear(yr);
end
```

## See Also

### More About

- “Problem-Based Workflow” on page 10-2
- “Create Efficient Optimization Problems” on page 10-32
- “Factory, Warehouse, Sales Allocation Model: Problem-Based”
- “Traveling Salesman Problem: Problem-Based”
- “Create Multiperiod Inventory Model in Problem-Based Framework”

## 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
- `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

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 16-444.

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 9-33. 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 the algorithms that `lsqlin` uses to solve linear least-squares problems, 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 6-22.

**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 Least-Squares, Problem-Based” on page 12-50.

---

## See Also

`intlinprog` | `linprog` | `prob2struct`

## More About

- “`intlinprog` Algorithm” on page 9-33
- “Linear Programming Algorithms” on page 9-2
- “Create Efficient Optimization Problems” on page 10-32

## 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.

```
oproblem = optimproblem('Constraints',cons);  
x = oproblem.Variables.x;  
x.Type = 'integer';  
oproblem.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 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  $\text{Inf} - \text{Inf} = \text{NaN}$ , 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 `showexpr`, `showconstr`, `showproblem`, or the equivalent `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

`showconstr` | `showexpr` | `showproblem` | `writeconstr` | `writeexpr` | `writeproblem`

## More About

- “Problem-Based Workflow” on page 10-2
- “Review or Modify Optimization Problems” on page 10-14

## Supported Operations on Optimization Variables and Expressions

### In this section...

“Notation for Supported Operations” on page 10-42

“Operations Returning Optimization Expressions” on page 10-42

“Operations Returning Optimization Variables” on page 10-43

“Operations on Optimization Expressions” on page 10-43

“Operations Returning Constraint Expressions” on page 10-44

“Some Undocumented Operations Work on Optimization Variables and Expressions” on page 10-44

### Notation for Supported Operations

For the legal operations on optimization variables and expressions:

- $x_1$  and  $x_2$  represent optimization arrays of arbitrary size (usually the same size).
- $x_3$  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  $x_1$ .

### Operations Returning Optimization Expressions

These operations on optimization variables or expressions return an optimization expression.

- Addition of a constant,  $x_1+c$  or  $c+x_1$
- Addition of two or more variables,  $x_1+x_2$
- Unary plus,  $+x_1$
- Subtraction of a constant,  $x_1-c$
- Subtraction of two or more variables,  $x_1-x_2$
- Unary minus,  $-x_1$



- Multiplication by a constant scalar,  $a \times x1$  or  $a .* x1$  or  $x1 * a$  or  $x1 .* a$
- Division by a constant scalar,  $x1/a$  or  $x1 ./ a$  or  $a \backslash x1$  or  $a . \backslash x1$
- Pointwise multiplication by an array,  $c .* x1$  or  $x1 .* c$
- Pointwise division by an array,  $x1 ./ c$  or  $c . \backslash x1$
- Pointwise multiplication of two optimization variables,  $x1 .* x2$
- Multiplication of two optimization variables of sizes that make sense for matrix multiplication,  $x1 * x2$
- Raising a 2-D variable with the same number of rows as columns to a power,  $x1^a$
- Pointwise raising a variable to a power,  $x1.^a$
- Multiplication of a 2-D variable by a 2-D matrix,  $M * x3$  or  $x3 * M$
- Dot product of a variable and an array  $\text{dot}(x1, c)$  or  $\text{dot}(c, x1)$
- Linear combination of variables, including  $\text{sum}$ ,  $\text{sum}(x1)$ ,  $\text{sum}(x1, \text{dim})$ ,  $\text{mean}(x1)$ , and  $\text{mean}(x1, \text{dim})$
- Transpose operation,  $x1'$  or  $x1.'$
- Concatenation ( $\text{cat}$ ,  $\text{vertcat}$ , and  $\text{horzcat}$ )
- Reshape operation,  $\text{reshape}(x1, [10 \ 1])$

## Operations Returning Optimization Variables

These operations on optimization variables return an optimization variable.

- N-D numeric indexing (includes colon and end)
- N-D logical indexing
- N-D string indexing
- N-D mixed indexing (combination of numeric, logical, colon, end, and string)
- Linear numeric indexing (includes colon and end)
- Linear logical indexing
- Linear string indexing

## 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” on page 10-6.

## **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.

## **See Also**

`OptimizationExpression` | `OptimizationVariable`

## **More About**

- “Problem-Based Optimization Setup”
- “Problem-Based Workflow” on page 10-2
- “Optimization Expressions” on page 10-4

## Mixed-Integer Linear Programming Basics: Problem-Based

This example shows how to solve a mixed-integer linear program. The example is not complex, but it shows typical steps in formulating a problem for 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 9-48.

### 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  $25 \text{ tons} \times 5\% = 1.25 \text{ tons}$  of carbon and 1.25 tons of molybdenum. The objective is to minimize the cost for blending the steel.

This problem is taken from Carl-Henrik Westerberg, Bengt Bjorklund, and Eskil Hultman, “*An Application of Mixed Integer Programming in a Swedish Steel Mill.*” *Interfaces* February 1977 Vol. 7, No. 2 pp. 39-43, whose abstract is at <http://interfaces.journal.informs.org/content/7/2/39.abstract>.

Four ingots of steel are available for purchase. Only one of each ingot is available.

Ingot	Weight in Tons	%Carbon	%Molybdenum	Cost/Ton
1	5	5	3	\$350
2	3	4	3	\$330
3	4	5	4	\$310
4	6	3	4	\$280

Three grades of alloy steel are available for purchase, and one grade of scrap steel. Alloy and scrap steels can be purchased in fractional amounts.

Alloy	%Carbon	%Molybdenum	Cost/Ton
1	8	6	\$500

Alloy	%Carbon	%Molybdenum	Cost/Ton
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 in tons of scrap steel that you purchase.

Create the optimization problem and the variables.

```
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;
```

There are 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);
```

```
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
sol.alloys
sol.scrap
fval

```

```
ans =
```

```
1  
1  
0  
1
```

```
ans =
```

```
7.2500  
0  
0.2500
```

```
ans =
```

```
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.

## See Also

### More About

- “Mixed-Integer Linear Programming Basics: Solver-Based” on page 9-48
- “Problem-Based 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*buy));
```

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” on page 10-6.

```
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');
[soll,fvall,exitstatus1,output1] = solve(prob,'options',opts)
```

```
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 -1.046990e+05.
```

```
Branch and Bound:
```

nodes explored	total time (s)	num int solution	integer fval	relative gap (%)
28	0.16	1	-9.945833e+04	4.818992e+00
70	0.21	2	-9.998889e+04	4.262813e+00
114	0.23	3	-1.002787e+05	3.891142e+00
1086	0.53	3	-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).

```
soll = struct with fields:
    buy: [6x5 double]
    induse: [6x5 double]
    sell: [6x5 double]
    store: [6x5 double]
```

```
fvall = 1.0028e+05
```

```
exitstatus1 =
    OptimalSolution
```

```
output1 = struct with fields:
    relativegap: 0
    absolutegap: 0
    numfeaspoints: 3
    numnodes: 1086
    constrviolation: 1.5111e-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)
```

```
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 -1.046990e+05.
```

Relative gap is 166.74%.

Branch and Bound:

nodes explored	total time (s)	num int solution	integer fval	relative gap (%)
28	0.13	2	-9.945833e+04	4.818992e+00
66	0.16	3	-9.987222e+04	4.384608e+00
159	0.20	4	-9.996667e+04	4.120028e+00
166	0.20	5	-1.001917e+05	3.886209e+00
249	0.23	6	-1.002139e+05	2.927655e+00
408	0.28	7	-1.002787e+05	2.538777e+00
1107	0.47	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: 1107  
    constrviolation: 3.6096e-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 1107 branch-and-bound steps,  
but using no initial point took 1086 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 10-22
- “Problem-Based Workflow” on page 10-2





# Quadratic Programming

---

- “Quadratic Programming Algorithms” on page 11-2
- “Quadratic Minimization with Bound Constraints” on page 11-15
- “Quadratic Minimization with Dense, Structured Hessian” on page 11-18
- “Large Sparse Quadratic Program with Interior Point Algorithm” on page 11-24
- “Bound-Constrained Quadratic Programming, Solver-Based” on page 11-29
- “Quadratic Programming for Portfolio Optimization Problems, Solver-Based” on page 11-34
- “Quadratic Programming with Bound Constraints: Problem-Based” on page 11-42
- “Large Sparse Quadratic Program, Problem-Based” on page 11-46
- “Bound-Constrained Quadratic Programming, Problem-Based” on page 11-50
- “Quadratic Programming for Portfolio Optimization, Problem-Based” on page 11-55

## Quadratic Programming Algorithms

<b>In this section...</b>
---------------------------

“Quadratic Programming Definition” on page 11-2
---

“interior-point-convex quadprog Algorithm” on page 11-2
---

“trust-region-reflective quadprog Algorithm” on page 11-9
---

### Quadratic Programming Definition

Quadratic programming is the problem of finding a vector  $x$  that minimizes a quadratic function, possibly subject to linear constraints:

$$\min_x \frac{1}{2}x^T Hx + c^T x$$

such that  $A \cdot x \leq b$ ,  $Aeq \cdot x = beq$ ,  $l \leq x \leq u$ .

### interior-point-convex quadprog Algorithm

The interior-point-convex algorithm performs the following steps:

1. “Presolve/Postsolve” on page 11-3
2. “Generate Initial Point” on page 11-3
3. “Predictor-Corrector” on page 11-4
4. “Stopping Conditions” on page 11-8
5. “Infeasibility Detection” on page 11-9

---

**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” (MATLAB). 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 begins by attempting to simplify the problem by removing redundancies and simplifying constraints. The tasks performed during the presolve step include:

- Check if any variables have equal upper and lower bounds. If so, check for feasibility, and then fix and remove the variables.
- Check if any linear inequality constraint involves just one variable. If so, check for feasibility, and change the linear constraint to a bound.
- Check if any linear equality constraint involves just one variable. If so, check for feasibility, and then fix and remove the variable.
- Check if any linear constraint matrix has zero rows. If so, check for feasibility, and delete the rows.
- Check if the bounds and linear constraints are consistent.
- Check if any variables appear only as linear terms in the objective function and do not appear in any linear constraint. If so, check for feasibility and boundedness, and fix the variables at their appropriate bounds.
- Change any linear inequality constraints to linear equality constraints by adding slack variables.

If algorithm detects an infeasible or unbounded problem, it halts and issues an appropriate exit message.

The algorithm might arrive at a single feasible point, which represents the solution.

If the algorithm does not detect an infeasible or unbounded problem in the presolve step, it continues, if necessary, with the other steps. At the end, the algorithm reconstructs the original problem, undoing any presolve transformations. This final step is the postsolve step.

For details, see Gould and Toint [63].

### Generate Initial Point

The initial point  $x_0$  for the algorithm is:

- 1 Initialize  $x_0$  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  $x_0$  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 11-4), 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  $Ax \leq b$  into inequalities of the form  $Ax \geq b$  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 11-4
- “Full Predictor-Corrector” on page 11-6

**Sparse Predictor-Corrector**

Similar to the `fmincon` interior-point algorithm on page 6-37, the sparse interior-point-convex algorithm tries to find a point where the Karush-Kuhn-Tucker (KKT) on page 3-13 conditions hold. For the quadratic programming problem described in “Quadratic Programming Definition” on page 11-2, these conditions are:

$$\begin{aligned}
 Hx + c - A_{eq}^T y - \bar{A}^T z &= 0 \\
 \bar{A}x - \bar{b} - s &= 0 \\
 A_{eq}x - b_{eq} &= 0 \\
 s_i z_i &= 0, \quad i = 1, 2, \dots, m \\
 s &\geq 0 \\
 z &\geq 0.
 \end{aligned}$$

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:

$$\begin{pmatrix} H & 0 & -A_{eq}^T & -\bar{A}^T \\ A_{eq} & 0 & 0 & 0 \\ \bar{A} & -I & 0 & 0 \\ 0 & Z & 0 & S \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta s \\ \Delta y \\ \Delta z \end{pmatrix} = - \begin{pmatrix} r_d \\ r_{eq} \\ r_{ineq} \\ r_{sz} \end{pmatrix}. \quad (11-1)$$

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  $\sigma$ , 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

$$\begin{aligned} Hx + c - \bar{A}^T y - v + w &= 0 \\ \bar{A}x &= \bar{b} \\ x + t &= u \\ v_i x_i &= 0, \quad i = 1, 2, \dots, m \\ w_i t_i &= 0, \quad i = 1, 2, \dots, n \\ x, v, w, t &\geq 0. \end{aligned} \tag{11-2}$$

To find the solution  $x$ , slack variables and dual variables to “Equation 11-2”, the algorithm basically considers a Newton-Raphson step:

$$\begin{pmatrix} H & -\bar{A}^T & 0 & -I & I \\ \bar{A} & 0 & 0 & 0 & 0 \\ -I & 0 & -I & 0 & 0 \\ V & 0 & 0 & X & 0 \\ 0 & 0 & W & 0 & T \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta t \\ \Delta v \\ \Delta w \end{pmatrix} = - \begin{pmatrix} Hx + c - \bar{A}^T y - v + w \\ \bar{A}x - \bar{b} \\ u - x - t \\ VX \\ WT \end{pmatrix} = - \begin{pmatrix} r_d \\ r_p \\ r_{ub} \\ r_{vx} \\ r_{wt} \end{pmatrix}, \quad (11-3)$$

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 11-3” by first converting it to the symmetric matrix form

$$\begin{pmatrix} -D & \bar{A}^T \\ \bar{A} & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = - \begin{pmatrix} R \\ r_p \end{pmatrix}, \quad (11-4)$$

where

$$\begin{aligned} D &= H + X^{-1}V + T^{-1}W \\ R &= -r_d - X^{-1}r_{vx} + T^{-1}r_{wt} + T^{-1}Wr_{ub}. \end{aligned}$$

All the matrix inverses in the definitions of  $D$  and  $R$  are simple to compute because the matrices are diagonal.

To derive “Equation 11-4” from “Equation 11-3”, notice that the second row of “Equation 11-4” is the same as the second matrix row of “Equation 11-3”. The first row of “Equation 11-4” comes from solving the last two rows of “Equation 11-3” for  $\Delta v$  and  $\Delta w$ , and then solving for  $\Delta t$ .

To solve “Equation 11-4”, 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 9-4.

## 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

$$\rho = \max(1, \|H\|, \|\bar{A}\|, \|A_{eq}\|, \|c\|, \|\bar{b}\|, \|b_{eq}\|)$$

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*  $\varphi$  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 11-4. Use the following definitions:

$$\begin{aligned}\rho &= \max(1, \|H\|, \|\bar{A}\|, \|A_{\text{eq}}\|, \|c\|, \|\bar{b}\|, \|b_{\text{eq}}\|) \\ r_{\text{eq}} &= A_{\text{eq}}x - b_{\text{eq}} \\ r_{\text{ineq}} &= \bar{A}x - \bar{b} - s \\ r_{\text{d}} &= Hx + c + A_{\text{eq}}^T \lambda_{\text{eq}} + \bar{A}^T \bar{\lambda}_{\text{ineq}} \\ g &= x^T Hx + f^T x - \bar{b}^T \bar{\lambda}_{\text{ineq}} - b_{\text{eq}}^T \lambda_{\text{eq}}.\end{aligned}$$

The notation  $\bar{A}$  and  $\bar{b}$  means the linear inequality coefficients, augmented with terms to represent bounds for the sparse algorithm. The notation  $\bar{\lambda}_{\text{ineq}}$  similarly represents Lagrange multipliers for the linear inequality constraints, including bound constraints. This was called  $z$  in “Predictor-Corrector” on page 11-4, and  $\lambda_{\text{eq}}$  was called  $y$ .

The merit function  $\varphi$  is

$$\frac{1}{\rho} (\max(\|r_{\text{eq}}\|_{\infty}, \|r_{\text{ineq}}\|_{\infty}, \|r_{\text{d}}\|_{\infty}) + g).$$

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,

$$\min_s \{q(s), s \in N\}. \quad (11-5)$$

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

$$\min \left\{ \frac{1}{2} s^T H s + s^T g \text{ such that } \|D s\| \leq \Delta \right\}, \quad (11-6)$$

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,  $\Delta$  is a positive scalar, and  $\| \cdot \|$  is the 2-norm. Good algorithms exist for solving “Equation 11-6” (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 11-6”. 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 11-6”, 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 11-6” 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

$$H \cdot s_2 = -g, \quad (11-7)$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \quad (11-8)$$

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 11-6” to determine the trial step  $s$ .
- 3 If  $f(x + s) < f(x)$ , then  $x = x + s$ .
- 4 Adjust  $\Delta$ .

These four steps are repeated until convergence. The trust-region dimension  $\Delta$  is adjusted according to standard rules. In particular, it is decreased if the trial step is not accepted, i.e.,  $f(x + s) \geq f(x)$ . See [46] and [49] for a discussion of this aspect.

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

The subspace trust-region method is used to determine a search direction. However, instead of restricting the step to (possibly) one reflection step, as in the nonlinear minimization case, a piecewise reflective line search is conducted at each iteration. See [45] for details of the line search.

### Preconditioned Conjugate Gradient Method

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

In a minimization context, you can assume that the Hessian matrix  $H$  is symmetric. However,  $H$  is guaranteed to be positive definite only in the neighborhood of a strong minimizer. Algorithm PCG exits when a direction of negative (or zero) curvature is encountered, i.e.,  $d^T H d \leq 0$ . The PCG output direction,  $p$ , is either a direction of negative curvature or an approximate (*tol* controls how approximate) solution to the Newton system  $Hp = -g$ . In either case  $p$  is used to help define the two-dimensional subspace used in the trust-region approach discussed in “Trust-Region Methods for Nonlinear Minimization” on page 6-2.

### Linear Equality Constraints

Linear constraints complicate the situation described for unconstrained minimization. However, the underlying ideas described previously can be carried through in a clean and efficient way. The trust-region methods in Optimization Toolbox solvers generate strictly feasible iterates.

The general linear equality constrained minimization problem can be written

$$\min\{f(x) \text{ such that } Ax = b\}, \quad (11-9)$$

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 11-9” 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

$$\begin{bmatrix} C & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \quad (11-10)$$

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

$$\min\{f(x) \text{ such that } l \leq x \leq u\}, \quad (11-11)$$

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  $\infty$ . 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 11-11",

$$(D(x))^{-2}g = 0, \quad (11-12)$$

where

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

and the vector  $v(x)$  is defined below, for each  $1 \leq i \leq n$ :

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

The nonlinear system "Equation 11-12" 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 11-12" is defined as the solution to the linear system

$$\widehat{M}Ds^N = -\widehat{g} \quad (11-13)$$

at the  $k$ th iteration, where

$$\widehat{g} = D^{-1}g = \text{diag}(|v|^{1/2})g, \quad (11-14)$$

and

$$\widehat{M} = D^{-1}HD^{-1} + \text{diag}(g)J^v. \quad (11-15)$$

Here  $J^v$  plays the role of the Jacobian of  $|v|$ . Each diagonal component of the diagonal matrix  $J^v$  equals 0, -1, or 1. If all the components of  $l$  and  $u$  are finite,  $J^v = \text{diag}(\text{sign}(g))$ . At a point where  $g_i = 0$ ,  $v_i$  might not be differentiable.  $J_{ii}^v = 0$  is defined at such a point. Nondifferentiability of this type is not a cause for concern because, for such a component, it is not significant which value  $v_i$  takes. Further,  $|v_i|$  will still be discontinuous at this point, but the function  $|v_i| \cdot g_i$  is continuous.

Second, reflections are used to increase the step size. A (single) reflection step is defined as follows. Given a step  $p$  that intersects a bound constraint, consider the first bound constraint crossed by  $p$ ; assume it is the  $i$ th bound constraint (either the  $i$ th upper or  $i$ th lower bound). Then the reflection step  $p^R = p$  except in the  $i$ th component, where  $p_i^R = -p_i$ .

## Quadratic Minimization with Bound Constraints

To minimize a large-scale quadratic with upper and lower bounds, you can use the `quadprog` function with the 'trust-region-reflective' algorithm.

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

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

```
load qpbox1 % Get H
lb = zeros(400,1); lb(400) = -inf;
ub = 0.9*ones(400,1); ub(400) = inf;
f = zeros(400,1); f([1 400]) = -2;
```

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

```
xstart = 0.5*ones(400,1);
options = optimoptions('quadprog','Algorithm','trust-region-reflective');
[x,fval,exitflag,output] = ...
    quadprog(H,f,[],[],[],[],lb,ub,xstart,options);
```

Looking at the resulting values of `exitflag`, `output.iterations`, and `output.cgiterations`,

```
exitflag,output.iterations,output.cgiterations
```

```
exitflag =
```

```
3
```

```
ans =
```

```
19
```

```
ans =
```

```
1637
```

You can see that while convergence occurred in 19 iterations, the high number of CG iterations indicates that the cost of solving the linear system is high. In light of this cost, try using a direct solver at each iteration by setting the `SubproblemAlgorithm` option to `'factorization'`:

```
options = optimoptions(options,'SubproblemAlgorithm','factorization');
[x,fval,exitflag,output] = ...
    quadprog(H,f,[],[],[],[],lb,ub,xstart,options);
```

Now the number of iterations has dropped to 10:

```
exitflag,output.iterations,output.cgiterations
```

```
exitflag =
```

```
    3
```

```
ans =
```

```
    10
```

```
ans =
```

```
    0
```

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

You can also use the default `'interior-point-convex'` algorithm to solve this convex problem:

```
options = optimoptions('quadprog','Algorithm','interior-point-convex');
[x,fval,exitflag,output] = ...
    quadprog(H,f,[],[],[],[],lb,ub,[],options);
```

Check the exit flag and iterations (the interior-point algorithm does not use CG iterations):

```
exitflag,output.iterations
```

```
exitflag =
```



1

ans =

8

## Quadratic Minimization with Dense, Structured Hessian

### In this section...

“Take advantage of a structured Hessian” on page 11-18

“Step 1: Decide what part of  $H$  to pass to `quadprog` as the first argument.” on page 11-19

“Step 2: Write a function to compute Hessian-matrix products for  $H$ .” on page 11-19

“Step 3: Call a quadratic minimization routine with a starting point.” on page 11-20

“Preconditioning” on page 11-21

### 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 11-21).

```
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',mtxmpy);

% 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] = runqbox4prec
%RUNQPBOX4PREC demonstrates 'HessianMultiplyFcn' option for QUADPROG with bounds.

problem = load('qbox4'); % 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 = @qbox4mult; % function handle to qbox4mult 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 qbox4mult 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 = qbox4mult(B,Y)
%QPBOX4MULT Hessian matrix product with dense structured Hessian.
% W = qbox4mult(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 runqbox4prec:
%     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] = runqbox4prec;
```

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 12-37

## 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

$$x(1) + x(2) + \dots + x(n) \leq 0,$$

where  $f = [-1; -2; -3; \dots; -n]$ .  $H$  is a sparse symmetric banded matrix.

### Create Sparse Quadratic Matrix

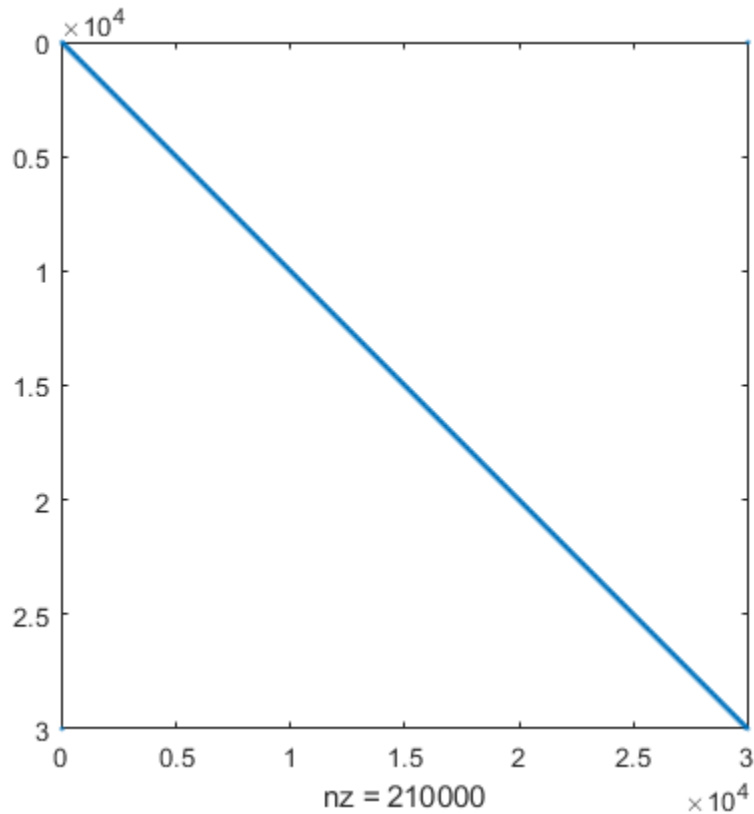
Create a symmetric circulant matrix based on shifts of the vector  $[3, 6, 2, 14, 2, 6, 3]$ , with 14 being on the main diagonal. Have the matrix be  $n$ -by- $n$ , where  $n = 30,000$ .

```
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 Lagr\n    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.

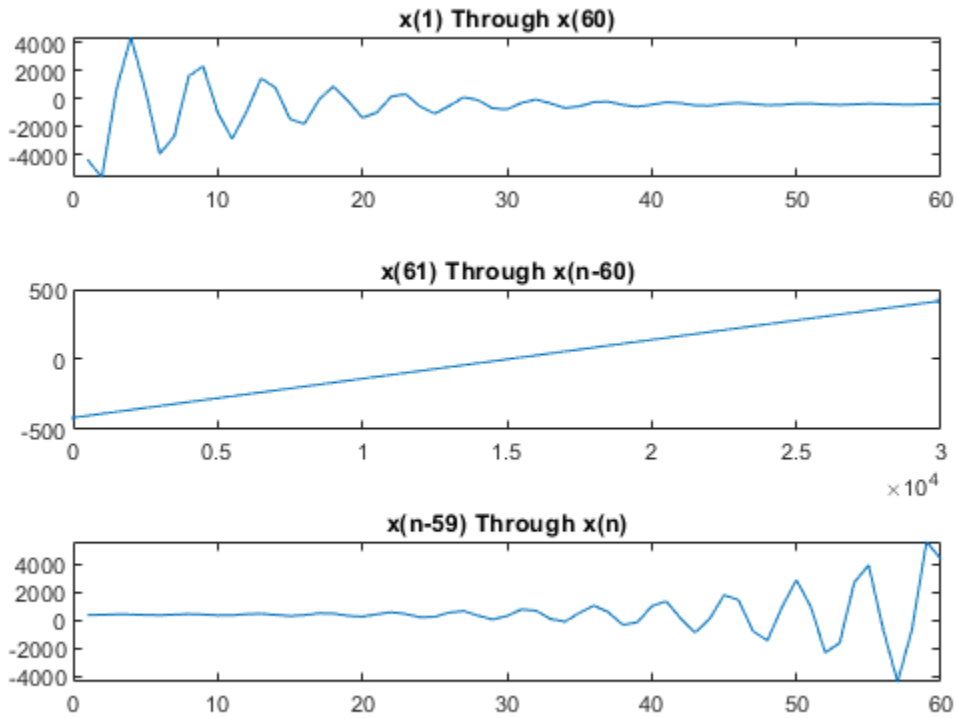
```
fprintf('The linear inequality constraint A*x has value %d.\n', A*x)
```

```
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" (MATLAB)

## 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 11-50.

### 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  $E_{\text{stretch}}$  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  $\Delta x$  represent a shift of 1 in the first coordinate direction, and  $\Delta y$  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  $x$ .

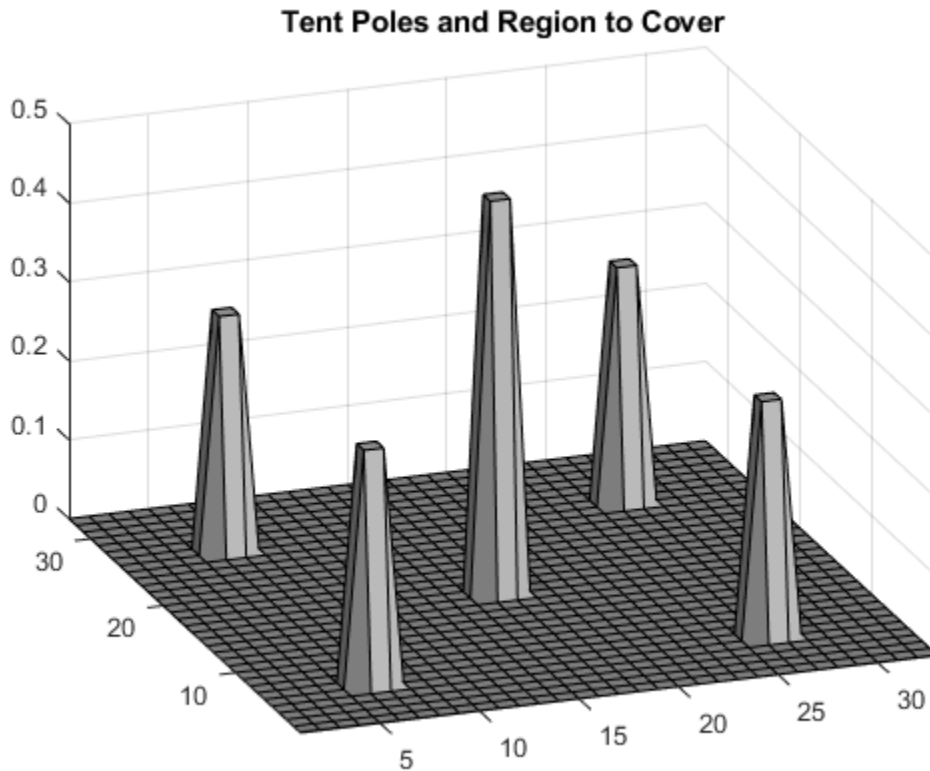
Specify the boundary condition that the height of the tent at the edges is zero. The tent poles have a cross section of 1-by-1 unit, and the tent has a total size of 33-by-33 units. Specify the height and location of each pole. Plot the square lot region and tent poles.

```
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')

```



### 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  $E_{\text{stretch}}$  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  $E_{\text{stretch}}$ .

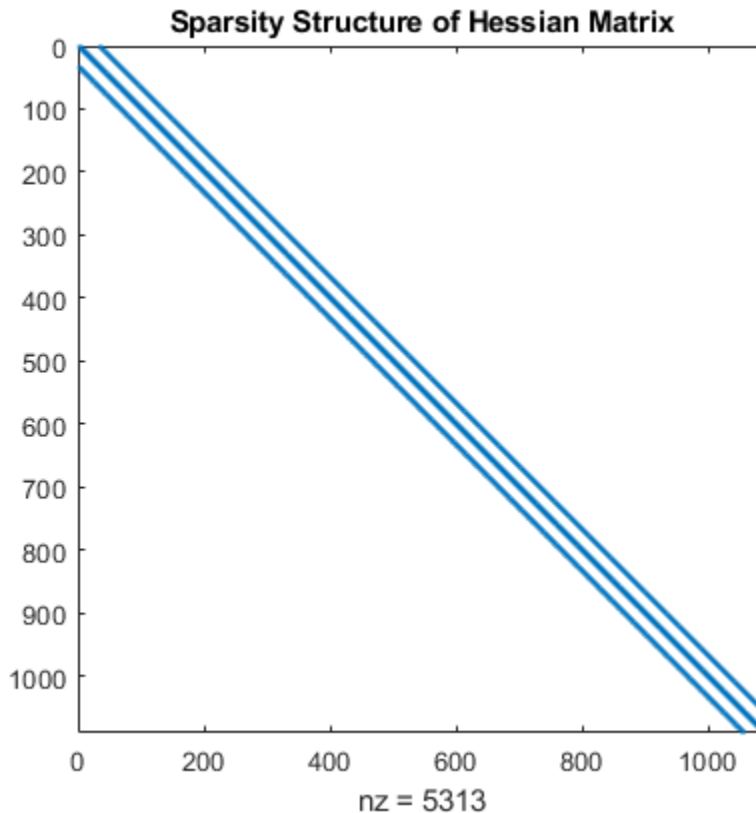
```
H = delsq(numgrid('S',33+2))*2;
```

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.

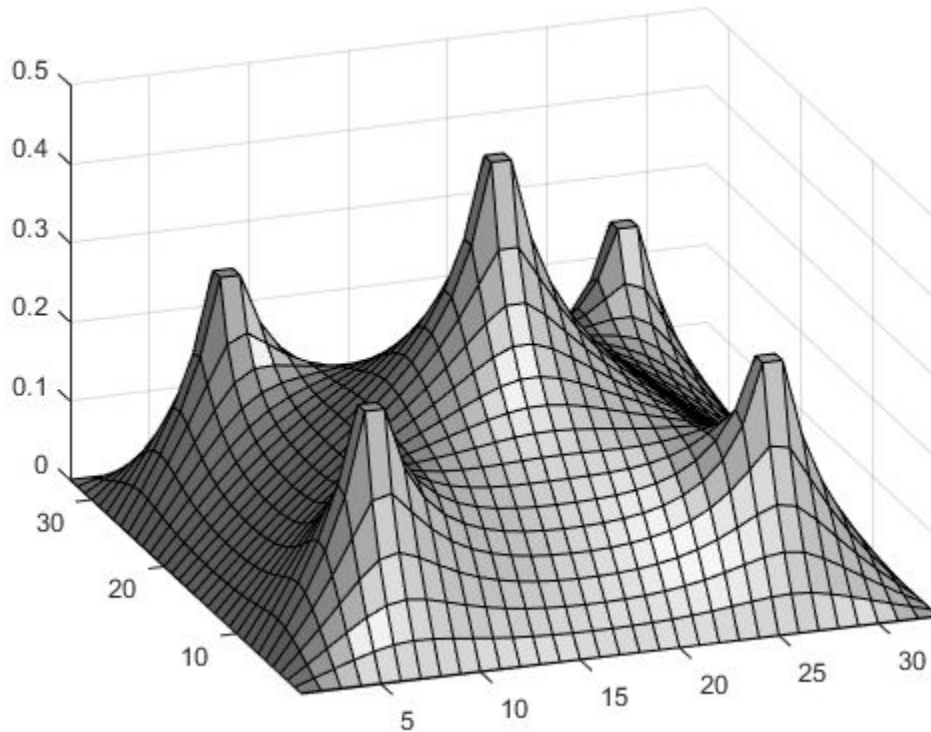
Optimization completed because the objective function is non-decreasing in feasible directions, to within the value of the optimality tolerance, and constraints are satisfied to within the value of the constraint tolerance.

### 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 11-50

## 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 11-24.

### 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,

$$0 \leq x_i \leq 1, \quad i = 1 \dots n.$$

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
```

### 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	8.850332e-10	5.252599e-12	3.858983e-05
10	2.277722e-04	4.422799e-13	2.626104e-15	6.204101e-06
11	1.992243e-04	1.140581e-16	2.161231e-18	4.391483e-07
12	1.950468e-04	0.000000e+00	1.387779e-17	1.429441e-08
13	1.949141e-04	1.114560e-16	1.206517e-18	9.731942e-10
14	1.949121e-04	6.670012e-16	2.483738e-18	2.209702e-12

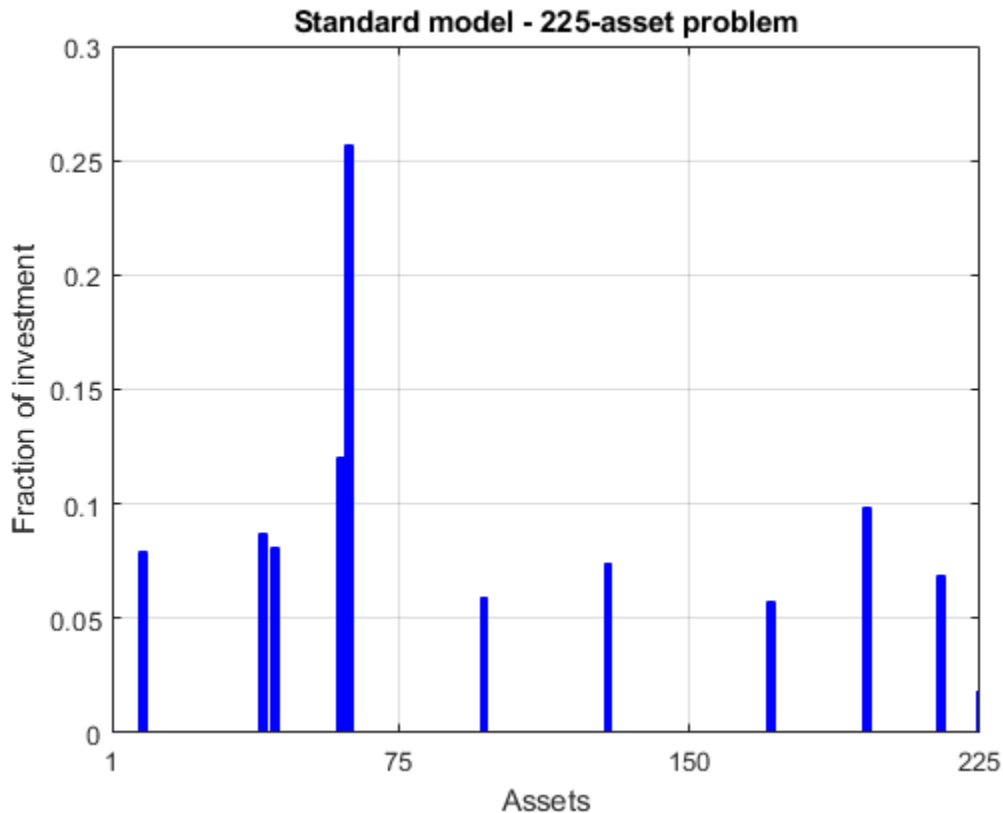
Minimum found that satisfies the constraints.

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 0.329037 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

$$\sum_{i=1}^{75} x_i \geq 0.3, \quad \sum_{i=76}^{150} x_i \geq 0.3, \quad \sum_{i=151}^{225} x_i \geq 0.3.$$

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	6.586818e-08	1.973094e-10	6.509001e-06
9	2.252657e-04	2.225507e-08	6.666550e-11	6.783212e-06
10	2.105838e-04	5.811527e-09	1.740855e-11	1.967570e-06
11	2.024362e-04	4.129608e-12	1.237090e-14	5.924109e-08
12	2.009703e-04	4.289971e-15	1.369512e-17	6.353270e-10
13	2.009650e-04	5.555452e-16	6.938894e-18	1.596041e-13

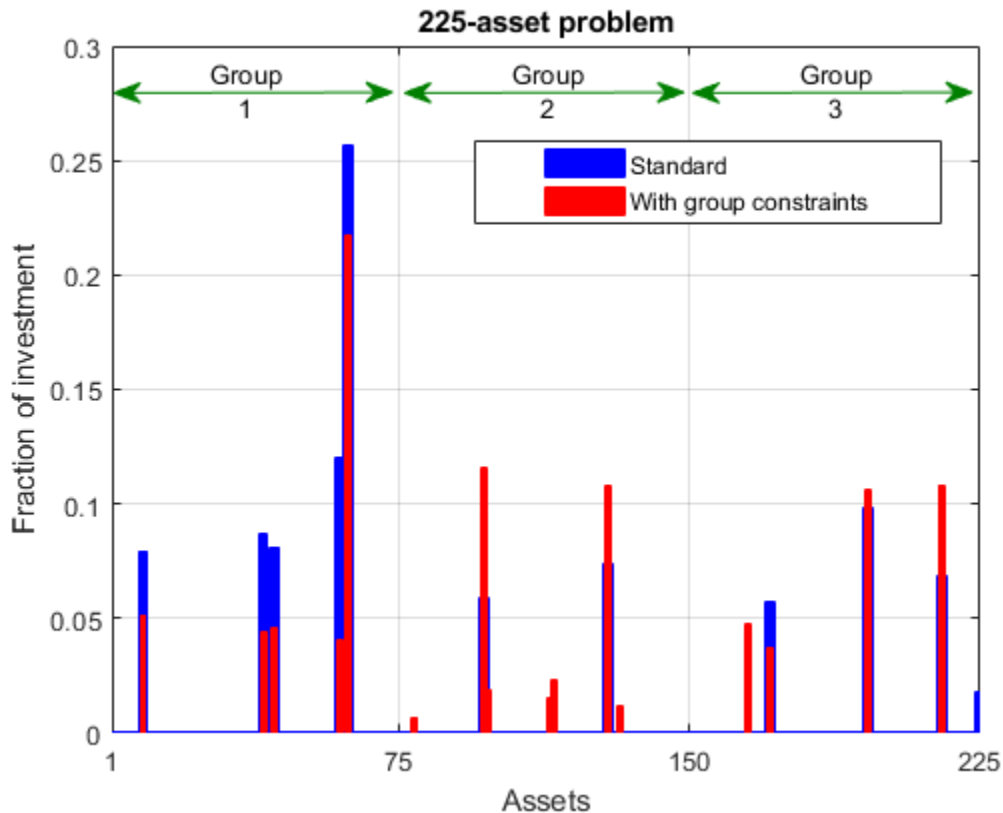
Minimum found that satisfies the constraints.

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 0.078887 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 0
```

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	3.542285e-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
6	2.077811e-05	2.562892e-08	3.612553e-11	5.617206e-07
7	1.611590e-05	4.711755e-10	6.641536e-13	5.652911e-08
8	1.491953e-05	4.926171e-12	6.940605e-15	2.427880e-09
9	1.477930e-05	1.314782e-13	1.850937e-16	2.454705e-10
10	1.476910e-05	1.942890e-16	7.679048e-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 0.578839 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 9-86

## 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 11-15.

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);
```

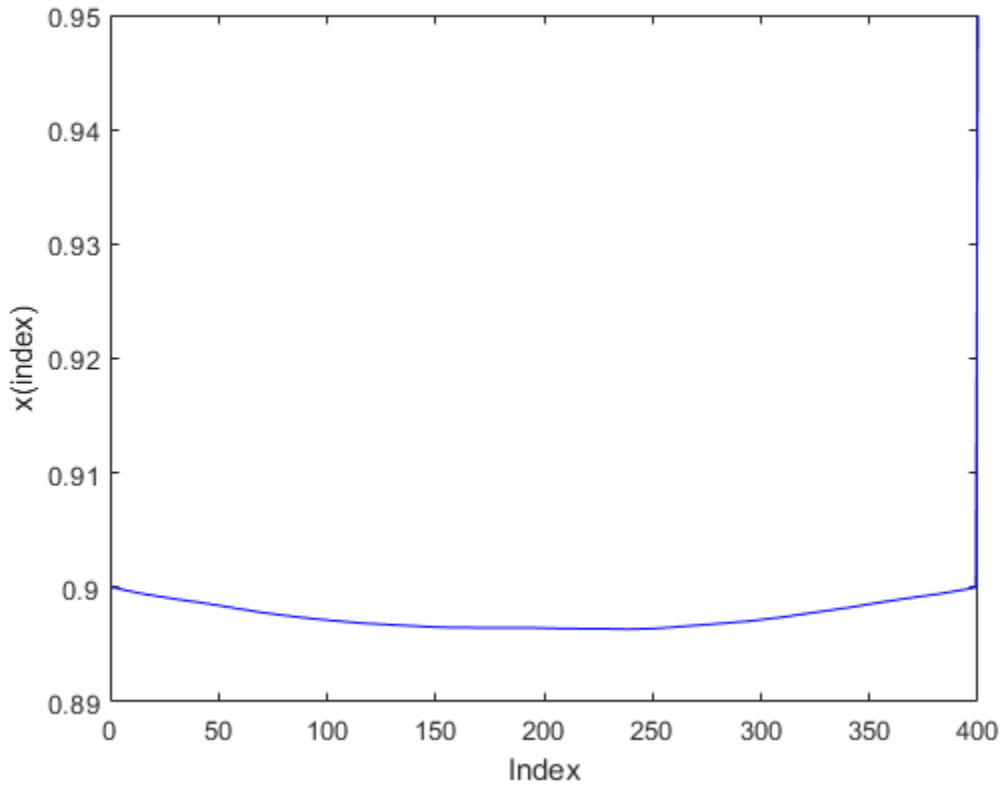
### Solve Problem and Examine Solution

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 = 1668
```

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 0.896446, 0.897823, 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 0.001369.
```

```
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 0.035100.

```
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 11-15

## 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

$$x(1) + x(2) + \dots + x(n) \leq 0,$$

where  $f = [-1; -2; -3; \dots; -n]$ .  $H$  is a sparse symmetric banded matrix.

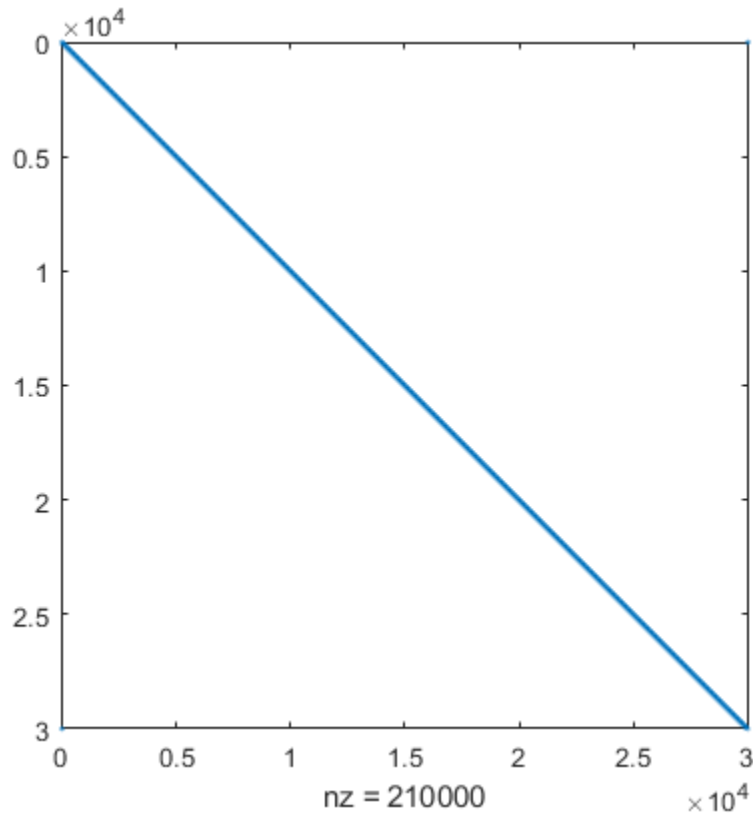
### Create Sparse Quadratic Matrix

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 Lag  
    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.
```

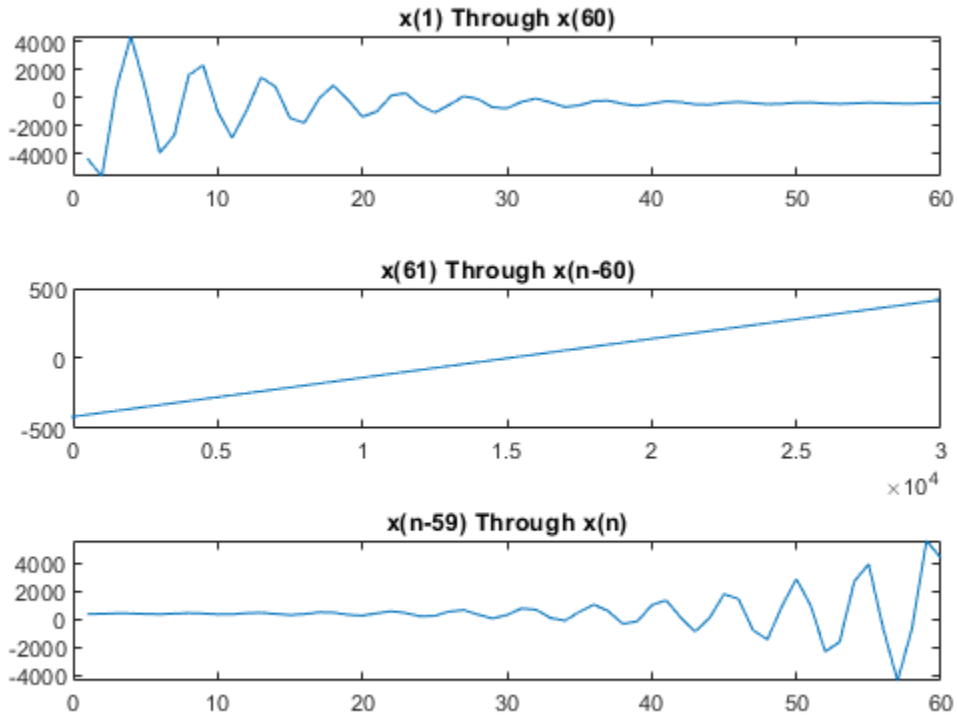
The sum of the solution components is zero to within tolerances.

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 11-24

## 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 11-29.

### 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  $E_{\text{stretch}}$  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  $\Delta x$  represent a shift of 1 in the first coordinate direction, and  $\Delta y$  represent a shift of 1 in the second coordinate direction.

$$E_{\text{stretch}}(p) = \left( -1 \left( x(p + \Delta_x) + x(p - \Delta_x) + x(p + \Delta_y) + x(p - \Delta_y) \right) + 4x(p) \right) x(p).$$

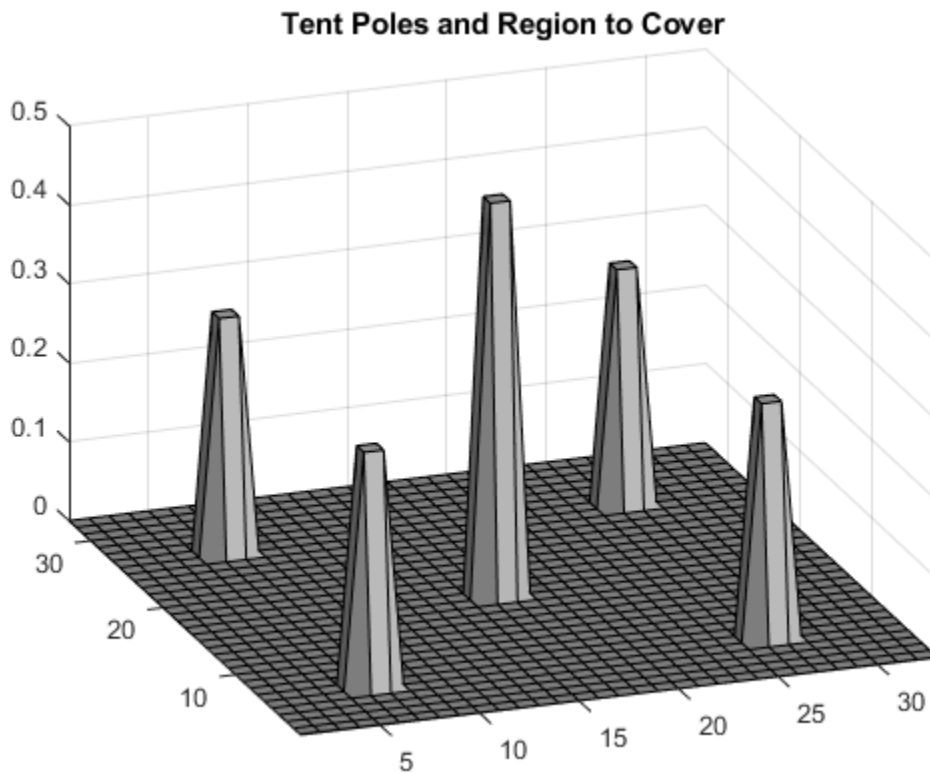
The natural shape of the tent minimizes the total potential energy. By discretizing the problem, you find that the total potential energy to minimize is the sum over all positions  $p$  of  $E_{\text{stretch}}(p) + cx(p)$ .

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);  
surf(height)  
axis tight  
view([-20,30]);  
title('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);
```

```
Your Hessian is not symmetric. Resetting H=(H+H')/2.
```

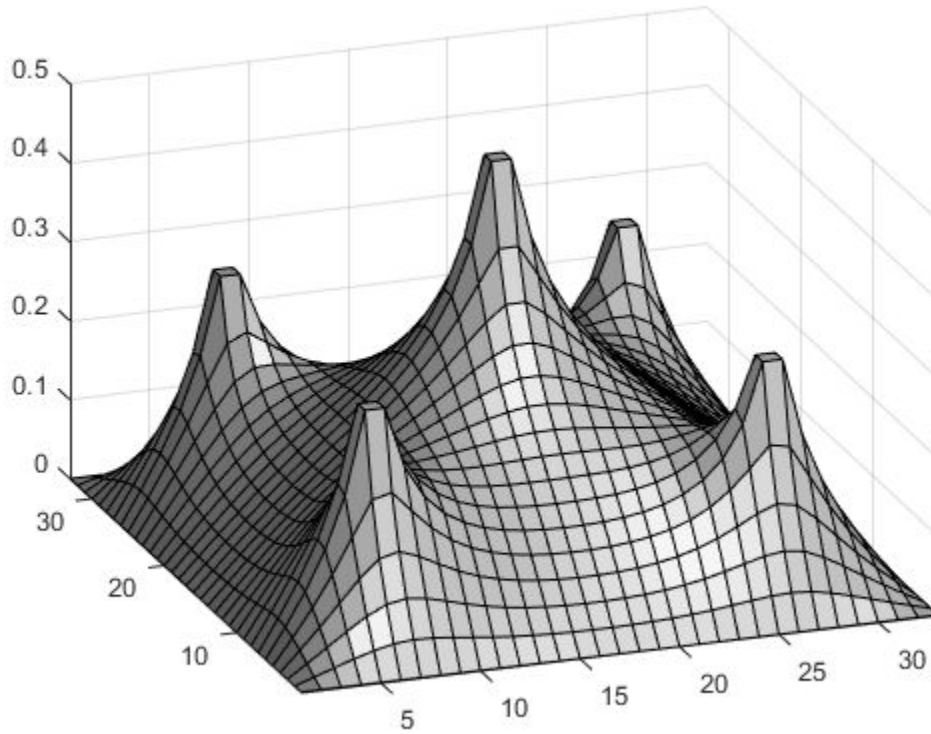
```
Minimum found that satisfies the constraints.
```

```
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.

```
surf(sol.x);  
axis tight;  
view([-20,30]);
```



## See Also

### More About

- “Bound-Constrained Quadratic Programming, Solver-Based” on page 11-29

## 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 11-34.

### 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,

$$0 \leq x_i \leq 1, \quad i = 1 \dots n.$$

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

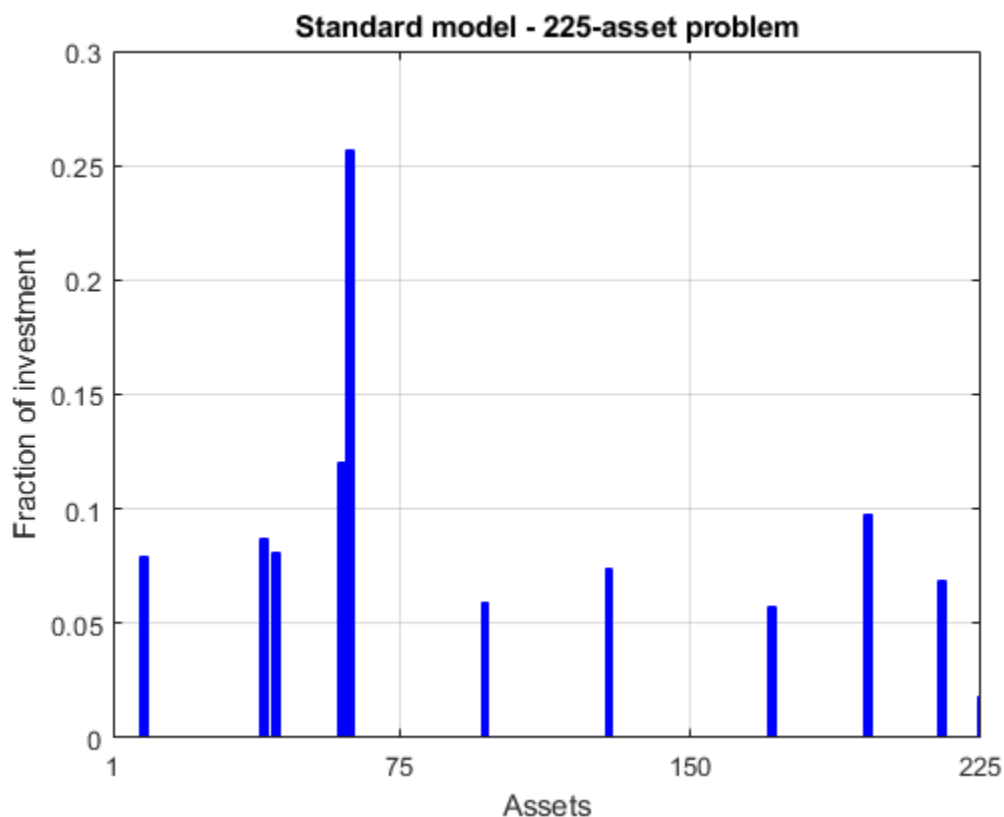
Minimum found that satisfies the constraints.

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

$$\sum_{i=1}^{75} x_i \geq 0.3, \quad \sum_{i=76}^{150} x_i \geq 0.3, \quad \sum_{i=151}^{225} x_i \geq 0.3.$$

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.

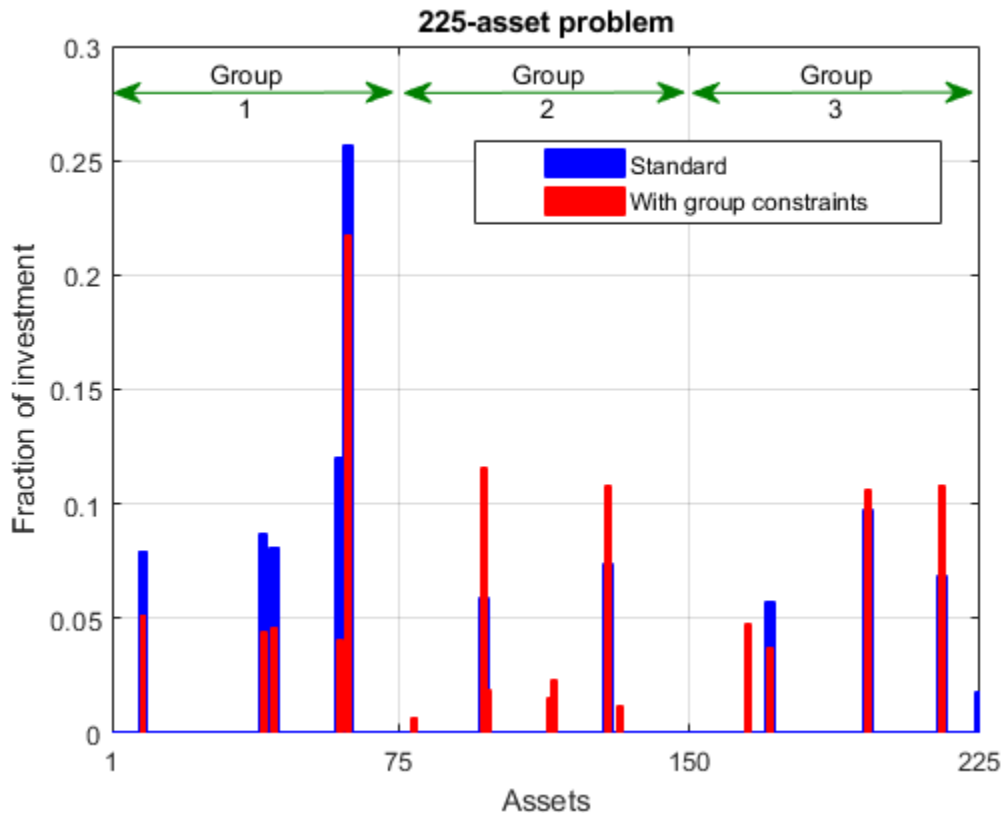
Elapsed time is 0.162406 seconds.

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 11-34





# Least Squares

---

- “Least-Squares (Model Fitting) Algorithms” on page 12-2
- “Nonlinear Data-Fitting” on page 12-11
- “lsqnonlin with a Simulink Model” on page 12-21
- “Nonlinear Least Squares With and Without Jacobian” on page 12-27
- “Linear Least Squares with Bound Constraints” on page 12-31
- “Optimization App with the lsqin Solver” on page 12-33
- “Jacobian Multiply Function with Linear Least Squares” on page 12-37
- “Large-Scale Constrained Linear Least-Squares, Solver-Based” on page 12-42
- “Shortest Distance to a Plane” on page 12-47
- “Nonnegative Least-Squares, Problem-Based” on page 12-50
- “Large-Scale Constrained Linear Least-Squares, Problem-Based” on page 12-54
- “Nonlinear Curve Fitting with lsqcurvefit” on page 12-60
- “Fit a Model to Complex-Valued Data” on page 12-62
- “Fit an Ordinary Differential Equation (ODE)” on page 12-67

## Least-Squares (Model Fitting) Algorithms

### In this section...

“Least Squares Definition” on page 12-2

“Trust-Region-Reflective Least Squares” on page 12-3

“Interior-Point Linear Least Squares” on page 12-6

“Levenberg-Marquardt Method” on page 12-7

### Least Squares Definition

Least squares, in general, is the problem of finding a vector  $x$  that is a local minimizer to a function that is a sum of squares, possibly subject to some constraints:

$$\min_x \|F(x)\|_2^2 = \min_x \sum_i F_i^2(x)$$

such that  $A \cdot x \leq b$ ,  $Aeq \cdot x = beq$ ,  $lb \leq x \leq ub$ .

There are several Optimization Toolbox solvers available for various types of  $F(x)$  and various types of constraints:

Solver	$F(x)$	Constraints
<code>mldivide</code>	$C \cdot x - d$	None
<code>lsqnonneg</code>	$C \cdot x - d$	$x \geq 0$
<code>lsqlin</code>	$C \cdot x - d$	Bound, linear
<code>lsqnonlin</code>	General $F(x)$	Bound
<code>lsqcurvefit</code>	$F(x, xdata) - ydata$	Bound

There are four least-squares algorithms in Optimization Toolbox solvers, in addition to the algorithms used in `mldivide`:

- Trust-region-reflective
- Levenberg-Marquardt
- `lsqlin` interior-point
- The algorithm used by `lsqnonneg`

All the algorithms are large-scale; see “Large-Scale vs. Medium-Scale Algorithms” on page 2-14. For a general survey of nonlinear least-squares methods, see Dennis [8]. Specific details on the Levenberg-Marquardt method can be found in Moré [28].

## Trust-Region-Reflective Least Squares

### Trust-Region-Reflective Least Squares Algorithm

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

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

$$\min_s \{q(s), s \in N\}. \quad (12-1)$$

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

$$\min \left\{ \frac{1}{2} s^T H s + s^T g \text{ such that } \|Ds\| \leq \Delta \right\}, \quad (12-2)$$

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,  $\Delta$  is a positive scalar, and  $\| \cdot \|$ .

$\|$  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

$$H \cdot s_2 = -g, \tag{12-3}$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \tag{12-4}$$

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  $\Delta$ .

These four steps are repeated until convergence. The trust-region dimension  $\Delta$  is adjusted according to standard rules. In particular, it is decreased if the trial step is not accepted, i.e.,  $f(x + s) \geq f(x)$ . See [46] and [49] for a discussion of this aspect.

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

### Large Scale Nonlinear Least Squares

An important special case for  $f(x)$  is the nonlinear least-squares problem

$$\min_x \sum_i f_i^2(x) = \min_x \|F(x)\|_2^2, \quad (12-5)$$

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

$$\min \|Js + F\|_2^2, \quad (12-6)$$

(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 12-37.

## Interior-Point Linear Least Squares

The `lsqlin` 'interior-point' algorithm uses the “interior-point-convex quadprog Algorithm” on page 11-2. The quadprog problem definition is to minimize a quadratic function

$$\min_x \frac{1}{2} x^T H x + c^T x$$

subject to linear constraints and bound constraints. The `lsqlin` function minimizes the squared 2-norm of the vector  $Cx - d$  subject to linear constraints and bound constraints. In other words, `lsqlin` minimizes

$$\begin{aligned}\|Cx - d\|_2^2 &= (Cx - d)^T(Cx - d) \\ &= (x^T C^T - d^T)(Cx - d) \\ &= (x^T C^T Cx) - (x^T C^T d - d^T Cx) + d^T d \\ &= \frac{1}{2}x^T(2C^T C)x + (-2C^T d)^T x + d^T d.\end{aligned}$$

This fits into the `quadprog` framework by setting the  $H$  matrix to  $2C^T C$  and the  $c$  vector to  $(-2C^T d)$ . (The additive term  $d^T d$  has no effect on the location of the minimum.) After this reformulation of the `lsqlin` problem, the `quadprog` 'interior-point-convex' algorithm calculates the solution.

---

**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" (MATLAB). 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`.

---

## Levenberg-Marquardt Method

In the least-squares problem a function  $f(x)$  is minimized that is a sum of squares.

$$\min_x f(x) = \|F(x)\|_2^2 = \sum_i F_i^2(x). \quad (12-7)$$

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

$$\min_{x \in \mathfrak{R}^n_{t_1}} \int_{t_1}^{t_2} (y(x, t) - \varphi(t))^2 dt, \quad (12-8)$$

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

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

$$\min_{x \in \mathfrak{R}^n} f(x) = \sum_{i=1}^m (\bar{y}(x, t_i) - \bar{\varphi}(t_i))^2, \quad (12-9)$$

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

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

In problems of this kind, the residual  $\|F(x)\|$  is likely to be small at the optimum since it is general practice to set realistically achievable target trajectories. Although the function in LS can be minimized using a general unconstrained minimization technique, as described in “Basics of Unconstrained Optimization” on page 6-5, certain characteristics of the problem can often be exploited to improve the iterative efficiency of the solution procedure. The gradient and Hessian matrix of LS have a special structure.

Denoting the  $m$ -by- $n$  Jacobian matrix of  $F(x)$  as  $J(x)$ , the gradient vector of  $f(x)$  as  $G(x)$ , the Hessian matrix of  $f(x)$  as  $H(x)$ , and the Hessian matrix of each  $F_i(x)$  as  $H_i(x)$ , you have

$$G(x) = 2J^T F(x) \quad (12-10)$$

$$H(x) = 2J^T J(x) + 2Q(x),$$

where

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

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



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

$$\min_{x \in \mathfrak{R}^n} \|J(x_k) - F(x_k)\|_2^2. \quad (12-11)$$

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

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

The Levenberg-Marquardt [25], and [27] method uses a search direction that is a solution of the linear set of equations

$$\left(J(x_k)^T J(x_k) + \lambda_k I\right) d_k = -J(x_k)^T F(x_k), \quad (12-12)$$

or, optionally, of the equations

$$\left(J(x_k)^T J(x_k) + \lambda_k \text{diag}\left(J(x_k)^T J(x_k)\right)\right) d_k = -J(x_k)^T F(x_k), \quad (12-13)$$

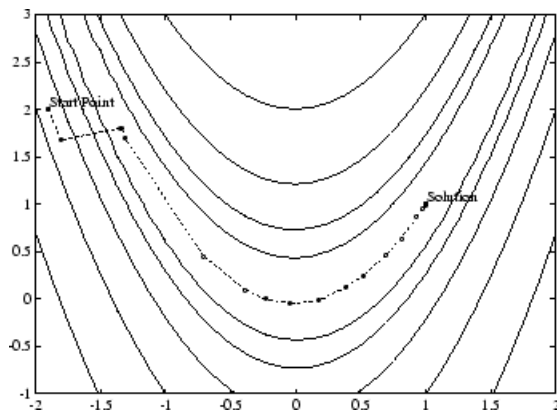
where the scalar  $\lambda_k$  controls both the magnitude and direction of  $d_k$ . Set option `ScaleProblem` to 'none' to choose "Equation 12-12", and set `ScaleProblem` to 'Jacobian' to choose "Equation 12-13".

You set the initial value of the parameter  $\lambda_0$  using the `InitDamping` option. Occasionally, the 0.01 default value of this option can be unsuitable. If you find that the Levenberg-Marquardt algorithm makes little initial progress, try setting `InitDamping` to a different value than the default, perhaps `1e2`.

When  $\lambda_k$  is zero, the direction  $d_k$  is identical to that of the Gauss-Newton method. As  $\lambda_k$  tends to infinity,  $d_k$  tends towards the steepest descent direction, with magnitude tending to zero. This implies that for some sufficiently large  $\lambda_k$ , the term  $F(x_k + d_k) < F(x_k)$  holds true. The term  $\lambda_k$  can therefore be controlled to ensure descent even when second-order terms, which restrict the efficiency of the Gauss-Newton method, are encountered. When the step is successful (gives a lower function value), the algorithm sets  $\lambda_{k+1} = \lambda_k/10$ . When the step is unsuccessful, the algorithm sets  $\lambda_{k+1} = \lambda_k * 10$ .

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. This is illustrated in “Figure 12-1, Levenberg-Marquardt Method on Rosenbrock's Function” on page 12-10. The solution for Rosenbrock's function converges after 90 function evaluations compared to 48 for the Gauss-Newton method. The poorer efficiency is partly because the Gauss-Newton method is generally more effective when the residual is zero at the solution. However, such information is not always available beforehand, and the increased robustness of the Levenberg-Marquardt method compensates for its occasional poorer efficiency.



**Figure 12-1, 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”.

## Nonlinear Data-Fitting

This example shows how to fit a nonlinear function to data using several Optimization Toolbox™ algorithms.

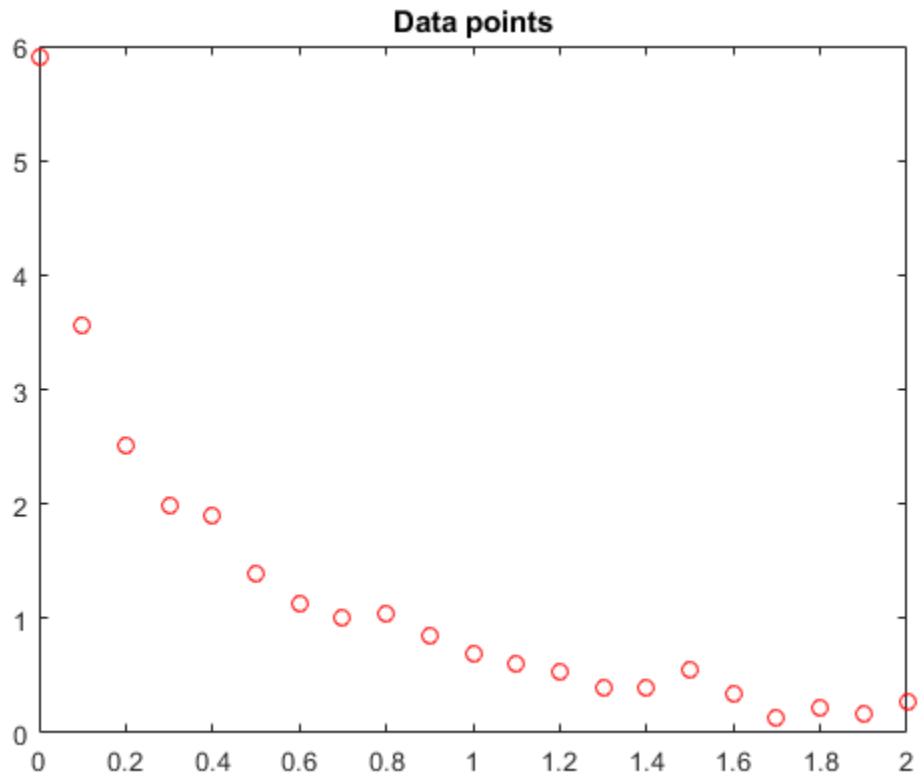
### Problem Setup

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')
```



```
% hold off
```

We would like to fit the function

$$y = c(1)*\exp(-\text{lam}(1)*t) + c(2)*\exp(-\text{lam}(2)*t)$$

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  $x_0$  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: 7.8829e-06
```

```
    iterations: 6
```

```
    funcCount: 35
```

```
    cgiterations: 0
```

```
    algorithm: 'trust-region-reflective'
```

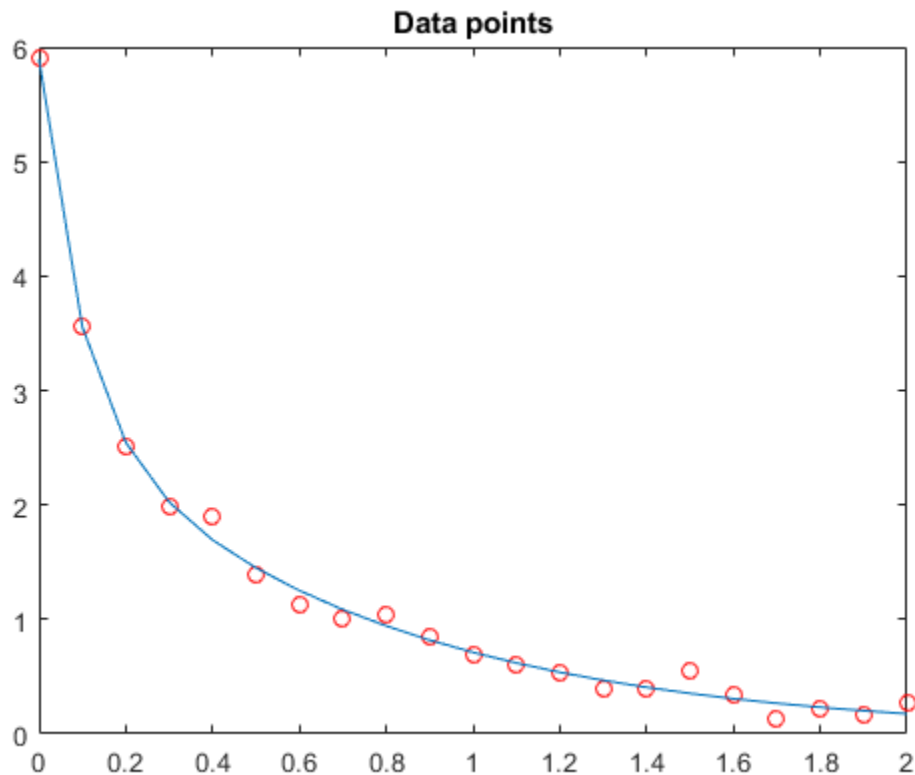
```
    stepsize: 0.0096
```

```
    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: 2.9515e-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: 4.4071e-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 form

### 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  $\text{lam}(1)$  and  $\text{lam}(2)$  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
    -20.2519    2.4796    25.3757    2.4795

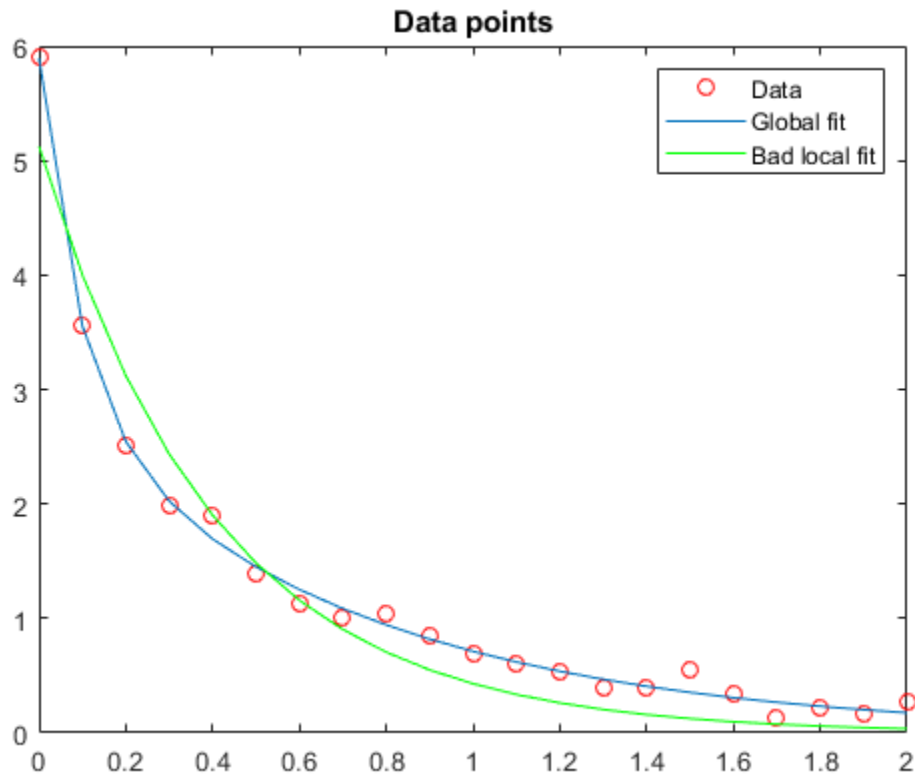
```

```
resnormbad = 2.2173
```

```
exitflagbad = 3
```

```
outputbad = struct with fields:
  firstorderopt: 0.0036
  iterations: 31
  funcCount: 160
  cgiterations: 0
  algorithm: 'trust-region-reflective'
  stepsize: 0.0012
  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 point is 0.147723.

## See Also

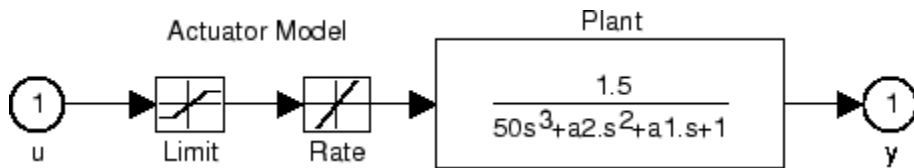
### More About

- “Nonlinear Least Squares With and Without Jacobian” on page 12-27

- “Nonlinear Curve Fitting with lsqcurvefit” on page 12-60

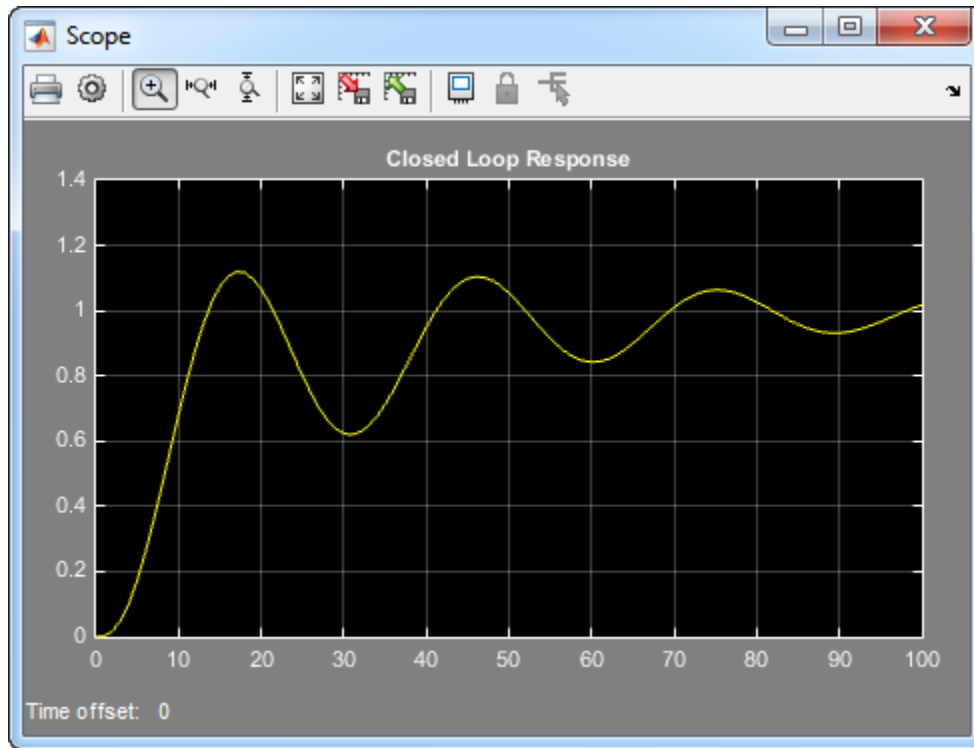
## Isqnonlin with a Simulink Model

Suppose that you want to optimize the control parameters in the Simulink model `optsim.mdl`. (This model can be found in the `optim/optindemos` folder. Note that Simulink must be installed on your system to load this model.) The model includes a nonlinear process plant modeled as a Simulink block diagram.



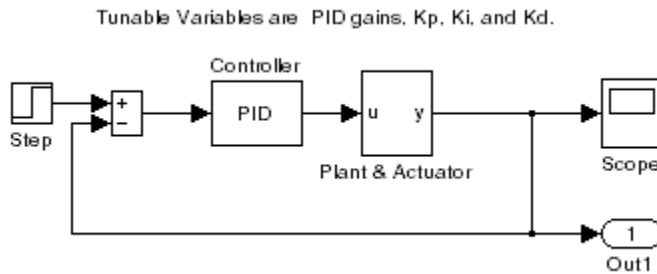
### Plant with Actuator Saturation

The plant is an under-damped third-order model with actuator limits. The actuator limits are a saturation limit and a slew rate limit. The actuator saturation limit cuts off input values greater than 2 units or less than -2 units. The slew rate limit of the actuator is 0.8 units/sec. The closed-loop response of the system to a step input is shown in "Closed-Loop Response" on page 12-22. 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.



### Closed-Loop Model

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

The routine `lsqnonlin` is used to perform a least-squares fit on the tracking of the output. The tracking is performed via the function `tracklsq`, which returns the error signal `yout`, the output computed by calling `sim`, minus the input signal `1`. The code for `tracklsq` is contained in the file `runtracklsq.m`, shown below.

The function `runtracklsq` sets up all the needed values and then calls `lsqnonlin` with the objective function `tracklsq`, which is nested inside `runtracklsq`. The variable `options` passed to `lsqnonlin` defines the criteria and display characteristics. In this case you ask for output, use the 'levenberg-marquardt' algorithm, and give termination tolerances for the step and objective function on the order of `0.001`.

To run the simulation in the model `optsim`, the variables `Kp`, `Ki`, `Kd`, `a1`, and `a2` (`a1` and `a2` are variables in the Plant block) must all be defined. `Kp`, `Ki`, and `Kd` are the variables to be optimized. The function `tracklsq` is nested inside `runtracklsq` so that the variables `a1` and `a2` are shared between the two functions. The variables `a1` and `a2` are initialized in `runtracklsq`.

The objective function `tracklsq` runs the simulation. The simulation can be run either in the base workspace or the current workspace, that is, the workspace of the function calling `sim`, which in this case is the workspace of `tracklsq`. In this example, the `SrcWorkspace` option is set to 'Current' to tell `sim` to run the simulation in the current workspace. The simulation is performed to 100 seconds.

When the simulation is completed, the `myobj` object is created in the current workspace (that is, the workspace of `tracklsq`). The Output block in the block diagram model puts the `yout` field of the object into the current workspace at the end of the simulation.

The following is the code for `runtracklsq`:

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

optsim                % Load the model
pid0 = [0.63 0.0504 1.9688]; % Set initial values
a1 = 3; a2 = 43;        % Initialize model plant variables
options = optimoptions(@lsqnonlin,'Algorithm','levenberg-marquardt',...
    'Display','off','StepTolerance',0.001,'OptimalityTolerance',0.001);
pid = lsqnonlin(@tracklsq, pid0, [], [], options);
Kp = pid(1); Ki = pid(2); Kd = pid(3);

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

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

    % Set sim options and compute function value
    myobj = sim('optsim','SrcWorkspace','Current', ...
        'StopTime','100');
    F = myobj.get('yout') - 1;
end
end
```

Copy the code for `runtracklsq` to a file named `runtracklsq.m`, placed in a folder on your MATLAB path.

When you run `runtracklsq`, the optimization gives the solution for the proportional, integral, and derivative (Kp, Ki, Kd) gains of the controller:

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

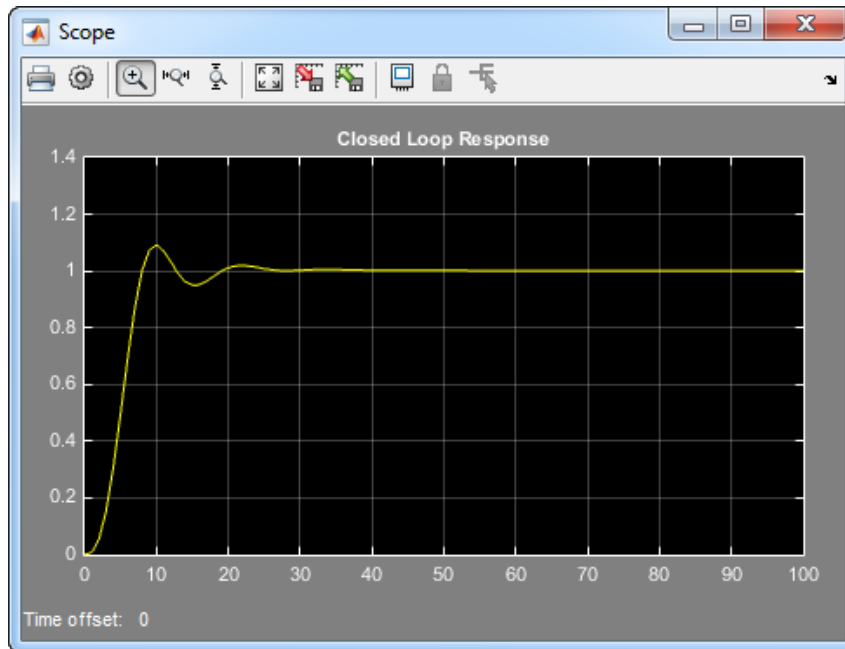
Kp =
    3.1330
```



$K_i =$   
0.1465

$K_d =$   
14.3918

Here is the resulting closed-loop step response.



### Closed-Loop Response Using Isqnonlin

**Note** The call to `sim` results in a call to one of the Simulink ordinary differential equation (ODE) solvers. A choice must be made about the type of solver to use. From the optimization point of view, a fixed-step solver is the best choice if that is sufficient to solve the ODE. However, in the case of a stiff system, a variable-step method might be required to solve the ODE.

The numerical solution produced by a variable-step solver, however, is not a smooth function of parameters, because of step-size control mechanisms. This lack of smoothness

can prevent the optimization routine from converging. The lack of smoothness is not introduced when a fixed-step solver is used. (For a further explanation, see [53].)

Simulink Design Optimization software is recommended for solving multiobjective optimization problems in conjunction with Simulink variable-step solvers. It provides a special numeric gradient computation that works with Simulink and avoids introducing a problem of lack of smoothness.

---

## Nonlinear Least Squares With and Without Jacobian

### In this section...

“Problem definition and solution technique” on page 12-27

“Step 1: Write a file myfun.m that computes the objective function values.” on page 12-27

“Step 2: Call the nonlinear least-squares routine.” on page 12-28

“Step 3: Include a Jacobian.” on page 12-28

### Problem definition and solution technique

This example shows how to solve a nonlinear least squares problem in two ways. It first shows the solution without using a Jacobian function. Then it shows how to include a Jacobian, and it shows the efficiency improvement that the Jacobian gives.

The problem has 10 terms with 2 unknowns: find  $x$ , a two-dimensional vector, that minimizes

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

starting at the point  $x_0 = [0.3, 0.4]$ .

Because `lsqnonlin` assumes that the sum of squares is not explicitly formed in the user function, the function passed to `lsqnonlin` should compute the vector valued function

$$F_k(x) = 2 + 2k - e^{kx_1} - e^{kx_2},$$

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

### Step 1: Write a file myfun.m that computes the objective function values.

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

## Step 2: Call the nonlinear least-squares routine.

```
x0 = [0.3,0.4]; % Starting guess  
[x,resnorm,res,eflag,output1] = lsqnonlin(@myfun,x0); % Invoke optimizer
```

Because the Jacobian is not computed in `myfun.m`, `lsqnonlin` calls the trust-region reflective algorithm with full finite differencing. Note that the `SpecifyObjectiveGradient` option in `options` is set to `false` by default.

After 72 function evaluations, this example gives the solution

```
x,resnorm  
  
x =  
    0.2578    0.2578  
  
resnorm =  
    124.3622
```

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

## Step 3: Include a Jacobian.

The objective function is simple enough to calculate its Jacobian. Following the definition in “Jacobians of Vector Functions” on page 2-34, a Jacobian function represents the matrix

$$J_{kj}(x) = \frac{\partial F_k(x)}{\partial x_j}.$$

Here,  $F_k(x)$  is the  $k$ th component of the objective function. This example has

$$F_k(x) = 2 + 2k - e^{kx_1} - e^{kx_2},$$

so

$$J_{k1}(x) = -ke^{kx_1}$$
$$J_{k2}(x) = -ke^{kx_2}.$$

Modify the objective function file.

```
function [F,J] = myfun(x)
k = 1:10;
F = 2 + 2*k-exp(k*x(1))-exp(k*x(2));
if nargin > 1
    J = zeros(10,2);
    J(k,1) = -k.*exp(k*x(1));
    J(k,2) = -k.*exp(k*x(2));
end
```

Set options so the solver uses the Jacobian.

```
opts = optimoptions(@lsqnonlin,'SpecifyObjectiveGradient',true);
```

Run the solver.

```
x0 = [0.3 0.4]; % Starting guess
[x,resnorm,res,eflag,output2] = lsqnonlin(@myfun,x0,[],[],opts);
```

The solution is the same as before.

```
x, resnorm
```

```
x =
    0.2578    0.2578
```

```
resnorm =
    124.3622
```

The advantage to using a Jacobian is that the solver takes fewer function evaluations, 24 instead of 72.

```
[output1.funcCount,output2.funcCount]
```

```
ans =
    72    24
```

### See Also

#### More About

- “Nonlinear Data-Fitting” on page 12-11
- “Nonlinear Curve Fitting with lsqcurvefit” on page 12-60

## Linear Least Squares with Bound Constraints

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

```
load particle % Get C, d
lb = zeros(400,1);
options = optimoptions('lsqlin','Algorithm','trust-region-reflective');
[x,resnorm,residual,exitflag,output] = ...
    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 machine epsilon.

The default diagonal preconditioning works fairly well:

```
exitflag,resnorm,output
```

```
exitflag =
    3
```

```
resnorm =
    22.5794
```

```
output =
    struct with fields:
```

```
        iterations: 10
        algorithm: 'trust-region-reflective'
    firstorderopt: 2.7870e-05
        cgiterations: 42
    constrviolation: []
        linearsolver: []
        message: 'Local minimum possible. lsqlin stopped because the relative change in function value is less than the square root of the machine epsilon.'
```

For bound constrained problems, the first-order optimality is the infinity norm of  $v \cdot g$ , where  $v$  is defined as in “Box Constraints” on page 6-25, and  $g$  is the gradient.

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

```
options = optimoptions(options,'SubproblemAlgorithm','factorization');  
[x,resnorm,residual,exitflag,output] = ...  
    lsqlin(C,d,[],[],[],[],lb,[],[],options);
```

Optimal solution found.

The first-order optimality measure decreases:

```
exitflag,resnorm,output
```

```
exitflag =  
    1
```

```
resnorm =  
    22.5794
```

```
output =  
    struct with fields:  
        iterations: 12  
        algorithm: 'trust-region-reflective'  
        firstorderopt: 5.5907e-15  
        cgiterations: 0  
        constrviolation: []  
        linearsolver: []  
        message: 'Optimal solution found.'
```

## See Also

### More About

- “Nonnegative Least-Squares, Problem-Based” on page 12-50



## Optimization App with the lsqlin Solver

### In this section...

“The Problem” on page 12-33

“Setting Up the Problem” on page 12-33

### The Problem

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

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

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

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

subject to the constraint

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

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

### Setting Up the Problem

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

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

- 2 Select `lsqlin` from the selection of solvers. Use the Interior point algorithm.

Solver: `lsqlin - Constrained linear least squares`  
 Algorithm: `Interior point`

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

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

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

Problem  
 C: `eye(3)`      d: `zeros(3,1)`

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

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

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

Constraints:  
 Linear inequalities:    A:       b:   
 Linear equalities:    Aeq: `[1 2 4]`      beq: `7`  
 Bounds:              Lower:       Upper:

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

Run solver and view results  
 Start    Pause    Stop  
 Current iteration:     Clear Results

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

Run solver and view results

Start Pause Stop

Current iteration:  Clear Results

Optimization running.  
 Objective function value: 2.333333333333333  
 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

- The **Current iteration** value when the algorithm terminated, which for this example is 1.
- The final value of the objective function when the algorithm terminated:  
Objective function value: 2.333333333333334
- 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.
- The final point, which for this example is
  - 0.333
  - 0.667
  - 1.333

## **See Also**

### **More About**

- “Shortest Distance to a Plane” on page 12-47

## Jacobian Multiply Function with Linear Least Squares

You can solve a least-squares problem of the form

$$\min_x \frac{1}{2} \|C \cdot x - d\|_2^2$$

such that  $lb \leq x \leq ub$ , for problems where  $C$  is very large, perhaps too large to be stored, by using a Jacobian multiply function. For this technique, use the 'trust-region-reflective' algorithm.

For example, consider the case where  $C$  is a  $2n$ -by- $n$  matrix based on a circulant matrix. This means the rows of  $C$  are shifts of a row vector  $v$ . This example has the row vector  $v$  with elements of the form  $(-1)^{k+1}/k$ :

$$v = [1, \quad -1/2, \quad 1/3, \quad -1/4, \quad \dots, \quad -1/n],$$

cyclically shifted:

$$C = \begin{bmatrix} 1 & -1/2 & 1/3 & \dots & -1/n \\ -1/n & 1 & -1/2 & \dots & 1/(n-1) \\ 1/(n-1) & -1/n & 1 & \dots & -1/(n-2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1/2 & 1/3 & -1/4 & \dots & 1 \\ 1 & -1/2 & 1/3 & \dots & -1/n \\ -1/n & 1 & -1/2 & \dots & 1/(n-1) \\ 1/(n-1) & -1/n & 1 & \dots & -1/(n-2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1/2 & 1/3 & -1/4 & \dots & 1 \end{bmatrix}.$$

This least-squares example considers the problem where

$$d = [n \quad - \quad 1; \quad n \quad - \quad 2; \quad \dots; \quad -n],$$

and the constraints are  $-5 \leq x(i) \leq 5$  for  $i = 1, \dots, n$ .

For large enough  $n$ , the dense matrix  $C$  does not fit into computer memory. ( $n = 10,000$  is too large on one tested system.)

A Jacobian multiply function has the following syntax:

```
w = jmfcn(Jinfo,Y,flag)
```

`Jinfo` is a matrix the same size as `C`, used as a preconditioner. If `C` is too large to fit into memory, `Jinfo` should be sparse. `Y` is a vector or matrix sized so that `C*Y` or `C'*Y` makes sense. `flag` tells `jmfcn` which product to form:

- `flag > 0` = `w = C*Y`
- `flag < 0` = `w = C'*Y`
- `flag = 0` = `w = C'*C*Y`

Since `C` is such a simply structured matrix, it is easy to write a Jacobian multiply function in terms of the vector `v`; i.e., without forming `C`. Each row of `C*Y` is the product of a 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 `dolsqJac3` function in the following code sets up the vector `v` and calls the solver `lsqlin`:

```
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 that might be too large to fit into memory.  
  
r = 1:2*n;  
d(r) = n-r;  
  
Jinfo = [speye(n);speye(n)]; % sparse matrix for preconditioning
```

```

% This matrix is a required input for the solver;
% preconditioning is not really being used in this example

% Pass the 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);

```

The Jacobian multiply function `lsqcirculant3` is as follows:

```

function w = lsqcirculant3(Jinfo,Y,flag,v)
% This function computes the Jacobian multiply functions
% for a 2n-by-n circulant matrix example

if flag > 0
    w = Jpositive(Y);
elseif flag < 0
    w = Jnegative(Y);
else
    w = Jnegative(Jpositive(Y));
end

function a = Jpositive(q)
    % Calculate C*q
    temp = v;

    a = zeros(size(q)); % allocating the matrix a
    a = [a;a]; % the result is twice as tall as the input

    for r = 1:size(a,1)
        a(r,:) = temp*q; % compute the rth row
        temp = 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% 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
```

When  $n = 3000$ ,  $C$  is an 18,000,000-element dense matrix. Here are the results of the `dolsqJac` function for  $n = 3000$  at selected values of  $x$ , and the output structure:

```
[x,resnorm,residual,exitflag,output] = dolsqJac3(3000);
```

```
Local minimum possible.
```

```
lsqlin stopped because the relative change in function value is less than the function
```

```
x(1)
```

```
ans =
    5.0000
```

```
x(1500)
```

```
ans =
   -0.5201
```

```
x(3000)
```

```
ans =
   -5.0000
```

```
output
```

```
output =
```

```
struct with fields:
```

```
iterations: 16
algorithm: 'trust-region-reflective'
firstorderopt: 5.9351e-05
```



```
cgiterations: 36
constrviolation: []
linearsolver: []
message: 'Local minimum possible. fminsqm stopped because the relative change
```

## See Also

`circshift` | `fliplr`

## More About

- “Quadratic Minimization with Dense, Structured Hessian” on page 11-18

## 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 12-54.

### 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))
```



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  $m \times n$  matrix  $P$ .

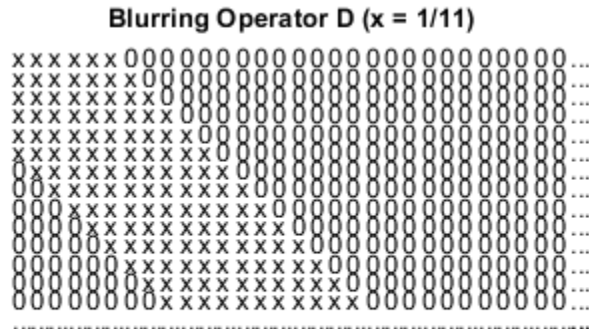
**Add Motion**

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)')
```



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')
```



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  $x$ :

$$\min(\|Dx - G\|^2) \text{ subject to } 0 \leq x \leq 1.$$

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.

```
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')
```



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:

$$\min(\|(D + \varepsilon I)x - G\|^2) \text{ subject to } 0 \leq x \leq 1.$$

The term  $\varepsilon I$  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')
```



Apparently, this simple regularization does not remove the artifacts.

## See Also

### More About

- “Large-Scale Constrained Linear Least-Squares, Problem-Based” on page 12-54

## 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 `pointtplane`.

```
pointtplane = 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 `pointtplane`.

```
obj = sum(x.^2);
pointtplane.Objective = obj;
```

Create the linear constraint and put it in the problem.

```
v = [1,2,4];
pointtplane.Constraints = dot(x,v) == 7;
```

The problem formulation is complete. To check for errors, review the problem.

```
showproblem(pointtplane)
```

```
OptimizationProblem :
    minimize :
        x(1)^2 + x(2)^2 + x(3)^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);
```

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  $x_{opt}$  is  $t*v$  for the value of  $t$  that satisfies the equation  $\text{dot}(t*v, v) = 7$ .

```
t = 7/dot(v,v)
```

```
t = 0.3333
```

```
xopt = t*v
```

```
xopt = 1x3
```

```
0.3333    0.6667    1.3333
```



Indeed, the vector `xopt` is equivalent to the point `sol.x` that `solve` finds.

## See Also

### More About

- “Optimization App with the `lsqlin` Solver” on page 12-33

## Nonnegative Least-Squares, Problem-Based

This example shows problem-based approaches for solving the problem

$$C*x = d$$

in the least-squares sense. Here  $C$  and  $d$  are given arrays and  $x$  is the unknown array, and  $x$  is restricted to be nonnegative. In other words,

$$\text{minimize } \text{sum}((C*x - d).^2)$$

subject to  $x \geq 0$ .

To begin, load arrays  $C$  and  $d$  into your workspace.

```
load particle
```

View the sizes of the arrays.

```
sizec = size(C)
```

```
sizec = 1x2
```

```
    2000    400
```

```
sized = size(d)
```

```
sized = 1x2
```

```
    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: '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] = solve(nonneglsq)
```

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
```

The default solver, `lsqlin`, worked quickly. But there are other solvers that handle problems of this nature. `lsqnonneg` solves problems of this type, as does `quadprog`. Try each.

```
[sol2,fval2,exitflag2] = solve(nonneglsq,'Solver','lsqnonneg');  
[sol3,fval3,exitflag3] = solve(nonneglsq,'Solver','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.
```

Examine the first ten entries of each solution.

```
solutions = table(sol.x(1:10),sol2.x(1:10),sol3.x(1:10),'VariableNames',{'lsqlin','lsqnonneg','quadprog'})
```

```
solutions=10x3 table  
      lsqlin      lsqnonneg      quadprog  
-----  
      0.049416      0.049416      0.049416  
      0.082985      0.082985      0.082985  
      0.078469      0.078469      0.078469  
      0.11682       0.11682       0.11682  
      0.07165       0.07165       0.07165  
      0.060525      0.060525      0.060525  
      1.2196e-09      0             1.2196e-09  
      2.703e-10      0             2.703e-10  
      1.5017e-10      0             1.5017e-10  
      1.2331e-10      0             1.2331e-10
```

Several `lsqnonneg` entries appear to be exactly zero, while the other solutions are not exactly zero. For example, the eighth entry in each solution is:

```
fprintf('The entries are %d, %d, and %d.\n',sol.x(8),sol2.x(8),sol3.x(8))
```

```
The entries are 2.702960e-10, 0, and 2.702960e-10.
```

Perhaps the other `quadprog` and `lsqlin` algorithm would be a bit more accurate.

```
options = optimoptions('lsqlin','Algorithm','trust-region-reflective');  
[sol,fval,exitflag] = solve(nonneglsq,'Solver','lsqlin','Options',options);
```

Local minimum possible.

lsqlin stopped because the relative change in function value is less than the function

```
disp(sol.x(8))
```

```
2.2800e-15
```

```
options = optimoptions('quadprog','Algorithm','trust-region-reflective');  
[sol3,fval3,exitflag3] = solve(nonneglsq,'Solver','quadprog','Options',options);
```

Local minimum possible.

quadprog stopped because the relative change in function value is less than the sqrt of

```
disp(sol3.x(8))
```

```
6.7615e-13
```

The solutions are slightly closer to zero using the 'trust-region-reflective' algorithm than the default interior-point algorithms.

## See Also

### More About

- “Linear Least Squares with Bound Constraints” on page 12-31

## 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 12-42.

### 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  $m \times n$  matrix  $P$ .

**Add Motion**

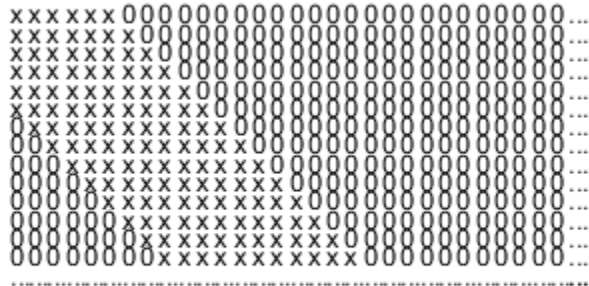
Simulate the effect of vertical motion blurring by averaging each pixel with the 5 pixels above and below. Construct a sparse matrix  $D$  to blur with a single matrix multiply.

```
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  $x$ :

$$\min(\|Dx - G\|^2) \text{ subject to } 0 \leq x \leq 1.$$

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);
```

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')
```



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:

$$\min(\|(D + \varepsilon I)x - G\|^2) \text{ subject to } 0 \leq x \leq 1.$$

The term  $\varepsilon I$  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);
```

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')
```



Apparently, this simple regularization does not remove the artifacts.

## See Also

### More About

- “Large-Scale Constrained Linear Least-Squares, Solver-Based” on page 12-42

## Nonlinear Curve Fitting with `lsqcurvefit`

`lsqcurvefit` enables you to fit parametrized nonlinear functions to data easily. You can use `lsqnonlin` as well; `lsqcurvefit` is simply a convenient way to call `lsqnonlin` for curve fitting.

In this example, the vector `xdata` represents 100 data points, and the vector `ydata` represents the associated measurements. Generate the data using the following script:

```
rng(5489,'twister') % reproducible
xdata = -2*log(rand(100,1));
ydata = (ones(100,1) + .1*randn(100,1)) + (3*ones(100,1)+...
    0.5*randn(100,1)).*exp(-(2*ones(100,1)+...
    .5*randn(100,1)).*xdata);
```

The modeled relationship between `xdata` and `ydata` is

$$ydata_i = a_1 + a_2 \exp(-a_3 xdata_i) + \varepsilon_i. \quad (12-14)$$

The script generates `xdata` from 100 independent samples from an exponential distribution with mean 2. It generates `ydata` from “Equation 12-14” using `a = [1;3;2]`, perturbed by adding normal deviates with standard deviations `[0.1;0.5;0.5]`.

The goal is to find parameters  $\hat{a}_i$ ,  $i = 1, 2, 3$ , for the model that best fit the data.

In order to fit the parameters to the data using `lsqcurvefit`, you need to define a fitting function. Define the fitting function `predicted` as an anonymous function:

```
predicted = @(a,xdata) a(1)*ones(100,1)+a(2)*exp(-a(3)*xdata);
```

To fit the model to the data, `lsqcurvefit` needs an initial estimate `a0` of the parameters. Enter

```
a0 = [2;2;2];
```

Run the solver `lsqcurvefit` as follows:

```
[ahat,resnorm,residual,exitflag,output,lambda,jacobian] =...
    lsqcurvefit(predicted,a0,xdata,ydata);
```

Local minimum possible.

`lsqcurvefit` stopped because the final change in the

sum of squares relative to its initial value is less than the default value of the function tolerance.

To see the resulting least-squares estimate of  $\hat{a}$ , enter:

```
ahat
```

```
ahat =  
    1.0169  
    3.1444  
    2.1596
```

The fitted values `ahat` are within 8% of `a = [1;3;2]`.

If you have Statistics and Machine Learning Toolbox software, use the `nlparci` function to generate confidence intervals for the `ahat` estimate.

## See Also

`lsqcurvefit` | `nlparci`

## More About

- “Nonlinear Data-Fitting” on page 12-11
- “Nonlinear Least Squares With and Without Jacobian” on page 12-27

## 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
cplxxydata = 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 `cplxxydata` for  $y$ ) is:

```
objfcn = @(v)v(1)+v(2).*exp(v(3)*xdata) - cplxxydata;
```

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,cplxdata,[],[],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:

$$\begin{aligned}y &= v_1 + iv_2 + (v_3 + iv_4) \exp((v_5 + iv_6)x) \\ &= (v_1 + v_3 \exp(v_5x) \cos(v_6x) - v_4 \exp(v_5x) \sin(v_6x)) \\ &\quad + i(v_2 + v_4 \exp(v_5x) \cos(v_6x) + v_3 \exp(v_5x) \sin(v_6x)).\end{aligned}$$

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(cplxdata),imag(cplxdata)];
```

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 2-20

## 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

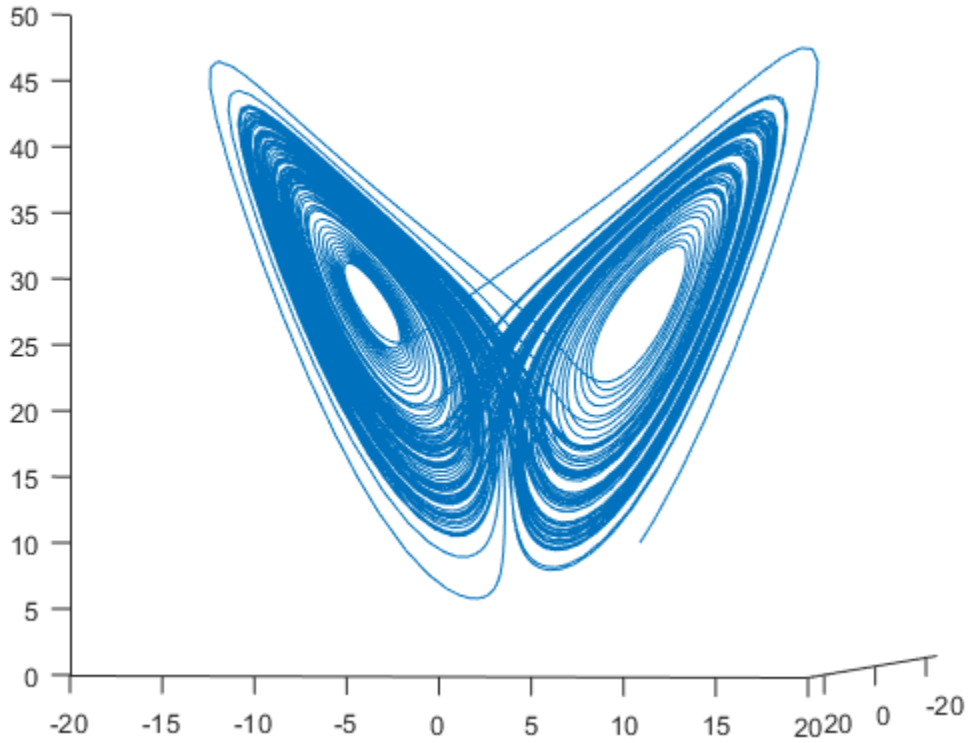
$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z.$$

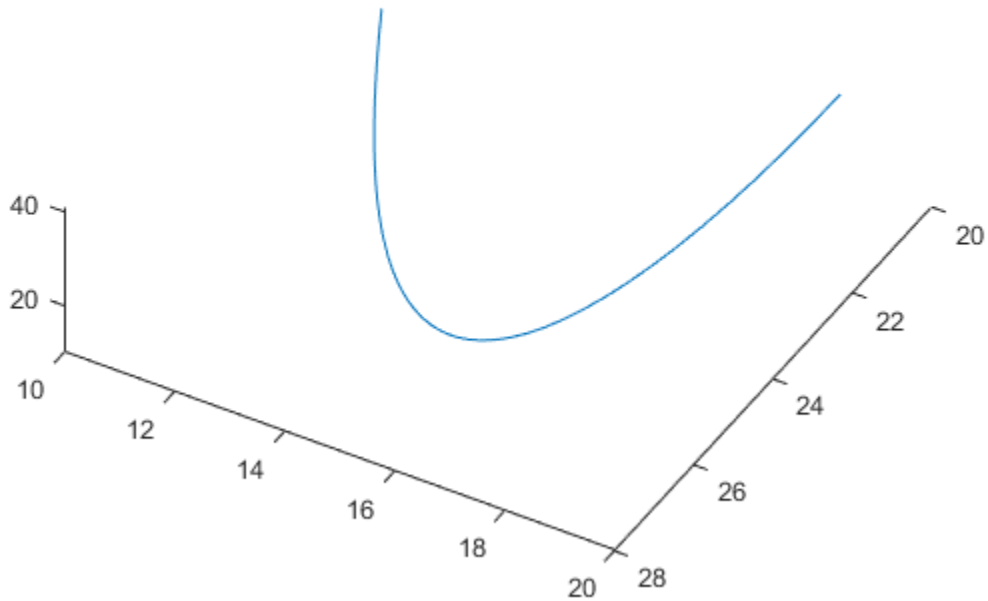
Lorenz's values of the parameters for a sensitive system are  $\sigma = 10$ ,  $\beta = 8/3$ ,  $\rho = 28$ . Start the system from  $[x(0), y(0), z(0)] = [10, 20, 10]$  and view the evolution of the system from time 0 through 100.

```
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  $t_0$  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.

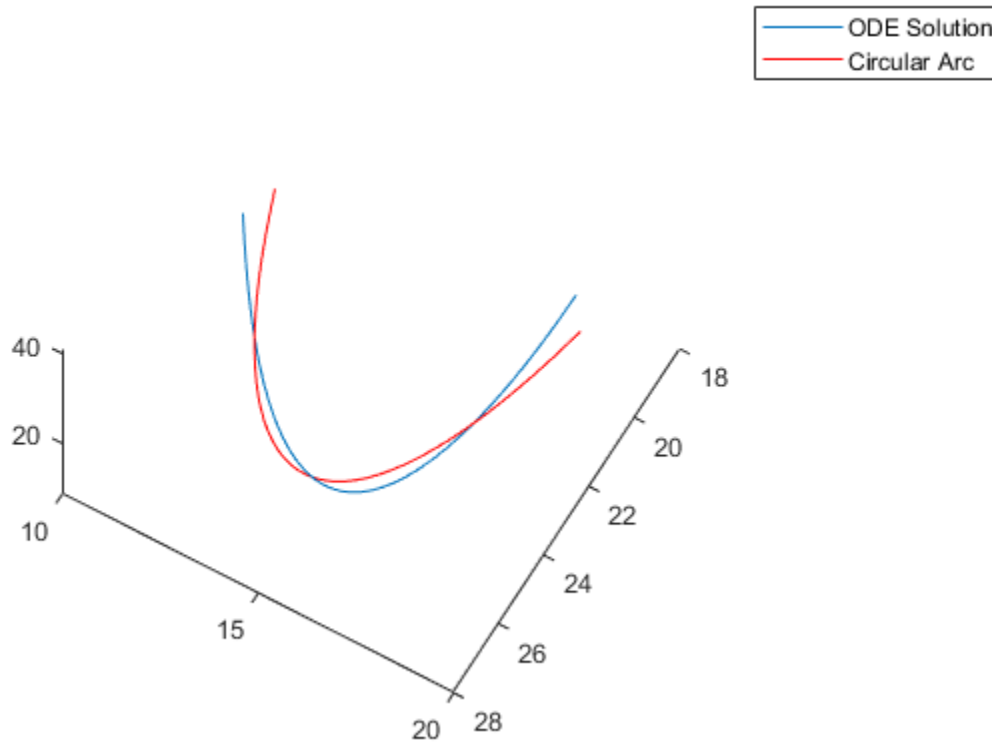
```
lb = [-pi/2, -pi, 5, -15, -pi, -40, -40, -40];
ub = [pi/2, pi, 60, 15, pi, 40, 40, 40];
theta0 = [0;0];
R0 = 20;
V0 = 1;
t0 = 0;
delta0 = zeros(3,1);
x0 = [theta0;R0;V0;t0;delta0];
[xbest,resnorm,residual] = lsqcurvefit(@fitlorenzfn,x0,tspan,a,lb,ub);
```

Local minimum possible.

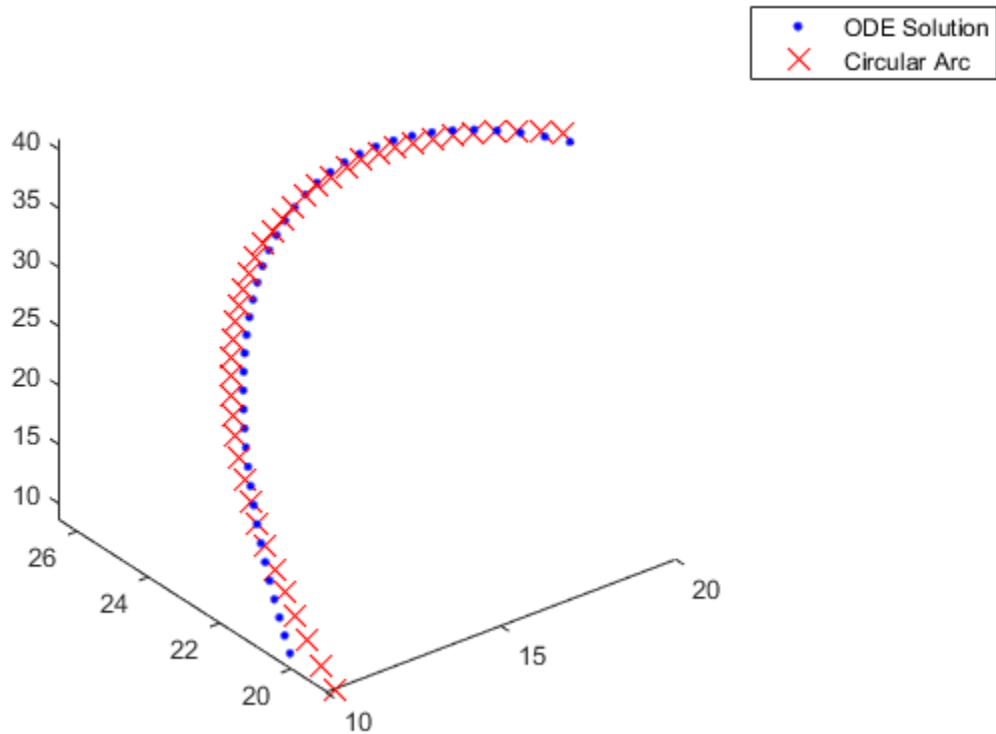
`lsqcurvefit` stopped because the final change in the sum of squares relative to its initial value is less than the value of the function tolerance.

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  $\sigma$ ,  $\beta$ , and  $\rho$  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
```

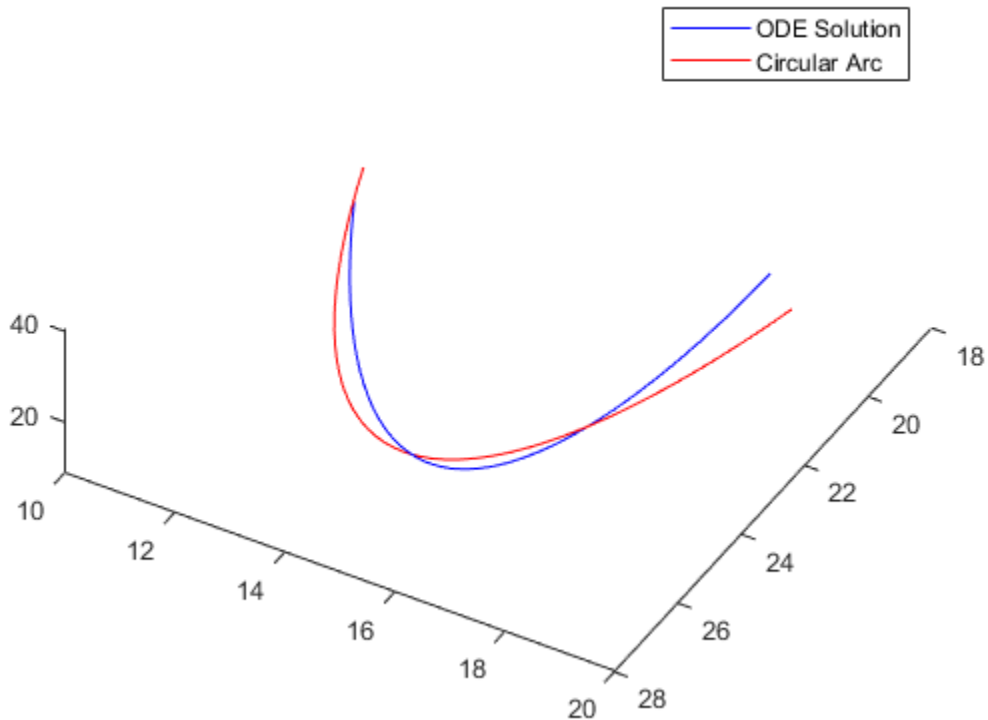
```
fprintf('Original parameters: %f, %f, %f',[sigma,beta,rho])
```

```
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 4-32, 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 4-32



# Systems of Equations

---

- “Equation Solving Algorithms” on page 13-2
- “Nonlinear Equations with Analytic Jacobian” on page 13-9
- “Nonlinear Equations with Finite-Difference Jacobian” on page 13-12
- “Nonlinear Equations with Jacobian” on page 13-14
- “Nonlinear Equations with Jacobian Sparsity Pattern” on page 13-17
- “Nonlinear Systems with Constraints” on page 13-20

## Equation Solving Algorithms

In this section...
“Equation Solving Definition” on page 13-2
“Trust-Region fsolve Algorithm” on page 13-2
“Trust-Region Dogleg Method” on page 13-5
“Levenberg-Marquardt Method” on page 13-7
“\ Algorithm” on page 13-7
“fzero Algorithm” on page 13-7

### Equation Solving Definition

Given a set of  $n$  nonlinear functions  $F_i(x)$ , where  $n$  is the number of components of the vector  $x$ , the goal of equation solving is to find a vector  $x$  that makes all  $F_i(x) = 0$ .

`fsolve` attempts to solve systems of equations by minimizing the sum of squares of the components. If the sum of squares is zero, the system of equation is solved. `fsolve` has three algorithms:

- Trust-region
- Trust-region dogleg
- Levenberg-Marquardt

All algorithms are large-scale; see “Large-Scale vs. Medium-Scale Algorithms” on page 2-14.

The `fzero` function solves a single one-dimensional equation.

The `mldivide` function solves systems of linear equations.

### Trust-Region fsolve Algorithm

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

To understand the trust-region approach to optimization, consider the unconstrained minimization problem, minimize  $f(x)$ , where the function takes vector arguments and

returns scalars. Suppose you are at a point  $x$  in  $n$ -space and you want to improve, i.e., move to a point with a lower function value. The basic idea is to approximate  $f$  with a simpler function  $q$ , which reasonably reflects the behavior of function  $f$  in a neighborhood  $N$  around the point  $x$ . This neighborhood is the trust region. A trial step  $s$  is computed by minimizing (or approximately minimizing) over  $N$ . This is the trust-region subproblem,

$$\min_s \{q(s), s \in N\}. \quad (13-1)$$

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

$$\min \left\{ \frac{1}{2} s^T H s + s^T g \text{ such that } \|Ds\| \leq \Delta \right\}, \quad (13-2)$$

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,  $\Delta$  is a positive scalar, and  $\| \cdot \|$  is the 2-norm. Good algorithms exist for solving “Equation 13-2” (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 13-2”. 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 13-2”, 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 13-2” 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

$$H \cdot s_2 = -g, \tag{13-3}$$

or a direction of negative curvature,

$$s_2^T \cdot H \cdot s_2 < 0. \tag{13-4}$$

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 13-2” to determine the trial step  $s$ .
- 3 If  $f(x + s) < f(x)$ , then  $x = x + s$ .
- 4 Adjust  $\Delta$ .

These four steps are repeated until convergence. The trust-region dimension  $\Delta$  is adjusted according to standard rules. In particular, it is decreased if the trial step is not accepted, i.e.,  $f(x + s) \geq f(x)$ . See [46] and [49] for a discussion of this aspect.

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

### **Preconditioned Conjugate Gradient Method**

A popular way to solve large symmetric positive definite systems of linear equations  $Hp = -g$  is the method of Preconditioned Conjugate Gradients (PCG). This iterative approach requires the ability to calculate matrix-vector products of the form  $H \cdot v$  where  $v$  is an arbitrary vector. The symmetric positive definite matrix  $M$  is a *preconditioner* for  $H$ .



That is,  $M = C^2$ , where  $C^{-1}HC^{-1}$  is a well-conditioned matrix or a matrix with clustered eigenvalues.

In a minimization context, you can assume that the Hessian matrix  $H$  is symmetric. However,  $H$  is guaranteed to be positive definite only in the neighborhood of a strong minimizer. Algorithm PCG exits when a direction of negative (or zero) curvature is encountered, i.e.,  $d^T H d \leq 0$ . The PCG output direction,  $p$ , is either a direction of negative curvature or an approximate ( $tol$  controls how approximate) solution to the Newton system  $H p = -g$ . In either case  $p$  is used to help define the two-dimensional subspace used in the trust-region approach discussed in "Trust-Region Methods for Nonlinear Minimization" on page 6-2.

## Trust-Region Dogleg Method

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

$$J(x_k) d_k = -F(x_k)$$

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

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

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

Using trust-region techniques (introduced in "Trust-Region Methods for Nonlinear Minimization" on page 6-2) improves robustness when starting far from the solution and handles the case when  $J(x_k)$  is singular. To use a trust-region strategy, a merit function is needed to decide if  $x_{k+1}$  is better or worse than  $x_k$ . A possible choice is

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

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

The Newton step  $d_k$  is a root of

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

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

$$\begin{aligned} \min_d m(d) &= \frac{1}{2} \|M(x_k + d)\|_2^2 = \frac{1}{2} \|F(x_k) + J(x_k)d\|_2^2 \\ &= \frac{1}{2} F(x_k)^T F(x_k) + d^T J(x_k)^T F(x_k) + \frac{1}{2} d^T J(x_k)^T J(x_k) d. \end{aligned} \tag{13-5}$$

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

$$\min_d \left[ \frac{1}{2} F(x_k)^T F(x_k) + d^T J(x_k)^T F(x_k) + \frac{1}{2} d^T J(x_k)^T J(x_k) d \right], \tag{13-6}$$

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

For an overview of trust-region methods, see Conn [4], and Nocedal [31].

### Trust-Region Dogleg Implementation

The key feature of this algorithm is the use of the Powell dogleg procedure for computing the step  $d$ , which minimizes “Equation 13-6”. For a detailed description, see Powell [34].

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

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

where  $\alpha$  is chosen to minimize “Equation 13-5”.

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 step  $d$  is chosen so that

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

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

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

## Levenberg-Marquardt Method

The Levenberg-Marquardt [25], and [27] method uses a search direction that is a solution of the linear set of equations

$$\left( J(x_k)^T J(x_k) + \lambda_k I \right) d_k = - J(x_k)^T F(x_k), \quad (13-7)$$

or, optionally, of the equations

$$\left( J(x_k)^T J(x_k) + \lambda_k \text{diag}\left( J(x_k)^T J(x_k) \right) \right) d_k = - J(x_k)^T F(x_k), \quad (13-8)$$

where the scalar  $\lambda_k$  controls both the magnitude and direction of  $d_k$ . Set option `ScaleProblem` to 'none' to choose "Equation 13-7", and set `ScaleProblem` to 'Jacobian' to choose "Equation 13-8".

When  $\lambda_k$  is zero, the direction  $d_k$  is the Gauss-Newton method. As  $\lambda_k$  tends to infinity,  $d_k$  tends towards the steepest descent direction, with magnitude tending to zero. This implies that for some sufficiently large  $\lambda_k$ , the term  $F(x_k + d_k) < F(x_k)$  holds true. The term  $\lambda_k$  can therefore be controlled to ensure descent even when second-order terms, which restrict the efficiency of the Gauss-Newton method, are encountered. The Levenberg-Marquardt method therefore uses a search direction that is a cross between the Gauss-Newton direction and the steepest descent direction.

## \ Algorithm

This algorithm is described in the MATLAB arithmetic operators section for `mldivide`.

## fzero Algorithm

`fzero` attempts to find the root of a scalar function  $f$  of a scalar variable  $x$ .

`fzero` looks for an interval around an initial point such that  $f(x)$  changes sign. If you give an initial interval instead of an initial point, `fzero` checks to make sure  $f(x)$  has different signs at the endpoints of the interval. The initial interval must be finite; it cannot contain  $\pm\text{Inf}$ .

`fzero` uses a combination of interval bisection, linear interpolation, and inverse quadratic interpolation in order to locate a root of  $f(x)$ . See `fzero` for more information.

## Nonlinear Equations with Analytic Jacobian

This example demonstrates the use of the default trust-region-dogleg `fsolve` algorithm (see “Large-Scale vs. Medium-Scale Algorithms” on page 2-14). It is intended for problems where

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

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

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

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

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

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

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

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

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

### Step 1: Write a file `bananaobj.m` to compute the objective function values and the Jacobian.

```
function [F,J] = bananaobj(x)
% Evaluate the vector function and the Jacobian matrix for
% the system of nonlinear equations derived from the general
% n-dimensional Rosenbrock function.
% Get the problem size
n = length(x);
if n == 0, error('Input vector, x, is empty.');
```

`end`

```
if mod(n,2) ~= 0,
    error('Input vector, x ,must have an even number of components.');
```

`end`

```
% Evaluate the vector function
odds = 1:2:n;
evens = 2:2:n;
F = zeros(n,1);
F(odds,1) = 1-x(odds);
F(evens,1) = 10.*(x(evens)-x(odds).^2);
% Evaluate the Jacobian matrix if nargin > 1
if nargin > 1
    c = -ones(n/2,1);    C = sparse(odds,odds,c,n,n);
    d = 10*ones(n/2,1); D = sparse(evens,evens,d,n,n);
    e = -20.*x(odds);   E = sparse(evens,odds,e,n,n);
    J = C + D + E;
end
```

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

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

Use the starting point  $x(i) = -1.9$  for the odd indices, and  $x(i) = 2$  for the even indices. Set `Display` to `'iter'` to see the solver's progress. Set `SpecifyObjectiveGradient` to `true` to use the Jacobian defined in `bananaobj.m`. The `fsolve` function generates the following output:

Iteration	Func-count	f(x)	Norm of step	First-order optimality	Trust-region radius
0	1	8563.84		615	1
1	2	3093.71	1	329	1
2	3	225.104	2.5	34.8	2.5
3	4	212.48	6.25	34.1	6.25
4	5	212.48	6.25	34.1	6.25
5	6	102.771	1.5625	6.39	1.56
6	7	102.771	3.90625	6.39	3.91
7	8	87.7443	0.976563	2.19	0.977
8	9	74.1426	2.44141	6.27	2.44
9	10	74.1426	2.44141	6.27	2.44
10	11	52.497	0.610352	1.52	0.61
11	12	41.3297	1.52588	4.63	1.53
12	13	34.5115	1.52588	6.97	1.53
13	14	16.9716	1.52588	4.69	1.53
14	15	8.16797	1.52588	3.77	1.53
15	16	3.55178	1.52588	3.56	1.53
16	17	1.38476	1.52588	3.31	1.53
17	18	0.219553	1.16206	1.66	1.53
18	19	0	0.0468565	0	1.53

Equation solved.

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

## Nonlinear Equations with Finite-Difference Jacobian

In the example “Nonlinear Equations with Analytic Jacobian” on page 13-9, the function `bananaobj` evaluates  $F$  and computes the Jacobian  $J$ . What if the code to compute the Jacobian is not available? By default, if you do not indicate that the Jacobian can be computed in the objective function (by setting the `SpecifyObjectiveGradient` option in `options` to `true`), `fsolve`, `lsqnonlin`, and `lsqcurvefit` instead use finite differencing to approximate the Jacobian. This is the default Jacobian option. You can select finite differencing by setting `SpecifyObjectiveGradient` to `false` using `optimoptions`.

This example uses `bananaobj` from the example “Nonlinear Equations with Analytic Jacobian” on page 13-9 as the objective function, but sets `SpecifyObjectiveGradient` to `false` so that `fsolve` approximates the Jacobian and ignores the second `bananaobj` output.

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

The example produces the following output:

Iteration	Func-count	f(x)	Norm of step	First-order optimality	Trust-region radius
0	65	8563.84		615	1
1	130	3093.71	1	329	1
2	195	225.104	2.5	34.8	2.5
3	260	212.48	6.25	34.1	6.25
4	261	212.48	6.25	34.1	6.25
5	326	102.771	1.5625	6.39	1.56
6	327	102.771	3.90625	6.39	3.91
7	392	87.7443	0.976562	2.19	0.977
8	457	74.1426	2.44141	6.27	2.44
9	458	74.1426	2.44141	6.27	2.44
10	523	52.497	0.610352	1.52	0.61
11	588	41.3297	1.52588	4.63	1.53
12	653	34.5115	1.52588	6.97	1.53
13	718	16.9716	1.52588	4.69	1.53
14	783	8.16797	1.52588	3.77	1.53
15	848	3.55178	1.52588	3.56	1.53
16	913	1.38476	1.52588	3.31	1.53
17	978	0.219553	1.16206	1.66	1.53
18	1043	0	0.0468565	0	1.53

Equation solved.



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

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

## Nonlinear Equations with Jacobian

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

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

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

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

To solve a large nonlinear system of equations,  $F(x) = 0$ , you can use the trust-region reflective algorithm available in `fsolve`, a large-scale algorithm (“Large-Scale vs. Medium-Scale Algorithms” on page 2-14).

### Step 1: Write a file `nlsf1.m` that computes the objective function values and the Jacobian.

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

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

```
xstart = -ones(1000,1);
fun = @nlsf1;
options = optimoptions(@fsolve,'Display','iter',...
    'Algorithm','trust-region',...
```

```
'SpecifyObjectiveGradient',true,'PrecondBandWidth',0);
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

A starting point is given as well as the function name. The default method for `fsolve` is trust-region-dogleg, so it is necessary to specify `'Algorithm'` as `'trust-region'` in the options argument in order to select the trust-region algorithm. Setting the `Display` option to `'iter'` causes `fsolve` to display the output at each iteration. Setting `'SpecifyObjectiveGradient'` to `true`, causes `fsolve` to use the Jacobian information available in `nlsf1.m`.

The commands display this output:

Iteration	Func-count	f(x)	Norm of step	First-order optimality
0	1	1011		19
1	2	16.1942	7.91898	2.35
2	3	0.0228027	1.33142	0.291
3	4	0.000103359	0.0433329	0.0201
4	5	7.3792e-07	0.0022606	0.000946
5	6	4.02299e-10	0.000268381	4.12e-05

Equation solved, inaccuracy possible.

`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.

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

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

If you want to use a tridiagonal preconditioner, i.e., a preconditioning matrix with three diagonals (or bandwidth of one), set `PrecondBandWidth` to the value 1:

```
options = optimoptions(@fsolve,'Display','iter','SpecifyObjectiveGradient',true,...
    'Algorithm','trust-region','PrecondBandWidth',1);
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

In this case the output is

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

0	1	1011		19
1	2	16.0839	7.92496	1.92
2	3	0.0458181	1.3279	0.579
3	4	0.000101184	0.0631898	0.0203
4	5	3.16615e-07	0.00273698	0.00079
5	6	9.72481e-10	0.00018111	5.82e-05

Equation solved, inaccuracy possible.

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.

Note that although the same number of iterations takes place, the number of PCG iterations has dropped, so less work is being done per iteration. See “Preconditioned Conjugate Gradient Method” on page 6-24.

Setting PrecondBandWidth to Inf (this is the default) means that the solver uses Cholesky factorization rather than PCG.

## Nonlinear Equations with Jacobian Sparsity Pattern

In the example “Nonlinear Equations with Analytic Jacobian” on page 13-9, the function `nlsf1` computes the Jacobian  $J$ , a sparse matrix, along with the evaluation of  $F$ . What if the code to compute the Jacobian is not available? By default, if you do not indicate that the Jacobian can be computed in `nlsf1` (by setting the `'SpecifyObjectiveGradient'` option in `options` to `true`), `fsolve`, `lsqnonlin`, and `lsqcurvefit` instead uses finite differencing to approximate the Jacobian.

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

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

Suppose the sparse matrix `Jstr`, computed previously, has been saved in file `nlsdat1.mat`. The following driver calls `fsolve` applied to `nlsf1a`, which is `nlsf1` without the Jacobian. Sparse finite-differencing is used to estimate the sparse Jacobian matrix as needed.

### Step 1: Write a file `nlsf1a.m` that computes the objective function values.

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

```
F(n) = (3-2*x(n)).*x(n)-x(n-1) + 1;  
F(1) = (3-2*x(1)).*x(1)-2*x(2) + 1;
```

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

```
xstart = -ones(1000,1);  
fun = @nlsfla;  
load nlsdat1 % Get Jstr  
options = optimoptions(@fsolve,'Display','iter','JacobPattern',Jstr,...  
    'Algorithm','trust-region','SubproblemAlgorithm','cg');  
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

In this case, the output displayed is

Iteration	Func-count	f(x)	Norm of step	First-order optimality
0	5	1011		19
1	10	16.1942	7.91898	2.35
2	15	0.0228025	1.33142	0.291
3	20	0.00010336	0.0433327	0.0201
4	25	7.37924e-07	0.0022606	0.000946
5	30	4.02301e-10	0.000268382	4.12e-05

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.

Alternatively, it is possible to choose a sparse direct linear solver (i.e., a sparse QR factorization) by indicating a “complete” preconditioner. For example, if you set PrecondBandWidth to Inf, then a sparse direct linear solver is used instead of a preconditioned conjugate gradient iteration:

```
xstart = -ones(1000,1);  
fun = @nlsfla;  
load nlsdat1 % Get Jstr  
options = optimoptions(@fsolve,'Display','iter','JacobPattern',Jstr,...  
    'Algorithm','trust-region','SubproblemAlgorithm','factorization');  
[x,fval,exitflag,output] = fsolve(fun,xstart,options);
```

and the resulting display is

Iteration	Func-count	f(x)	Norm of step	First-order optimality
0	5	1011		19
1	10	15.9018	7.92421	1.89
2	15	0.0128161	1.32542	0.0746
3	20	1.73502e-08	0.0397923	0.000196

4            25            1.10732e-18            4.55495e-05            2.74e-09

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.

When using the sparse direct solver, there are no CG iterations. Notice that the final optimality and  $f(x)$  value (which for `fsolve`,  $f(x)$ , is the sum of the squares of the function values) are closer to zero than using the PCG method, which is often the case.

## Nonlinear Systems with Constraints

### In this section...

“Solve Equations with Inequality Constraints” on page 13-20

“Use Different Start Points” on page 13-21

“Use Different Algorithms” on page 13-21

“Use lsqnonlin with Bounds” on page 13-22

“Set Equations and Inequalities as fmincon Constraints” on page 13-23

### Solve Equations with Inequality Constraints

`fsolve` solves systems of nonlinear equations. However, it does not allow you to include any constraints, even bound constraints. The question is, how can you solve systems of nonlinear equations when you have constraints?

The short answer is, there are no guarantees that a solution exists that satisfies your constraints. There is no guarantee that any solution exists, even one that does not satisfy your constraints. Nevertheless, there are techniques that can help you search for solutions that satisfy your constraints.

To illustrate the techniques, consider how to solve the equations

$$F_1(x) = (x_1 + 1)(10 - x_1) \frac{1 + x_2^2}{1 + x_2^2 + x_2} \tag{13-9}$$

$$F_2(x) = (x_2 + 2)(20 - x_2) \frac{1 + x_1^2}{1 + x_1^2 + x_1},$$

where the components of  $x$  must be nonnegative. Clearly, there are four solutions to the equations:

$$\begin{array}{rcl} x & = & (-1, -2) \\ x & = & (10, -2), \\ x & = & (-1, 20), \\ x & = & (10, 20). \end{array}$$

There is only one solution that satisfies the constraints, namely  $x = (10, 20)$ .



To solve the equations numerically, first enter code to calculate  $F(x)$ .

```
function F = fbnd(x)
F(1) = (x(1)+1)*(10-x(1))*(1+x(2)^2)/(1+x(2)^2+x(2));
F(2) = (x(2)+2)*(20-x(2))*(1+x(1)^2)/(1+x(1)^2+x(1));
```

Save this code as the file `fbnd.m` on your MATLAB path.

## Use Different Start Points

Generally, a system of  $N$  equations in  $N$  variables has isolated solutions, meaning each solution has no nearby neighbors that are also solutions. So one way to search for a solution that satisfies some constraints is to generate a number of initial points  $x_0$ , and run `fsolve` starting at each  $x_0$ .

For this example, to look for a solution to “Equation 13-9”, take 10 random points that are normally distributed with mean 0 and standard deviation 100.

```
rng default % for reproducibility
N = 10; % try 10 random start points
pts = 100*randn(N,2); % initial points are rows in pts
soln = zeros(N,2); % allocate solution
opts = optimoptions('fsolve','Display','off');
for k = 1:N
    soln(k,:) = fsolve(@fbnd,pts(k,:),opts); % find solutions
end
```

Examine the solutions in `soln`, and you find several that satisfy the constraints.

## Use Different Algorithms

There are three `fsolve` algorithms. Each can lead to different solutions.

For this example, take  $x_0 = [1, 9]$  and examine the solution each algorithm returns.

```
x0 = [1,9];
opts = optimoptions(@fsolve,'Display','off',...
    'Algorithm','trust-region-dogleg');
x1 = fsolve(@fbnd,x0,opts)
```

```
x1 =  
    -1.0000    -2.0000  
  
opts.Algorithm = 'trust-region';  
x2 = fsolve(@fbnd,x0,opts)  
  
x2 =  
    -1.0000    20.0000  
  
opts.Algorithm = 'levenberg-marquardt';  
x3 = fsolve(@fbnd,x0,opts)  
  
x3 =  
    0.9523    8.9941
```

Here, all three algorithms find different solutions for the same initial point. In fact, `x3` is not even a solution, but is simply a locally stationary point.

## Use `lsqnonlin` with Bounds

`lsqnonlin` tries to minimize the sum of squares of the components of a vector function  $F(x)$ . Therefore, it attempts to solve the equation  $F(x) = 0$ . Furthermore, `lsqnonlin` accepts bound constraints.

Formulate the example problem for `lsqnonlin` and solve it.

```
lb = [0,0];  
rng default  
x0 = 100*randn(2,1);  
[x,res] = lsqnonlin(@fbnd,x0,lb)  
  
x =  
    10.0000  
    20.0000  
  
res =
```

2.4783e-25

You can use `lsqnonlin` with the Global Optimization Toolbox `MultiStart` solver to search over many initial points automatically. See “MultiStart Using `lsqcurvefit` or `lsqnonlin`” (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  $\theta(x)$ , which evaluates to  $\theta$  for each  $x$ .
- Set the `fsolve` objective function as the nonlinear equality constraints in `fmincon`.
- Give any other constraints in the usual `fmincon` syntax.

For this example, write a function file for the nonlinear inequality constraint.

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

c = []; % no nonlinear inequality
ceq = fbnd(x); % the fsolve objective is fmincon constraints
```

Save this code as the file `fminconstr.m` on your MATLAB path.

Solve the constrained problem.

```
lb = [0,0]; % lower bound constraint
rng default % reproducible initial point
x0 = 100*randn(2,1);
opts = optimoptions(@fmincon,'Algorithm','interior-point','Display','off');
x = fmincon(@(x)theta,x0,[],[],[],[],lb,[],@fminconstr,opts)
```

`x =`

```
10.0000
20.0000
```

## See Also

`fmincon` | `fsolve` | `lsqnonlin`

**More About**

- “Systems of Nonlinear Equations”

# Parallel Computing for Optimization

---

- “What Is Parallel Computing in Optimization Toolbox?” on page 14-2
- “Using Parallel Computing in Optimization Toolbox” on page 14-6
- “Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™” on page 14-10
- “Improving Performance with Parallel Computing” on page 14-18

## What Is Parallel Computing in Optimization Toolbox?

<b>In this section...</b>
“Parallel Optimization Functionality” on page 14-2
“Parallel Estimation of Gradients” on page 14-3
“Nested Parallel Functions” on page 14-4

### Parallel Optimization Functionality

Parallel computing is the technique of using multiple processors on a single problem. The reason to use parallel computing is to speed computations.

The 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

$$\nabla f(x) \approx \left[ \frac{f(x + \Delta_1 e_1) - f(x)}{\Delta_1}, \frac{f(x + \Delta_2 e_2) - f(x)}{\Delta_2}, \dots, \frac{f(x + \Delta_n e_n) - f(x)}{\Delta_n} \right],$$

where

- $f$  represents objective or constraint functions
- $e_i$  are the unit direction vectors
- $\Delta_i$  is the size of a step in the  $e_i$  direction

To estimate  $\nabla f(x)$  in parallel, Optimization Toolbox solvers distribute the evaluation of  $(f(x + \Delta_i e_i) - f(x))/\Delta_i$  to extra processors.

## Parallel Central Differences

You can choose to have gradients estimated by central finite differences instead of the default forward finite differences. The basic central finite difference formula is

$$\nabla f(x) \approx \left[ \frac{f(x + \Delta_1 e_1) - f(x - \Delta_1 e_1)}{2\Delta_1}, \dots, \frac{f(x + \Delta_n e_n) - f(x - \Delta_n e_n)}{2\Delta_n} \right].$$

This takes twice as many function evaluations as forward finite differences, but is usually much more accurate. Central finite differences work in parallel exactly the same as forward finite differences.

Enable central finite differences by using `optimoptions` to set the `FiniteDifferenceType` option to 'central'. To use forward finite differences, set the `FiniteDifferenceType` option to 'forward'.

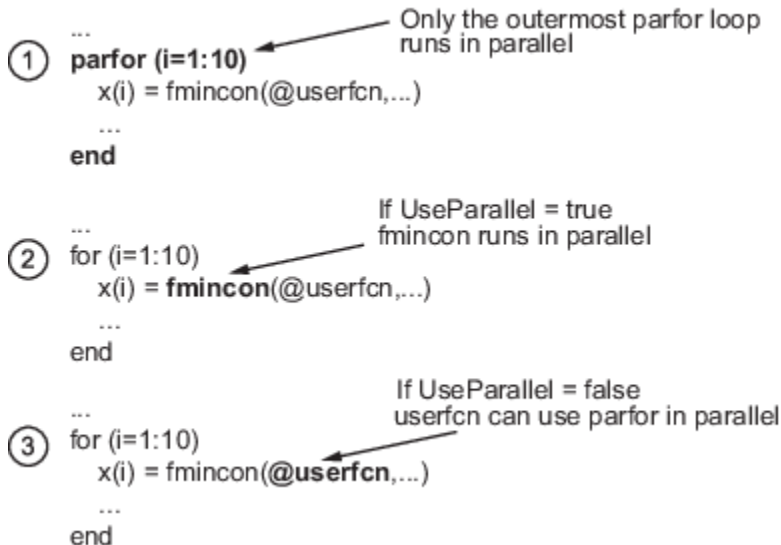
## Nested Parallel Functions

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.

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



### When `parfor` Runs In Parallel



## See Also

“Using Parallel Computing in Optimization Toolbox” on page 14-6 | “Improving Performance with Parallel Computing” on page 14-18 | “Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™” on page 14-10

## Using Parallel Computing in Optimization Toolbox

### In this section...

“Using Parallel Computing with Multicore Processors” on page 14-6

“Using Parallel Computing with a Multiprocessor Network” on page 14-7

“Deploy Parallel Optimization” on page 14-8

“Testing Parallel Computations” on page 14-8

### Using Parallel Computing with Multicore Processors

If you have a multicore processor, you might see speedup using parallel processing. You can establish a parallel pool of several workers with a Parallel Computing Toolbox license. For a description of Parallel Computing Toolbox software, see “Getting Started with Parallel Computing Toolbox” (Parallel Computing Toolbox).

Suppose you have a dual-core processor, and want to use parallel computing:

- Enter

```
parpool
```

at the command line. MATLAB starts a pool of workers using the multicore processor. If you had previously set a nondefault cluster profile, you can enforce multicore (local) computing:

```
parpool('local')
```

---

**Note** Depending on your preferences, MATLAB can start a parallel pool automatically. To enable this feature, check **Automatically create a parallel pool** in **Home > Parallel > Parallel Preferences**.

---

- For command-line use, enter
 

```
options = optimoptions('solvername','UseParallel',true);
```
- For Optimization app, check **Options > Approximated derivatives > Evaluate in parallel**.

When you run an applicable solver with `options`, applicable solvers automatically use parallel computing.

To stop computing optimizations in parallel, set `UseParallel` to `false`, or set the Optimization app not to compute in parallel. To halt all parallel computation, enter

```
delete(gcp)
```

## Using Parallel Computing with a Multiprocessor Network

If you have multiple processors on a network, use Parallel Computing Toolbox functions and MATLAB Parallel Server™ software to establish parallel computation. Here are the steps to take:

- 1 Make sure your system is configured properly for parallel computing. Check with your systems administrator, or refer to the Parallel Computing Toolbox documentation.

To perform a basic check:

- a At the command line, enter

```
parpool(profile)
```

where `profile` is your cluster profile.

- b Workers must be able to access your objective function file and, if applicable, your nonlinear constraint function file. There are two ways of ensuring access:

- i Distribute the files to the workers using the `parpool AttachedFiles` argument. For example, if `objfun.m` is your objective function file, and `constrfun.m` is your nonlinear constraint function file, enter

```
parpool('AttachedFiles',{ 'objfun.m', 'constrfun.m' });
```

Workers access their own copies of the files.

- ii Give a network file path to your files. If `network_file_path` is the network path to your objective or constraint function files, enter

```
pctRunOnAll('addpath network_file_path')
```

Workers access the function files over the network.

- c Check whether a file is on the path of every worker by entering

```
pctRunOnAll('which filename')
```

If any worker does not have a path to the file, it reports

*filename* not found.

- 2 • For command-line use, enter

```
options = optimoptions('solvername','UseParallel',true);
```

where 'solvername' represents one of the nonlinear solvers that support parallel evaluation.

- For Optimization app, check **Options > Approximated derivatives > Evaluate in parallel**.

After you establish your parallel computing environment, applicable solvers automatically use parallel computing whenever you call them with `options`.

To stop computing optimizations in parallel, set `UseParallel` to `false`, or set the Optimization app not to compute in parallel. To halt all parallel computation, enter

```
delete(gcf)
```

## Deploy Parallel Optimization

If you deploy code that calls an optimization solver, and want the solver to use parallel computing, ensure that you explicitly create a parallel pool in your code. Otherwise, the deployed code can fail to run in parallel, and so run only in serial, because MATLAB Compiler™'s dependency analysis can fail to make parallel functionality available. For example, call `parpool` explicitly, in addition to setting the solver's `UseParallel` option to `true`.

## Testing Parallel Computations

To test see if a problem runs correctly in parallel,

- 1 Try your problem without parallel computation to ensure that it runs properly serially. Make sure this is successful (gives correct results) before going to the next test.
- 2 Set `UseParallel` to `true`, and ensure that there is no parallel pool using `delete(gcf)`. Uncheck **Automatically create a parallel pool** in **Home > Parallel > Parallel Preferences** so MATLAB does not create a parallel pool. Your problem runs `parfor` serially, with loop iterations in reverse order from a `for` loop. Make sure this is successful (gives correct results) before going to the next test.

- 3 Set `UseParallel` to `true`, and create a parallel pool using `parpool`. Unless you have a multicore processor or a network set up, you won't see any speedup. This testing is simply to verify the correctness of the computations.

Remember to call your solver using an options 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 14-18
- “Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™” on page 14-10

## 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);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of 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
```

Generation	Func-count	Best f(x)	Mean f(x)	Stall 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(gcf)) == 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);
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of 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.

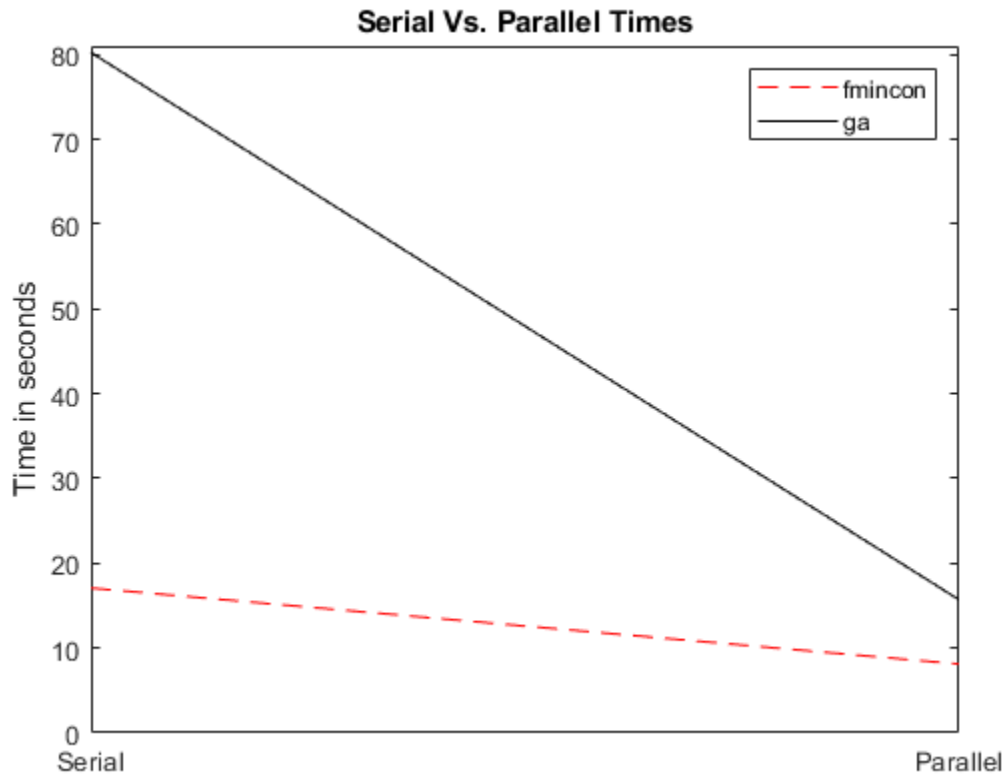
```
rng default % to get the same evaluations as the previous run
if gaAvailable
    gaoptions = optimoptions(gaoptions,'UseParallel',true);
    startTime = tic;
    gasol = ga(@expensive_objfun,nvar,[],[],[],[],[],[],[],gaoptions);
    time_ga_parallel = toc(startTime);
    fprintf('Parallel GA optimization takes %g seconds.\n',time_ga_parallel);
end
```

Generation	Func-count	Best f(x)	Mean f(x)	Stall 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

- “What Is Parallel Computing in Optimization Toolbox?” on page 14-2
- “Using Parallel Computing in Optimization Toolbox” on page 14-6

- “Improving Performance with Parallel Computing” on page 14-18

## Improving Performance with Parallel Computing

In this section...
“Factors That Affect Speed” on page 14-18
“Factors That Affect Results” on page 14-18
“Searching for Global Optima” on page 14-19

### 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 14-4. `parfor` does not work in parallel when called from within another `parfor` loop. If you have programmed your objective or constraint functions to take advantage of parallel processing, the limitation of no nested `parfor` loops may cause a solver to run more slowly than you expect. In particular, the parallel computation of finite differences takes precedence, since that is an outer loop. This causes any parallel code within the objective or constraint functions to execute serially.
- 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 14-6
- “Minimizing an Expensive Optimization Problem Using Parallel Computing Toolbox™” on page 14-10



# Argument and Options Reference

---

- “Function Input Arguments” on page 15-2
- “Function Output Arguments” on page 15-5
- “Optimization Options Reference” on page 15-8
- “Current and Legacy Option Name Tables” on page 15-31
- “Output Function Syntax” on page 15-37
- “Plot Function Syntax” on page 15-47
- “intlinprog Output Function and Plot Function Syntax” on page 15-48

## 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: $A*x \leq b$ .	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

Argument	Description	Used by Functions
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 $-\text{Inf}$ (for lower bounds) or $\text{Inf}$ (for upper bounds). For example, if $\text{lb}(i) = -\text{Inf}$ , the variable $x(i)$ is unbounded below.	fgoalattain, fmincon, fminimax, fseminf, linprog, lsqcurvefit, lsqlin, lsqnonlin, quadprog
nonlcon	The function that computes the nonlinear inequality and equality constraints. "Passing Extra Parameters" on page 2-70 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 15-8 or the individual function reference pages.	All functions

<b>Argument</b>	<b>Description</b>	<b>Used by Functions</b>
<code>seminfcon</code>	The function that computes the nonlinear inequality and equality constraints <i>and</i> the semi-infinite constraints. <code>seminfcon</code> is the name of a function file or MEX-file. “Passing Extra Parameters” on page 2-70 explains how to parametrize <code>seminfcon</code> , if necessary.  See the function reference pages for <code>fseminf</code> for more information on <code>seminfcon</code> .	<code>fseminf</code>
<code>weight</code>	A weighting vector to control the relative underattainment or overattainment of the objectives.	<code>fgoalattain</code>
<code>xdata, ydata</code>	The input data <code>xdata</code> and the observed output data <code>ydata</code> that are to be fitted to an equation.	<code>lsqcurvefit</code>
<code>x0</code>	Starting point (a scalar, vector or matrix).  (For <code>fzero</code> , <code>x0</code> can also be a two-element vector representing a finite interval that is known to contain a zero.)	All functions except <code>fminbnd</code> and <code>linprog</code>
<code>x1, x2</code>	The interval over which the function is minimized.	<code>fminbnd</code>

## See Also

### More About

- “Function Output Arguments” on page 15-5

## Function Output Arguments

Argument	Description	Used by Functions
<code>attainfactor</code>	The attainment factor at the solution <code>x</code> .	<code>fgoalattain</code>
<code>exitflag</code>	<p>An integer identifying the reason the optimization algorithm terminated. See the function reference pages for descriptions of <code>exitflag</code> specific to each function, and “Exit Flags and Exit Messages” on page 3-3.</p> <p>You can also return a message stating why an optimization terminated by calling the optimization function with the output argument <code>output</code> and then displaying <code>output.message</code>.</p>	All functions
<code>fval</code>	The value of the objective function <code>fun</code> at the solution <code>x</code> .	<code>fgoalattain</code> , <code>fminbnd</code> , <code>fmincon</code> , <code>fminimax</code> , <code>fminsearch</code> , <code>fminunc</code> , <code>fseminf</code> , <code>fsolve</code> , <code>fzero</code> , <code>linprog</code> , <code>quadprog</code>
<code>grad</code>	The value of the gradient of <code>fun</code> at the solution <code>x</code> . If <code>fun</code> does not compute the gradient, <code>grad</code> is a finite-differencing approximation of the gradient.	<code>fmincon</code> , <code>fminunc</code>

Argument	Description	Used by Functions
hessian	The value of the Hessian of fun at the solution x. For large-scale methods, if fun does not compute the Hessian, hessian is a finite-differencing approximation of the Hessian. For the quasi-newton, active-set, or sqp methods, hessian is the value of the Quasi-Newton approximation to the Hessian at the solution x. See “Hessian” on page 3-30.	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 3-27. lambda is a structure where each field is for a different constraint type. For structure field names, see individual function descriptions. (For lsqnonneg, lambda is simply a vector, as lsqnonneg only handles one kind of constraint.)	fgoalattain, fmincon, fminimax, fseminf, linprog, lsqcurvefit, lsqlin, lsqnonlin, lsqnonneg, quadprog
maxfval	$\max\{\text{fun}(x)\}$ at the solution x.	fminimax
output	An output structure that contains information about the results of the optimization, see “Output Structures” on page 3-26. For structure field names, see individual function descriptions.	All functions
residual	The value of the residual at the solution x.	lsqcurvefit, lsqlin, lsqnonlin, lsqnonneg

Argument	Description	Used by Functions
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 <code>exitflag</code> $> 0$ , 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 15-8
“Hidden Options” on page 15-23

### 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 15-23.



**Optimization Options**

<b>Option Name</b>	<b>Description</b>	<b>Used by Functions</b>	<b>Restrictions</b>
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:  $U - L \leq \text{AbsoluteGapTolerance}.$	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, lsqin, lsqnonlin, quadprog	

Option Name	Description	Used by Functions	Restrictions
BranchRule	<p>Rule for choosing the component for branching:</p> <ul style="list-style-type: none"> <li>• 'maxpscost' — The fractional component with maximum pseudocost. See “Branch and Bound” on page 9-40.</li> <li>• 'strongpscost' — The fractional component with maximum pseudocost, with a careful estimate of pseudocost. See “Branch and Bound” on page 9-40.</li> <li>• 'reliability' — The fractional component with maximum pseudocost, with an even more careful estimate of pseudocost than in 'strongpscost'. See “Branch and Bound” on page 9-40.</li> <li>• 'mostfractional' — The component whose fractional part is closest to 1/2.</li> <li>• 'maxfun' — The fractional component with maximal corresponding component in the absolute value of objective vector <math>f</math>.</li> </ul>	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	<p>Level of cut generation (see “Cut Generation” on page 9-35):</p> <ul style="list-style-type: none"> <li>• 'none' — No cuts. Makes CutMaxIterations irrelevant.</li> <li>• 'basic' — Normal cut generation.</li> <li>• 'intermediate' — Use more cut types.</li> <li>• 'advanced' — Use most cut types.</li> </ul>	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	<p>Level of display.</p> <ul style="list-style-type: none"> <li>• 'off' displays no output.</li> <li>• 'iter' displays output at each iteration, and gives the default exit message.</li> <li>• 'iter-detailed' displays output at each iteration, and gives the technical exit message.</li> <li>• 'notify' displays output only if the function does not converge, and gives the default exit message.</li> <li>• 'notify-detailed' displays output only if the function does not converge, and gives the technical exit message.</li> <li>• 'final' displays just the final output, and gives the default exit message.</li> <li>• 'final-detailed' displays just the final output, and gives the technical exit message.</li> </ul>	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	<p>Scalar or vector step size factor for finite differences. When you set <code>FiniteDifferenceStepSize</code> to a vector <math>v</math>, the forward finite differences <math>\delta</math> are</p> $\delta = v \cdot \text{sign}'(x) \cdot \max(\text{abs}(x), \text{TypicalX});$ <p>where <math>\text{sign}'(x) = \text{sign}(x)</math> except <math>\text{sign}'(0) = 1</math>. Central finite differences are</p> $\delta = v \cdot \max(\text{abs}(x), \text{TypicalX});$ <p>Scalar <code>FiniteDifferenceStepSize</code> expands to a vector. The default is <math>\sqrt{\text{eps}}</math> for forward finite differences, and <math>\text{eps}^{(1/3)}</math> for central finite differences.</p>	fgoalattain, fmincon, fminimax, fminunc, fseminf, fsolve, lsqcurvefit, lsqnonlin	optimoptions only. For optimset, use FinDiffRelStep
FiniteDifferenceType	<p>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 <code>fmincon</code> interior-point evaluations if the <code>HonorBounds</code> option is set to <code>false</code>.</p>	fgoalattain, fmincon, fminimax, fminunc, fseminf, fsolve, lsqcurvefit, lsqnonlin	optimoptions only. For optimset, use FinDiffType

Option Name	Description	Used by Functions	Restrictions
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 2-28).	fmincon, fminunc	optimoptions only. For optimset, use HessFcn
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

Option Name	Description	Used by Functions	Restrictions
Heuristics	<p>Algorithm for searching for feasible points (see “Heuristics for Finding Feasible Solutions” on page 9-36):</p> <ul style="list-style-type: none"> <li>• 'basic'</li> <li>• 'intermediate'</li> <li>• 'advanced'</li> <li>• 'rss'</li> <li>• 'rins'</li> <li>• 'round'</li> <li>• 'diving'</li> <li>• 'rss-diving'</li> <li>• 'rins-diving'</li> <li>• 'round-diving'</li> <li>• 'none'</li> </ul>	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 9-36.	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

Option Name	Description	Used by Functions	Restrictions
IntegerPreprocess	<p>Types of integer preprocessing (see “Mixed-Integer Program Preprocessing” on page 9-35):</p> <ul style="list-style-type: none"> <li>• 'none' — Use very few integer preprocessing steps.</li> <li>• 'basic' — Use a moderate number of integer preprocessing steps.</li> <li>• 'advanced' — Use all available integer preprocessing steps.</li> </ul>	intlinprog	optimoptions only
IntegerTolerance	<p>Real from 1e-6 through 1e-3, where the maximum deviation from integer that a component of the solution <math>x</math> can have and still be considered an integer. IntegerTolerance is not a stopping criterion.</p>	intlinprog	optimoptions only
JacobianMultiplyFcn	<p>User-defined Jacobian multiply function, specified as a function handle. Ignored unless SpecifyObjectiveGradient is true for fsolve, lsqcurvefit, and lsqnonlin.</p>	fsolve, lsqcurvefit, lsqlin, lsqnonlin	



Option Name	Description	Used by Functions	Restrictions
LinearSolver	Type of internal linear solver in algorithm: <ul style="list-style-type: none"> <li>'auto' — Use 'sparse' if the passed quadratic matrix is sparse (H for quadprog, C for lsqlin), 'dense' otherwise.</li> <li>'sparse' — Use sparse linear algebra.</li> <li>'dense' — Use dense linear algebra.</li> </ul>	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, fsemif, 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

Option Name	Description	Used by Functions	Restrictions
MaxFeasiblePoints	Strictly positive integer. <code>intlinprog</code> stops if it finds <code>MaxFeasiblePoints</code> integer feasible points.	<code>intlinprog</code>	optimoptions only
MaxNodes	Strictly positive integer that is the maximum number of nodes the solver explores in its branch-and-bound process.	<code>intlinprog</code>	
MaxTime	Maximum amount of time in seconds allowed for the algorithm.	<code>intlinprog</code> , <code>linprog</code>	
NodeSelection	Choose the node to explore next. <ul style="list-style-type: none"> <li>• 'simplebestproj' — Best projection. See “Branch and Bound” on page 9-40.</li> <li>• 'minobj' — Explore the node with the minimum objective function.</li> <li>• 'mininfeas' — Explore the node with the minimal sum of integer infeasibilities. See “Branch and Bound” on page 9-40.</li> </ul>	<code>intlinprog</code>	optimoptions only
ObjectiveCutoff	Real greater than <code>-Inf</code> . The default is <code>Inf</code> .	<code>intlinprog</code>	optimoptions only

Option Name	Description	Used by Functions	Restrictions
ObjectiveImprovementThreshold	Nonnegative real. <code>intlinprog</code> changes the current feasible solution only when it locates another with an objective function value that is at least <code>ObjectiveImprovementThreshold</code> lower: $(fold - fnew)/(1 +  fold ) > ObjectiveImprovementThreshold$ .	<code>intlinprog</code>	optimoptions only
ObjectiveLimit	If the objective function value goes below <code>ObjectiveLimit</code> and the iterate is feasible, then the iterations halt.	<code>fmincon</code> , <code>fminunc</code> , <code>quadprog</code>	
OptimalityTolerance	Termination tolerance on the first-order optimality.	<code>fgoalattain</code> , <code>fmincon</code> , <code>fminimax</code> , <code>fminsearch</code> , <code>fminunc</code> , <code>fseminf</code> , <code>fsolve</code> , <code>linprog</code> (interior-point only), <code>lsqcurvefit</code> , <code>lsqlin</code> , <code>lsqnonlin</code> , <code>quadprog</code>	optimoptions only. For <code>optimset</code> , use <code>TolFun</code>
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 Syntax” on page 15-37 or “intlinprog Output Function and Plot Function Syntax” on page 15-48.	<code>fgoalattain</code> , <code>fminbnd</code> , <code>fmincon</code> , <code>fminimax</code> , <code>fminsearch</code> , <code>fminunc</code> , <code>fseminf</code> , <code>fsolve</code> , <code>fzero</code> , <code>intlinprog</code> , <code>lsqcurvefit</code> , <code>lsqnonlin</code>	

Option Name	Description	Used by Functions	Restrictions
PlotFcn	<p>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.</p> <ul style="list-style-type: none"> <li>• 'optimplotx' plots the current point</li> <li>• 'optimplotfunccount' plots the function count</li> <li>• 'optimplotfval' plots the function value</li> <li>• 'optimplotconstrviolation' plots the maximum constraint violation</li> <li>• 'optimplotresnorm' plots the norm of the residuals</li> <li>• 'optimplotfirstorderopt' plots the first-order of optimality</li> <li>• 'optimplotstepsize' plots the step size</li> <li>• 'optimplotmilp' plots the gap for mixed-integer linear programs</li> </ul> <p>See “Plot Function Syntax” on page 15-47 or “intlinprog Output Function and Plot</p>	<p>fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, intlinprog, lsqcurvefit, lsqnonlin. See the individual function reference pages for the values that apply.</p>	<p>optimoptions only. For optimset, use PlotFcns</p>

Option Name	Description	Used by Functions	Restrictions
	Function Syntax” on page 15-48.		
RelativeGapTolerance	<p>Real from 0 through 1. <code>intlinprog</code> 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 <code>RelativeGapTolerance</code>:</p> $(U - L) / (\text{abs}(U) + 1) \leq \text{RelativeGapTolerance}.$ <p><code>intlinprog</code> automatically modifies the tolerance for large L magnitudes:</p> $\text{tolerance} = \min(1/(1+ L ), \text{RelativeGapTolerance})$	intlinprog	optimoptions only
RootLPAlgorithm	<p>Algorithm for solving linear programs:</p> <ul style="list-style-type: none"> <li>• 'dual-simplex' — Dual simplex algorithm</li> <li>• 'primal-simplex' — Primal simplex algorithm</li> </ul>	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

Option Name	Description	Used by Functions	Restrictions
ScaleProblem	For <code>fmincon</code> interior-point and <code>sqp</code> algorithms, <code>true</code> causes the algorithm to normalize all constraints and the objective function by their initial values. Disable by setting to the default <code>false</code> .	<code>fmincon</code>	
SpecifyConstraintGradient	User-defined gradients for the nonlinear constraints.	<code>fgoalattain</code> , <code>fmincon</code> , <code>fminimax</code>	optimoptions only. For <code>optimset</code> , use <code>GradConstraint</code>
SpecifyObjectiveGradient	User-defined gradients or Jacobians for the objective functions.	<code>fgoalattain</code> , <code>fmincon</code> , <code>fminimax</code> , <code>fminunc</code> , <code>fseminf</code> , <code>fsolve</code> , <code>lsqcurvefit</code> , <code>lsqnonlin</code>	optimoptions only. For <code>optimset</code> , use <code>GradObj</code> or <code>Jacobian</code>
StepTolerance	Termination tolerance on $x$ .	All functions except <code>linprog</code> and <code>lsqlin</code>	optimoptions only. For <code>optimset</code> , use <code>TolX</code>
SubproblemAlgorithm	Determines how the iteration step is calculated.	<code>fmincon</code>	
TypicalX	Array that specifies typical magnitude of array of parameters $x$ . The size of the array is equal to the size of <code>x0</code> , the starting point. Primarily for scaling finite differences for gradient estimation.	<code>fgoalattain</code> , <code>fmincon</code> , <code>fminimax</code> , <code>fminunc</code> , <code>fsolve</code> , <code>lsqcurvefit</code> , <code>lsqlin</code> , <code>lsqnonlin</code> , <code>quadprog</code>	

Option Name	Description	Used by Functions	Restrictions
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 2-82.

Function reference pages list these options in *italics*.

- “Hidden Optimization Toolbox Options” on page 15-23
- “Hidden Global Optimization Toolbox Options” on page 15-29

### Hidden Optimization Toolbox Options

This table lists the hidden Optimization Toolbox options.

**Options that `optimoptions` Hides**

<b>Option Name</b>	<b>Description</b>	<b>Used by Functions</b>	<b>Restrictions</b>
<i>Diagnostics</i>	Display diagnostic information about the function to be minimized or solved.	All but <code>fminbnd</code> , <code>fminsearch</code> , <code>fzero</code> , and <code>lsqnonneg</code>	
<i>DiffMaxChange</i>	Maximum change in variables for finite differencing.	<code>fgoalattain</code> , <code>fmincon</code> , <code>fminimax</code> , <code>fminunc</code> , <code>fseminf</code> , <code>fsolve</code> , <code>lsqcurvefit</code> , <code>lsqnonlin</code>	
<i>DiffMinChange</i>	Minimum change in variables for finite differencing.	<code>fgoalattain</code> , <code>fmincon</code> , <code>fminimax</code> , <code>fminunc</code> , <code>fseminf</code> , <code>fsolve</code> , <code>lsqcurvefit</code> , <code>lsqnonlin</code>	



Option Name	Description	Used by Functions	Restrictions
<i>FunValCheck</i>	<p>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.</p> <hr/> <p><b>Note</b> <i>FunValCheck</i> does not return an error for Inf when used with <i>fminbnd</i>, <i>fminsearch</i>, or <i>fzero</i>, which handle Inf appropriately.</p> <hr/> <p>'off' displays no error.</p>	fgoalattain, fminbnd, fmincon, fminimax, fminsearch, fminunc, fseminf, fsolve, fzero, lsqcurvefit, lsqnonlin	
<i>HessPattern</i>	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 $x_0$ , the starting point.	fmincon, fminunc	
<i>HessUpdate</i>	Quasi-Newton updating scheme.	fminunc	
<i>InitBarrierParam</i>	Initial barrier value.	fmincon	

<b>Option Name</b>	<b>Description</b>	<b>Used by Functions</b>	<b>Restrictions</b>
<i>InitDamping</i>	Initial Levenberg-Marquardt parameter.	fsolve, lsqcurvefit, lsqnonlin	optimoptions only
<i>InitTrustRegionRadiu s</i>	Initial radius of the trust region.	fmincon	
<i>JacobPattern</i>	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	
<i>LPPreprocess</i>	Type of preprocessing for the solution to the relaxed linear program (see “Linear Program Preprocessing” on page 9-34): <ul style="list-style-type: none"> <li>• 'none' — No preprocessing.</li> <li>• 'basic' — Use preprocessing.</li> </ul>	intlinprog	optimoptions only

<b>Option Name</b>	<b>Description</b>	<b>Used by Functions</b>	<b>Restrictions</b>
<i>MaxPCGIter</i>	Maximum number of iterations of preconditioned conjugate gradients method allowed.	fmincon, fminunc, fsolve, lsqcurvefit, lsqlin, lsqnonlin, quadprog	
<i>MaxProjCGIter</i>	A tolerance for the number of projected conjugate gradient iterations; this is an inner iteration, not the number of iterations of the algorithm.	fmincon	
<i>MaxSQPIter</i>	Maximum number of iterations of sequential quadratic programming method allowed.	fgoalattain, fmincon, fminimax	
<i>MeritFunction</i>	Use goal attainment/ minimax merit function (multiobjective) vs. fmincon (single objective).	fgoalattain, fminimax	
<i>PrecondBandWidth</i>	Upper bandwidth of preconditioner for PCG. Setting to 'Inf' uses a direct factorization instead of CG.	fmincon, fminunc, fsolve, lsqcurvefit, lsqlin, lsqnonlin, quadprog	
<i>Preprocess</i>	Level of LP preprocessing prior to simplex or dual simplex algorithm iterations.	linprog	optimoptions only

<b>Option Name</b>	<b>Description</b>	<b>Used by Functions</b>	<b>Restrictions</b>
<i>RelLineSrchBnd</i>	Relative bound on line search step length.	fgoalattain, fmincon, fminimax, fseminf	
<i>RelLineSrchBndDuration</i>	Number of iterations for which the bound specified in RelLineSrchBnd should be active.	fgoalattain, fmincon, fminimax, fseminf	
<i>ScaleProblem</i>	When using the Algorithm option 'levenberg-marquardt', setting the ScaleProblem option to 'jacobian' sometimes helps the solver on badly-scaled problems.	fsolve, lsqcurvefit, lsqnonlin	
<i>TolConSQP</i>	Constraint violation tolerance for the inner SQP iteration.	fgoalattain, fmincon, fminimax, fseminf	
<i>TolPCG</i>	Termination tolerance on the PCG iteration.	fmincon, fminunc, fsolve, lsqcurvefit, lsqlin, lsqnonlin, quadprog	
<i>TolProjCG</i>	A relative tolerance for projected conjugate gradient algorithm; this is for an inner iteration, not the algorithm iteration.	fmincon	

<b>Option Name</b>	<b>Description</b>	<b>Used by Functions</b>	<b>Restrictions</b>
<i>TolProjCGAbs</i>	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

<b>Option Name</b>	<b>Used by Functions</b>
<i>Cache</i>	patternsearch
<i>CacheSize</i>	patternsearch
<i>CacheTol</i>	patternsearch
<i>DisplayInterval</i>	particleswarm, simulannealbnd
<i>FunValCheck</i>	particleswarm
<i>HybridInterval</i>	simulannealbnd
<i>InitialPenalty</i>	ga, patternsearch
<i>MaxMeshSize</i>	patternsearch
<i>MeshRotate</i>	patternsearch
<i>MigrationDirection</i>	ga
<i>MigrationFraction</i>	ga
<i>MigrationInterval</i>	ga
<i>PenaltyFactor</i>	ga, patternsearch
<i>PlotInterval</i>	ga, patternsearch, simulannealbnd
<i>StallTest</i>	ga
<i>TolBind</i>	patternsearch

## **See Also**

### **More About**

- “Current and Legacy Option Name Tables” on page 15-31

## Current and Legacy Option Name Tables

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')
```

These two tables have identical information. One is in alphabetical order by legacy option name, the other is in order by current option name. The tables show values only when the values differ between legacy and current, and show only names that differ or have different values. For changes in Global Optimization Toolbox solvers, see “Options Changes in R2016a” (Global Optimization Toolbox).

**Option Names in Legacy Order**

<b>Legacy Name</b>	<b>Current Name</b>	<b>Legacy Values</b>	<b>Current Values</b>
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	IntegerPreprocesses		
Jacobian	SpecifyObjectiveGradient		



Legacy Name	Current Name	Legacy Values	Current Values
JacobMult	JacobianMultiplyFcn		
LPMaxIter	LPMaxIterations		
MaxFunEvals	MaxFunctionEvaluations		
MaxIter	MaxIterations		
MaxNumFeasPoints	MaxFeasiblePoints		
MinAbsMax	AbsoluteMaxObjectiveCount		
PlotFcns	PlotFcn		
RelObjThreshold	ObjectiveImprovementThreshold		
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**

<b>Current Name</b>	<b>Legacy Name</b>	<b>Current Values</b>	<b>Legacy Values</b>
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 is nonempty.	'user-supplied', 'bfgs', 'lbfgs', 'fin-diff-grads', 'on', 'off'
HessianFcn	HessFcn		
HessianMultiplyFcn	HessMult		
HonorBounds	AlwaysHonorConstraints	true, false	'bounds', 'none'
IntegerPreprocesses	IPPreprocess		

Current Name	Legacy Name	Current Values	Legacy Values
IntegerTolerance	TolInteger		
JacobianMultiplyFcn	JacobMult		
LPMaxIterations	LPMaxIter		
LPOptimalityTolerance	TolFunLP		
MaxFeasiblePoints	MaxNumFeasPoints		
MaxFunctionEvaluations	MaxFunEvals		
MaxIterations	MaxIter		
ObjectiveImprovementThreshold	RelObjThreshold		
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 15-8

# Output Function Syntax

## In this section...

“What Are Output Functions?” on page 15-37  
“Structure of the Output Function” on page 15-38  
“Fields in `optimValues`” on page 15-38  
“States of the Algorithm” on page 15-44  
“Stop Flag” on page 15-45

## What Are Output Functions?

The `OutputFcn` field of `options` specifies one or more functions that an optimization function calls at each iteration. Typically, you might use an output function to plot points at each iteration or to display optimization quantities from the algorithm. Using an output function you can view, but not set, optimization quantities.

---

**Caution** `intlinprog` output functions and plot functions differ from those in other solvers. See “`intlinprog` Output Function and Plot Function Syntax” on page 15-48.

---

To set up an output function, do the following:

- 1 Write the output function as a function file or local function.
- 2 Use `optimoptions` to set the value of `OutputFcn` to be a function handle, that is, the name of the function preceded by the `@` sign. For example, if the output function is `outfun.m`, the command

```
options = optimoptions(@solvername,'OutputFcn', @outfun);
```

specifies `OutputFcn` to be the handle to `outfun`. To specify more than one output function, use the syntax

```
options = optimoptions(@solvername,'OutputFcn',{@outfun, @outfun2});
```

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

See “Output Functions” on page 3-39 for an example of an output function.

“Passing Extra Parameters” on page 2-70 explains how to parametrize the output function `OutputFcn`, if necessary.

## Structure of the Output Function

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

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

where

- `x` is the point computed by the algorithm at the current iteration.
- `optimValues` is a structure containing data from the current iteration. “Fields in `optimValues`” on page 15-38 describes the structure in detail.
- `state` is the current state of the algorithm. “States of the Algorithm” on page 15-44 lists the possible values.
- `stop` is a flag that is `true` or `false` depending on whether the optimization routine should quit or continue. See “Stop Flag” on page 15-45 for more information.

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

## Fields in `optimValues`

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

### Corresponding Output Arguments

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

## Command-Line Display

The values of some fields of `optimValues` are displayed at the command line when you call the optimization function with the `Display` field of `options` set to `'iter'`, as described in “Iterative Display” on page 3-17. 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 3-17.
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 3-17.
degenerate	Measure of degeneracy. A point is <i>degenerate</i> if  The partial derivative with respect to one of the variables is 0 at the point.  A bound constraint is active for that variable at the point.  See “Degeneracy” on page 15-44.	fmincon (TRR), lsqcurvefit (TRR), lsqnonlin (TRR)	None



OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
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 3-17.
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 3-17.
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 3-17.
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 3-17.

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
gradient	Current gradient of objective function — either analytic gradient if you provide it or finite-differencing approximation. Final value equals optimization function output <code>grad</code> .	<code>fgoalattain</code> , <code>fmincon</code> , <code>fminimax</code> , <code>fminunc</code> , <code>fseminf</code> , <code>fsolve</code> , <code>lsqcurvefit</code> , <code>lsqnonlin</code>	None
iteration	Iteration number — starts at 0. Final value equals optimization function output <code>output.iterations</code> .	<code>fgoalattain</code> , <code>fminbnd</code> , <code>fmincon</code> , <code>fminimax</code> , <code>fminsearch</code> , <code>fminunc</code> , <code>fsolve</code> , <code>fseminf</code> , <code>fzero</code> , <code>lsqcurvefit</code> , <code>lsqnonlin</code>	Iteration  See “Iterative Display” on page 3-17.
lambda	The Levenberg-Marquardt parameter, <code>lambda</code> , at the current iteration. See “Levenberg-Marquardt Method” on page 12-7.	<code>fsolve</code> (LM), <code>lsqcurvefit</code> (LM), <code>lsqnonlin</code> (LM)	Lambda
lssteplength	Actual step length divided by initially predicted step length	<code>fmincon</code> (AS, SQP), <code>fminunc</code> (Q)	Steplength or Line search steplength or Step-size  See “Iterative Display” on page 3-17.
maxfval	Maximum function value	<code>fminimax</code>	None
positivedefinite	0 if algorithm detects negative curvature while computing Newton step.  1 otherwise.	<code>fmincon</code> (TRR), <code>fminunc</code> (TR), <code>fsolve</code> (TRR), <code>lsqcurvefit</code> (TRR), <code>lsqnonlin</code> (TRR)	None

<b>OptimValues Field (optimValues.field)</b>	<b>Description</b>	<b>Returned by Functions</b>	<b>Command-Line Display</b>
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 3-17.
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 3-17.
resnorm	2-norm of the residual squared.	lsqcurvefit, lsqnonlin	Resnorm  See “Iterative Display” on page 3-17.
searchdirection	Search direction.	fgoalattain, fmincon (AS, SQP), fminimax, fminunc (Q), fseminf, fsolve (LM), lsqcurvefit (LM), lsqnonlin (LM)	None
stepaccept	Status of the current trust-region step. Returns true if the current trust- region step was successful, and false if the trust-region step was unsuccessful.	fsolve (D)	None

OptimValues Field (optimValues.field)	Description	Returned by Functions	Command-Line Display
stepsize	Current step size (displacement in $x$ ). Final value equals optimization function output <code>output.stepsize</code> .	fgoalattain, fmincon, fminimax, fminunc, fseminf, fsolve, lsqcurvefit, lsqnonlin	Step-size or Norm of Step  See “Iterative Display” on page 3-17.
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 3-17.

### 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,

$$r = \min(\text{abs}(\text{ub}-x), x-\text{lb})$$

Then the value of `degenerate` is the minimum entry of the vector  $r + \text{abs}(\text{grad})$ , 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:

- $\text{grad}(i) = 0$
- $x(i)$  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.

State	Description
'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 <code>x</code> and <code>optimValues</code> are the same as at the last call to the output function in which <code>state=='iter'</code> .
'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:

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

## Stop Flag

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

### Stopping an Optimization Based on Data in `optimValues`

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

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

### **Stopping an Optimization Based on GUI Input**

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

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

## Plot Function Syntax

The `PlotFcn` option specifies one or more functions that an optimization function calls at each iteration to plot various measures of progress while the algorithm executes. The structure of a plot function is the same as that for an output function. For more information on writing and calling a plot function, see “Output Function Syntax” on page 15-37. For an example of using built-in plot functions, “Using a Plot Function” on page 3-33.

To view a predefined plot function listed for `PlotFcn`, open the function in the MATLAB Editor. For example, to view the file corresponding to the norm of residuals, enter:

```
edit optimplotresnorm.m
```

You can use any predefined plot function as a template for writing a custom plot function.

## See Also

### More About

- “Output Function Syntax” on page 15-37
- “Using a Plot Function” on page 3-33

## intlinprog Output Function and Plot Function Syntax

### In this section...

“What Are Output Functions and Plot Functions?” on page 15-48

“Custom Function Syntax” on page 15-49

“optimValues Structure” on page 15-50

### 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 Syntax” on page 15-37 and “Plot Function Syntax” on page 15-47.

- 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 9-52.

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 15-50.
- `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: <ul style="list-style-type: none"> <li>'rootlp' — <code>intlinprog</code> solved the root LP problem.</li> <li>'cutgen' — <code>intlinprog</code> added cuts and improved the lower bound.</li> <li>'heuristics' — <code>intlinprog</code> found new feasible points using heuristics.</li> <li>'branching' — <code>intlinprog</code> is creating and exploring nodes in a branch-and-bound tree.</li> </ul>
fval	Best objective function found so far at an integer feasible point. When phase = 'rootlp', fval is the objective function value at the root node, which is not necessarily an integer feasible point.
lowerbound	Global lower bound of the objective function value. Empty when phase = 'rootlp'.
relativegap	Relative gap between lowerbound and fval. Empty when phase = 'rootlp' or numfeaspoints = 0.
numnodes	Number of explored nodes. Nonzero only when phase = 'branching'.
numfeaspoints	Number of integer feasible solutions found.
time	Time in seconds spent so far, measured with <code>tic</code> and <code>toc</code> from the time when state = 'init'.

# Functions — Alphabetical List

---

## evaluate

**Package:** optim.problemdef

Evaluate optimization expression

## Syntax

```
val = evaluate(expr,pt)
```

## Description

`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)
```

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.

## See Also

`OptimizationExpression` | `infeasibility` | `solve`

## Topics

“Problem-Based Optimization Setup”

“Problem-Based 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 having N outputs.

`[out1,out2,...,outN] = fcn2optimexpr(fcn,in1,in2,...,inK,Name,Value)` modifies the expression creation process according to name-value parameters.

## Examples

### Convert Objective Function to Expression

To use a MATLAB™ function in the problem-based approach, first convert it to an optimization expression.

For example, to use the objective function  $-\exp(-x^2/2)$ , create an optimization variable `x` and use it in a converted anonymous function:

```
x = optimvar('x');
obj = fcn2optimexpr(@(t)-exp(-t^2/2),x);
prob = optimproblem('Objective',obj);
showproblem(prob)
```

```
OptimizationProblem :
```

```
minimize :
    anonymousFunction1(x)

where:

    anonymousFunction1 = @(t)-exp(-t^2/2);
```

For more complex functions, convert a function file. For example, suppose that you have a function file named `expfn2.m` that computes an objective in two optimization variables.

type `expfn2`

```
function f = expfn2(t,u)
f = -exp(-t^2/2)*u/(1 + u^2);
```

Include this objective in a problem.

```
x = optimvar('x');
y = optimvar('y','LowerBound',0);
obj = fcn2optimexpr(@expfn2,x,y);
prob = optimproblem('Objective',obj);
showproblem(prob)
```

```
OptimizationProblem :

minimize :
    expfn2(x, y)

variable bounds:
    0 <= y
```

If your function has several outputs, you can use them as elements of the objective function. For example, suppose that `u` is a 2-by-2 variable and `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);
showproblem(prob)
```

```
OptimizationProblem :

minimize :
    ((arg2 .* arg3) ./ (1 + arg1.^2))

where:

    [arg1,~,~] = expfn3(u, v);
    [arg2,~,~] = expfn3(u, v);
    [~,arg3,~] = 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.

Create the constraint that `expfn2` is less than or equal to  $-1/2$ . This function of two variables is in the `expfn2.m` file.

type `expfn2`

```
function f = expfn2(t,u)
f = -exp(-t^2/2)*u/(1 + u^2);
```

Create optimization variables, convert the function file to an optimization expression, then express the constraint as `confn`.

```
x = optimvar('x');
y = optimvar('y', 'LowerBound', 0);
```

```
expr1 = fcn2optimexpr(@expfn2,x,y);
confn = expr1 <= -1/2;
showconstr(confn)

    expfn2(x, y) <= (-0.5)
```

Click the blue link [extraParams](#) above to see the following output:

```
extraParams{1}:
```

```
-0.5000
```

Create another constraint that `expfn2` is greater than `x + y`.

```
confn2 = expr1 >= x + y;
```

Create an optimization problem and place the constraints in the problem.

```
prob = optimproblem;
prob.Constraints.conf1 = confn;
prob.Constraints.conf2 = confn2;
showproblem(prob)
```

```
OptimizationProblem :

    minimize :
    subject to conf1:
        expfn2(x, y) <= (-0.5)

    subject to conf2:
        expfn2(x, y) >= (x + y)

    variable bounds:
        0 <= y
```

### Compute Common Objective and Constraint Efficiently

If the objective and nonlinear constraint come from a common, time-consuming function, save time by using the 'ReuseEvaluation' name-value pair. For example, `rosenbrocknorm` computes both the Rosenbrock objective function, and the norm of the argument for use in the constraint  $\|x\|^2 \leq 4$ .

```
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`, specifying `'ReuseEvaluation'`.

```
x = optimvar('x',2);
[f,c] = fcn2optimexpr(@rosenbrocknorm,x,'ReuseEvaluation',true);
```

Create objective and constraint expressions from the returned expressions. Include these expressions in an optimization problem. Review the problem using `showproblem`.

```
prob = optimproblem('Objective',f);
prob.Constraints.cineq = c <= 4;
showproblem(prob)
```

```
OptimizationProblem :

    minimize :
    [argout,~] = generatedFunction_rosenbrocknorm_withReuse(x)

    subject to cineq:
    arg_LHS <= 4

    where:

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

Solve the problem starting from the initial point  $x_0$ .  $x = [-1;1]$ , timing the result.

```
x0.x = [-1;1];
tic
[sol,fval,exitflag,output] = solve(prob,x0)
```

```
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: [2x1 double]

fval = 3.6222e-11

exitflag =
  OptimalSolution

output = struct with fields:
  iterations: 43
  funcCount: 161
  constrviolation: 0
  stepsize: 9.1067e-08
  algorithm: 'interior-point'
  firstorderopt: 6.3912e-07
  cgiterations: 10
  message: '↵Local minimum found that satisfies the constraints.↵Optimization terminated because the algorithm reached its maximum number of iterations.'
  solver: 'fmincon'

toc

Elapsed time is 161.910171 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 2-65.

## 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.

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],'Reuse',true)` specifies that `out1` and `out2` are scalars and that these variables will be reused between objective and constraint functions without recalculation.

**OutputSize — Sizes of output expressions**

integer vector | cell array of integer vectors

Sizes of output expressions, specified as:

- An integer vector — If there is one output `out1`, `OutputSize` specifies the size of `out1`. If there are multiple outputs `out1, ..., outN`, `OutputSize` specifies that all outputs have the same size.

---

**Note** A scalar has size `[1,1]`.

---

- A cell array of integer vectors — The size of output `outj` is the `j`th element of `OutputSize`.

If you do not pass an `OutputSize` name-value pair, then `fcn2optimexpr` passes data to `fcn` in order to determine the sizes of the outputs (see “Algorithms” on page 16-12). By passing `OutputSize`, you enable `fcn2optimexpr` to skip this step. Skipping this evaluation saves time. Additionally, if you do not pass an `OutputSize` name-value pair, and if 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 — Enable reusable values**

`false` (default) | `true`

Enable reusable values, specified as `false` (do not enable) or `true` (enable).

`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, so avoids recalculating the value.

There is a small overhead in enabling reusable values, 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.

## **Algorithms**

To find the output size of each returned expression when you do not provide an `OutputSize`, `fcn2optimexpr` evaluates your function at the following point for each element of the problem variables.

- If there is a finite upper bound `ub` and a finite lower bound `lb`, the evaluation point is  $(lb + ub)/2 + ((ub - lb)/2)*eps$ .
- If there is a finite lower bound and no upper bound, the evaluation point is  $lb + \max(1, abs(lb))*eps$ .
- If there is a finite upper bound and no lower bound, the evaluation point is  $ub - \max(1, abs(ub))*eps$ .
- If there are no bounds, the evaluation point is  $1 + eps$ .
- In addition, if the variable is specified as an integer, the evaluation point is `floor` of the point given previously.

It is possible that this evaluation point leads to an error in function evaluation. To avoid this error, specify `OutputSize`.

## See Also

### Topics

“Problem-Based Workflow” on page 10-2

“Optimization Expressions” on page 10-4

**Introduced in R2019a**

## fgoalattain

Solve multiobjective goal attainment problems

`fgoalattain` solves the goal attainment problem, a formulation for minimizing a multiobjective optimization problem.

`fgoalattain` finds the minimum of a problem specified by

$$\text{minimize } \gamma \text{ such that } \left\{ \begin{array}{l} F(x) - \text{weight} \cdot \gamma \leq \text{goal} \\ c(x) \leq 0 \\ \text{ceq}(x) = 0 \\ A \cdot x \leq b \\ \text{Aeq} \cdot x = \text{beq} \\ lb \leq x \leq ub. \end{array} \right.$$

`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 2-40.

## 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

`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 2-70 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  $A*x \leq b$ .

`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  $lb \leq x \leq ub$ . 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 2-42.

---

---

**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  $c(x) \leq 0$  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`, where `problem` is a structure described in `problem`. Create the `problem` structure by

exporting a problem from the Optimization app, as described in “Exporting Your Work” on page 5-11.

`[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  $x_0 = 1$ .

```
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  $A*x \leq b$ .

```
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  $0 \leq x_1 \leq 3$ ,  $2 \leq x_2 \leq 5$ .

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  $\|x\|^2 \leq 4$ .

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];
```

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  $x_1 + x_2 \leq 4$ .

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  $A*x \leq b$ .

```
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)
```

Iter	F-count	Attainment factor	Max constraint	Line search steplength	Directional derivative	Procedure
0	4	0	4			
1	9	-1	2.5	1	-0.535	
2	14	-1.115e-08	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	6.748e-13	1	0.883	Hessian modi

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 = 1x2
    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  $x_1 + x_2 \leq 4$ .

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  $A*x \leq b$ .

```
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  $x_1 + x_2 \leq 4$ .

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  $A*x \leq b$ .

```
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: 4.1454e-13
    algorithm: 'active-set'
    firstorderopt: []
    constrviolation: 6.7482e-13
    message: '...'

lambda = struct with fields:
    lower: [2×1 double]
    upper: [2×1 double]
    eqlin: [0×1 double]
    eqnonlin: [0×1 double]
    ineqlin: 0.5394
    ineqnonlin: [0×1 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  $x_1 + x_2 \leq 4$ .

```
A = [1 1];  
b = 4;
```

Solve the goal attainment problem starting from the point  $x_0 = [1 \ 1]$ .

```
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 = 1x2
```

```
    2.0694    1.9306
```

```
fval = 2x1
```

```
    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” (MATLAB)).

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 `j`th column of `G` is the gradient of the `j`th 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  $\gamma$  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  $\gamma$  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  $\gamma$  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  $\gamma$  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 16-25.

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 8-23.

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

$$A*x \leq b,$$

where **x** is the column vector of N variables **x(:)**, and **b** is a column vector with M elements.

For example, to specify

$$\begin{array}{rclcl} x_1 & + & 2x_2 & \leq & 10 \\ 3x_1 & + & 4x_2 & \leq & 20 \\ 5x_1 & + & 6x_2 & \leq & 30, \end{array}$$

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

$$A*x \leq b,$$

where **x** is the column vector of N variables **x(:)**, and **A** is a matrix of size M-by-N.

For example, to specify

$$x_1 + 2x_2 \leq 10$$

$$\begin{array}{rclcl} 3x_1 & + & 4x_2 & \leq & 20 \\ 5x_1 & + & 6x_2 & \leq & 30, \end{array}$$

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

### **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

$$\text{Aeq} * x = \text{beq},$$

where `x` is the column vector of `N` variables `x(:)`, and `beq` is a column vector with `Me` elements.

For example, to specify

$$\begin{array}{rclcl} x_1 & + & 2x_2 & + & 3x_3 & = & 10 \\ 2x_1 & + & 4x_2 & + & x_3 & = & 20, \end{array}$$

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

$$\text{Aeq} * x = \text{beq},$$

where `x` is the column vector of `N` variables `x(:)`, and `Aeq` is a matrix of size `Me-by-N`.

For example, to specify

$$\begin{array}{rcccccc} x_1 & + & 2x_2 & + & 3x_3 & = & 10 \\ 2x_1 & + & 4x_2 & + & x_3 & = & 20, \end{array}$$

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

### **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

$$x(i) \geq lb(i) \quad \text{for all } i.$$

If `numel(lb) < numel(x0)`, then `lb` specifies that

$$x(i) \geq lb(i) \quad \text{for } 1 \leq i \leq \text{numel}(lb).$$

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  $x_0$  is equal to the number of elements in  $ub$ , then  $ub$  specifies that

$$x(i) \leq ub(i) \quad \text{for all } i.$$

If  $\text{numel}(ub) < \text{numel}(x_0)$ , then  $ub$  specifies that

$$x(i) \leq ub(i) \quad \text{for } 1 \leq i \leq \text{numel}(ub).$$

If there are fewer elements in  $ub$  than in  $x_0$ , solvers issue a warning.

Example: To specify that all  $x$  components are less than 1, use  $ub = \text{ones}(\text{size}(x_0))$ .

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  $\text{ceq}(x)$ .

- $c(x)$  is the array of nonlinear inequality constraints at  $x$ . `fgoalattain` attempts to satisfy  $c(x) \leq 0$  for all entries of  $c$ .
- $\text{ceq}(x)$  is the array of nonlinear equality constraints at  $x$ . `fgoalattain` attempts to satisfy  $\text{ceq}(x) = 0$  for all entries of  $\text{ceq}$ .

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  $\text{ceq}(x)$ . See “Nonlinear

Constraints” on page 2-48 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 2-70 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 2-82.

For details about options that have different names for `optimset`, see “Current and Legacy Option Name Tables” on page 15-31.

<b>Option</b>	<b>Description</b>
ConstraintTolerance	Termination tolerance on the constraint violation, a positive scalar. The default is 1e-6. See "Tolerances and Stopping Criteria" on page 2-84.  For <code>optimset</code> , the name is <code>TolCon</code> .
<i>Diagnostics</i>	Display of diagnostic information about the function to be minimized or solved. The choices are 'on' or 'off' (the default).
<i>DiffMaxChange</i>	Maximum change in variables for finite-difference gradients (a positive scalar). The default is Inf.
<i>DiffMinChange</i>	Minimum change in variables for finite-difference gradients (a positive scalar). The default is 0.

Option	Description
Display	<p>Level of display (see “Iterative Display” on page 3-17):</p> <ul style="list-style-type: none"> <li>• 'off' or 'none' displays no output.</li> <li>• 'iter' displays output at each iteration, and gives the default exit message.</li> <li>• 'iter-detailed' displays output at each iteration, and gives the technical exit message.</li> <li>• 'notify' displays output only if the function does not converge, and gives the default exit message.</li> <li>• 'notify-detailed' displays output only if the function does not converge, and gives the technical exit message.</li> <li>• 'final' (default) displays only the final output, and gives the default exit message.</li> <li>• 'final-detailed' displays only the final output, and gives the technical exit message.</li> </ul>
EqualityGoalCount	<p>Number of objectives required for the objective <code>fun</code> to equal the goal <code>goal</code> (a nonnegative integer). The objectives must be partitioned into the first few elements of <code>F</code>. The default is 0. For an example, see “Multi-Objective Goal Attainment Optimization” on page 8-23.</p> <p>For <code>optimset</code>, the name is <code>GoalsExactAchieve</code>.</p>



Option	Description
FiniteDifferenceStepSize	<p>Scalar or vector step size factor for finite differences. When you set FiniteDifferenceStepSize to a vector <math>v</math>, the forward finite differences <math>\delta</math> are</p> $\delta = v \cdot \text{sign}'(x) \cdot \max(\text{abs}(x), \text{TypicalX});$ <p>where <math>\text{sign}'(x) = \text{sign}(x)</math> except <math>\text{sign}'(0) = 1</math>. Central finite differences are</p> $\delta = v \cdot \max(\text{abs}(x), \text{TypicalX});$ <p>Scalar FiniteDifferenceStepSize expands to a vector. The default is <math>\sqrt{\text{eps}}</math> for forward finite differences, and <math>\text{eps}^{1/3}</math> for central finite differences.</p> <p>For <code>optimset</code>, the name is <code>FinDiffRelStep</code>.</p>
FiniteDifferenceType	<p>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.</p> <p>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.</p> <p>For <code>optimset</code>, the name is <code>FinDiffType</code>.</p>

Option	Description
FunctionTolerance	<p>Termination tolerance on the function value (a positive scalar). The default is 1e-6. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolFun</code>.</p>
<i>FunValCheck</i>	<p>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.</p>
MaxFunctionEvaluations	<p>Maximum number of function evaluations allowed (a positive integer). The default is 100*numberOfVariables. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxFunEvals</code>.</p>
MaxIterations	<p>Maximum number of iterations allowed (a positive integer). The default is 400. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxIter</code>.</p>
<i>MaxSQPIter</i>	<p>Maximum number of SQP iterations allowed (a positive integer). The default is 10*max(numberOfVariables, numberOfInequalities + numberOfBounds).</p>
<i>MeritFunction</i>	<p>If this option is set to 'multiobj' (the default), use goal attainment merit function. If this option is set to 'singleobj', use the <code>fmincon</code> merit function.</p>

Option	Description
OptimalityTolerance	<p>Termination tolerance on the first-order optimality (a positive scalar). The default is <math>1e-6</math>. See “First-Order Optimality Measure” on page 3-12.</p> <p>For <code>optimset</code>, the name is <code>TolFun</code>.</p>
OutputFcn	<p>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 (<code>[]</code>). See “Output Function Syntax” on page 15-37.</p>
PlotFcn	<p>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 (<code>[]</code>).</p> <ul style="list-style-type: none"> <li>• 'optimplotx' plots the current point.</li> <li>• 'optimplotfunccount' plots the function count.</li> <li>• 'optimplotfval' plots the objective function values.</li> <li>• 'optimplotconstrviolation' plots the maximum constraint violation.</li> <li>• 'optimplotstepsize' plots the step size.</li> </ul> <p>For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.</p> <p>For <code>optimset</code>, the name is <code>PlotFcns</code>.</p>

Option	Description
<i>RelLineSrchBnd</i>	Relative bound (a real nonnegative scalar value) on the line search step length such that the total displacement in $x$ satisfies $ \Delta x(i)  \leq \text{relLineSrchBnd} \cdot \max( x(i) ,  typicalx(i) )$ . This option provides control over the magnitude of the displacements in $x$ when the solver takes steps that are too large. The default is none ( []).
<i>RelLineSrchBndDuration</i>	Number of iterations for which the bound specified in <i>RelLineSrchBnd</i> should be active. The default is 1.
SpecifyConstraintGradient	<p>Gradient for nonlinear constraint functions defined by the user. When this option is set to <code>true</code>, <code>fgoalattain</code> expects the constraint function to have four outputs, as described in <code>nonlcon</code>. When this option is set to <code>false</code> (the default), <code>fgoalattain</code> estimates gradients of the nonlinear constraints using finite differences.</p> <p>For <code>optimset</code>, the name is <code>GradConstr</code> and the values are 'on' or 'off'.</p>
SpecifyObjectiveGradient	<p>Gradient for the objective function defined by the user. Refer to the description of <code>fun</code> to see how to define the gradient. Set this option to <code>true</code> to have <code>fgoalattain</code> use a user-defined gradient of the objective function. The default, <code>false</code>, causes <code>fgoalattain</code> to estimate gradients using finite differences.</p> <p>For <code>optimset</code>, the name is <code>GradObj</code> and the values are 'on' or 'off'.</p>

Option	Description
StepTolerance	Termination tolerance on $x$ (a positive scalar). The default is $1e-6$ . See “Tolerances and Stopping Criteria” on page 2-84.  For <code>optimset</code> , the name is <code>TolX</code> .
<i>TolConSQP</i>	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 <code>TypicalX</code> is equal to the number of elements in <code>x0</code> , the starting point. The default value is <code>ones(numberofvariables,1)</code> . The <code>fgoalattain</code> function uses <code>TypicalX</code> for scaling finite differences for gradient estimation.
UseParallel	Indication of parallel computing. When <code>true</code> , <code>fgoalattain</code> estimates gradients in parallel. The default is <code>false</code> . 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
<code>objective</code>	Objective function <code>fun</code>
<code>x0</code>	Initial point for $x$
<code>goal</code>	Goals to attain
<code>weight</code>	Relative importance factors of goals
<code>Aineq</code>	Matrix for linear inequality constraints

Field Name	Entry
bineq	Vector for linear inequality constraints
Aeq	Matrix for linear equality constraints
beq	Vector for linear equality constraints
lb	Vector of lower bounds
ub	Vector of upper bounds
nonlcon	Nonlinear constraint function
solver	'fgoalattain'
options	Options created with <code>optimoptions</code>

You must supply at least the `objective`, `x0`, `goal`, `weight`, `solver`, and `options` fields in the problem structure.

The simplest way to obtain a problem structure is to export the problem from the Optimization app.

Data Types: `struct`

## Output Arguments

### **x** — Solution

real vector | real array

Solution, returned as a real vector or real array. The size of `x` is the same as the size of `x0`. Typically, `x` is a local solution to the problem when `exitflag` is positive. For information on the quality of the solution, see “When the Solver Succeeds” on page 4-22.

### **fval** — Objective function 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  $\gamma$  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 <code>options.ConstraintTolerance</code>
5	Magnitude of the directional derivative was less than the specified tolerance, and the constraint violation was less than <code>options.ConstraintTolerance</code>
0	Number of iterations exceeded <code>options.MaxIterations</code> or the number of function evaluations exceeded <code>options.MaxFunctionEvaluations</code>
-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.

<code>iterations</code>	Number of iterations taken
<code>funcCount</code>	Number of function evaluations
<code>lssteplength</code>	Size of the line search step relative to the search direction
<code>constrviolation</code>	Maximum of the constraint functions
<code>stepsize</code>	Length of the last displacement in $x$
<code>algorithm</code>	Optimization algorithm used
<code>firstorderopt</code>	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.

## **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 14-6.



## See Also

fmincon | fminimax | optimoptions | optimtool

## Topics

“Multi-Objective Goal Attainment Optimization” on page 8-23

“Generate and Plot a Pareto Front” on page 8-19

“Create Function Handle” (MATLAB)

“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"])
```

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"]);
```

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);
```

```
LP:           Optimal objective value is -1027.472366.
```

```
Heuristics:   Found 1 solution using rounding.
              Upper bound is -1027.233133.
              Relative gap is 0.00%.
```

```
Cut Generation: Applied 1 mir cut, and 2 strong CG cuts.
                Lower 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-onion"];
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

$$\text{var}(\text{numindex}(j)) = \text{var}(\text{strindex1}(j), \dots, \text{strindexk}(j)) \text{ for all } j.$$

## See Also

`OptimizationVariable` | `solve`

## Topics

“Create Initial Point for Optimization with Named Index Variables” on page 10-49

“Named Index for Optimization Variables” on page 10-22

**Introduced in R2018a**

## fminbnd

Find minimum of single-variable function on fixed interval

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.

## 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

`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  $x_1 < x < x_2$ .

`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`, where `problem` is a structure.

Create `problem` by exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11.

`[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  $1 \leq x \leq 3$ .

```
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” (MATLAB).

### 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

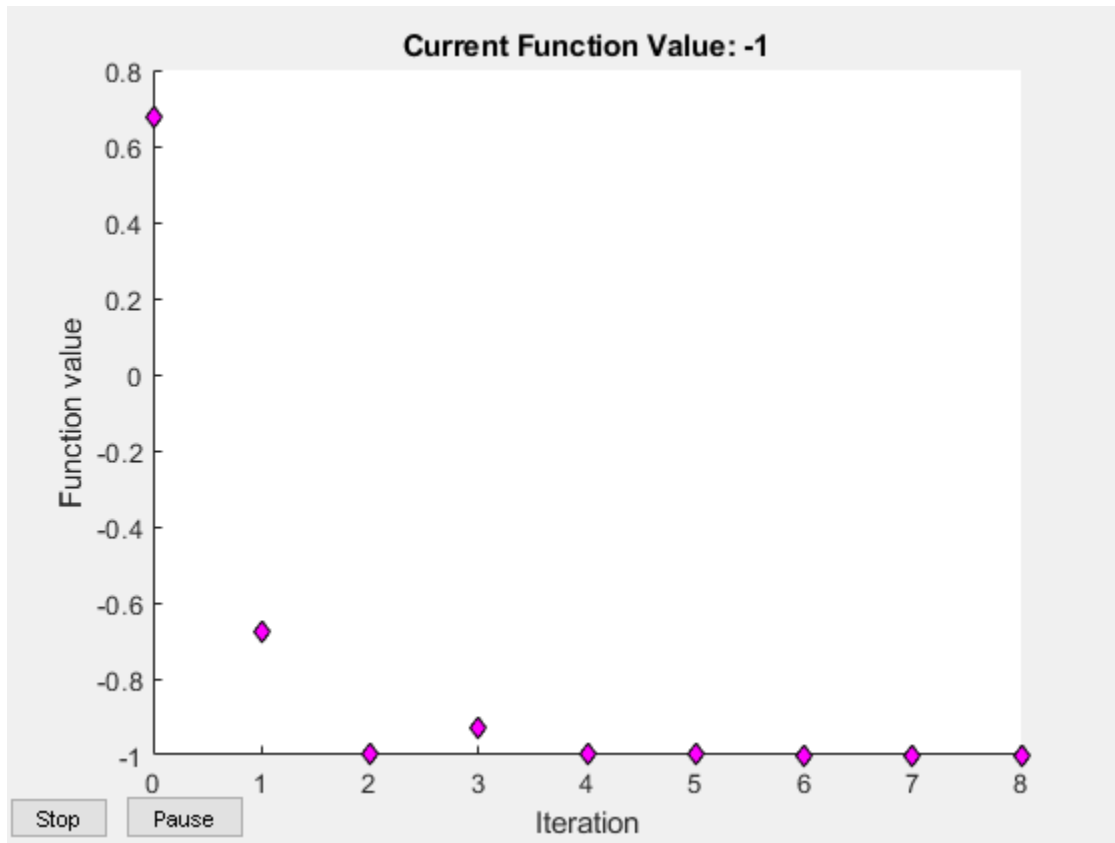
fval = -1.0000

### 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 15-8 for detailed information.

<b>Display</b>	Level of display (see “Iterative Display” on page 3-17): <ul style="list-style-type: none"><li>• 'notify' (default) displays output only if the function does not converge.</li><li>• 'off' or 'none' displays no output.</li><li>• 'iter' displays output at each iteration.</li><li>• 'final' displays just the final output.</li></ul>
<b>FunValCheck</b>	Check whether objective function values are valid. The default 'off' allows <code>fminbnd</code> to proceed when the objective function returns a value that is <code>complex</code> or <code>NaN</code> . The 'on' setting throws an error when the objective function returns a value that is <code>complex</code> or <code>NaN</code> .
<b>MaxFunEvals</b>	Maximum number of function evaluations allowed, a positive integer. The default is 500. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.
<b>MaxIter</b>	Maximum number of iterations allowed, a positive integer. The default is 500. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.
<b>OutputFcn</b>	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 ( <code>[]</code> ). See “Output Function Syntax” on page 15-37.

**PlotFcns**

Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none ( []).

- @optimplotx plots the current point
- @optimplotfunccount plots the function count
- @optimplotfval plots the function value

For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.

**TolX**

Termination tolerance on  $x$ , a positive scalar. The default is  $1e-4$ . See “Tolerances and Stopping Criteria” on page 2-84.

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 <code>optimset</code>

The simplest way to obtain a problem structure is to export the problem from the Optimization app.

Data Types: struct

## Output Arguments

### **x — Solution**

real 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 4-22.

### **fval — Objective function value at solution**

real number

Objective function value at the solution, returned as a real number. Generally,  $fval = fun(x)$ .

### **exitflag — Reason `fminbnd` stopped**

integer

Reason `fminbnd` stopped, returned as an integer.

1	Function converged to a solution $x$ .
0	Number of iterations exceeded <code>options.MaxIter</code> or number of function evaluations exceeded <code>options.MaxFunEvals</code> .
-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:

<code>iterations</code>	Number of iterations taken
<code>funcCount</code>	Number of function evaluations
<code>algorithm</code>	'golden section search, parabolic interpolation'
<code>message</code>	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  $2*(\text{ToLX} + 3*\text{abs}(x)*\text{sqrt}(\text{eps}))$ . See [1] or [2] for details about the algorithm.

## 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. *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

`fmincon` | `fminsearch` | `optimset` | `optimtool`

## Topics

“Create Function Handle” (MATLAB)

“Anonymous Functions” (MATLAB)

**Introduced before R2006a**

## fmincon

Find minimum of constrained nonlinear multivariable function

Nonlinear programming solver.

Finds the minimum of a problem specified by

$$\min_x f(x) \text{ such that } \begin{cases} c(x) \leq 0 \\ ceq(x) = 0 \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub, \end{cases}$$

$b$  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 2-40.

## Syntax

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

## Description

$x = \text{fmincon}(\text{fun}, x_0, A, b)$  starts at  $x_0$  and attempts to find a minimizer  $x$  of the function described in `fun` subject to the linear inequalities  $A*x \leq b$ .  $x_0$  can be a scalar, vector, or matrix.

---

**Note** “Passing Extra Parameters” on page 2-70 explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

---

$x = \text{fmincon}(\text{fun}, x_0, A, b, \text{Aeq}, \text{beq})$  minimizes `fun` subject to the linear equalities  $\text{Aeq}*x = \text{beq}$  and  $A*x \leq b$ . If no inequalities exist, set  $A = []$  and  $b = []$ .

$x = \text{fmincon}(\text{fun}, x_0, A, b, \text{Aeq}, \text{beq}, \text{lb}, \text{ub})$  defines a set of lower and upper bounds on the design variables in  $x$ , so that the solution is always in the range  $\text{lb} \leq x \leq \text{ub}$ . If no equalities exist, set  $\text{Aeq} = []$  and  $\text{beq} = []$ . If  $x(i)$  is unbounded below, set  $\text{lb}(i) = -\text{Inf}$ , and if  $x(i)$  is unbounded above, set  $\text{ub}(i) = \text{Inf}$ .

---

**Note** If the specified input bounds for a problem are inconsistent, `fmincon` throws an error. In this case, output  $x$  is  $x_0$  and `fval` is `[]`.

---

For the default 'interior-point' algorithm, `fmincon` sets components of  $x_0$  that violate the bounds  $\text{lb} \leq x \leq \text{ub}$ , 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 2-42.

---

$x = \text{fmincon}(\text{fun}, x_0, A, b, \text{Aeq}, \text{beq}, \text{lb}, \text{ub}, \text{nonlcon})$  subjects the minimization to the nonlinear inequalities  $c(x)$  or equalities  $\text{ceq}(x)$  defined in `nonlcon`. `fmincon` optimizes such that  $c(x) \leq 0$  and  $\text{ceq}(x) = 0$ . If no bounds exist, set  $\text{lb} = []$  and/or  $\text{ub} = []$ .

$x = \text{fmincon}(\text{fun}, x_0, A, b, \text{Aeq}, \text{beq}, \text{lb}, \text{ub}, \text{nonlcon}, \text{options})$  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 = []`.

$x = \text{fmincon}(\text{problem})$  finds the minimum for `problem`, where `problem` is a structure described in “Input Arguments” on page 16-81. Create the `problem` structure by

exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11.

`[x,fval] = fmincon( ___ )`, for any syntax, returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag,output] = fmincon( ___ )` additionally returns a value `exitflag` that describes the exit condition of `fmincon`, and a structure `output` with information about the optimization process.

`[x,fval,exitflag,output,lambda,grad,hessian] = fmincon( ___ )` additionally returns:

- `lambda` — Structure with fields containing the Lagrange multipliers at the solution `x`.
- `grad` — Gradient of `fun` at the solution `x`.
- `hessian` — Hessian of `fun` at the solution `x`. See “fmincon Hessian” on page 3-31.

## 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 1-13.

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

Find the minimum value starting from the point `[-1,2]`, constrained to have  $x(1) + 2x(2) \leq 1$ . Express this constraint in the form  $Ax \leq b$  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.

$x = 1 \times 2$

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  $A*x \leq b$  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.

$x = 1 \times 2$

```
0.4149    0.1701
```

### Bound Constraints

Find the minimum of an objective function in the presence of bound constraints.

The objective function is a simple algebraic function of two variables.

```
fun = @(x)1+x(1)/(1+x(2)) - 3*x(1)*x(2) + x(2)*(1+x(1));
```

Look in the region where  $x$  has positive values,  $x(1) \leq 1$ , 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. Find the minimum of `fun`, subject to the bound constraints.

```
x0 = [0.5,1];
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
```

Local minimum found that satisfies the constraints.

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

<stopping criteria details>

```
x =
```

```
1.0000    2.0000
```

A different initial point can lead to a different solution.

```
x0 = x0/5;  
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
```

Local minimum found that satisfies the constraints.

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

<stopping criteria details>

```
x =  
  
1.0e-06 *  
0.4000    0.4000
```

To see which solution is better, see “Obtain the Objective Function Value” on page 16-75.

### **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  $0 \leq x(1) \leq 0.5$ ,  $0.2 \leq x(2) \leq 0.8$ .

```
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-ord optimal.
0	3	1.000000e+00	0.000e+00	1.000e+00	0.000e+00	2.000e-
1	12	8.913011e-01	0.000e+00	1.176e-01	2.353e-01	1.107e-
2	22	8.047847e-01	0.000e+00	8.235e-02	1.900e-01	1.330e-
3	28	4.197517e-01	0.000e+00	3.430e-01	1.217e-01	6.172e-
4	31	2.733703e-01	0.000e+00	1.000e+00	5.254e-02	5.705e-
5	34	2.397111e-01	0.000e+00	1.000e+00	7.498e-02	3.164e-
6	37	2.036002e-01	0.000e+00	1.000e+00	5.960e-02	3.106e-
7	40	1.164353e-01	0.000e+00	1.000e+00	1.459e-01	1.059e-
8	43	1.161753e-01	0.000e+00	1.000e+00	1.754e-01	7.383e-
9	46	5.901600e-02	0.000e+00	1.000e+00	1.547e-02	7.278e-
10	49	4.533081e-02	2.898e-03	1.000e+00	5.393e-02	1.252e-
11	52	4.567454e-02	2.225e-06	1.000e+00	1.492e-03	1.679e-
12	55	4.567481e-02	4.405e-12	1.000e+00	2.095e-06	1.502e-
13	58	4.567481e-02	2.220e-16	1.000e+00	2.442e-09	1.287e-

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 2-25. The objective function is Rosenbrock's function,

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

which has gradient

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

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

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

Save this code as a file named `rosenbrockwithgrad.m` on your MATLAB® path.

Create options to use the objective function gradient.

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

Create the other inputs for the problem. Then call `fmincon`.

```
fun = @rosenbrockwithgrad;
x0 = [-1,2];
A = [];
b = [];
Aeq = [];
beq = [];
lb = [-2,-2];
ub = [2,2];
nonlcon = [];
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the 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 16-71 using a problem structure instead of separate arguments.

Create the options and a problem structure. See [problem](#) for the field names and required fields.

```
options = optimoptions('fmincon','Display','iter','Algorithm','sqp');
problem.options = options;
problem.solver = 'fmincon';
problem.objective = @(x)100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
problem.x0 = [0,0];
```

Create a function file for the nonlinear constraint function representing  $\text{norm}(x)^2 \leq 1$ .

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

Save this as a file named `unitdisk.m` on your MATLAB path.

Include the nonlinear constraint function in `problem`.

```
problem.nonlcon = @unitdisk;
```

Solve the problem.

```
x = fmincon(problem)
```

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-01
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-01
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-01
6	37	2.036002e-01	0.000e+00	1.000e+00	5.960e-02	3.106e-01
7	40	1.164353e-01	0.000e+00	1.000e+00	1.459e-01	1.059e-01
8	43	1.161753e-01	0.000e+00	1.000e+00	1.754e-01	7.383e-01
9	46	5.901602e-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-01
12	55	4.567481e-02	4.406e-12	1.000e+00	2.095e-06	1.501e-01
13	58	4.567481e-02	0.000e+00	1.000e+00	2.158e-09	1.511e-01

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.

<stopping criteria details>

x =

```
0.7864    0.6177
```

The iterative display and solution are the same as in “Nondefault Options” on page 16-71.

### Obtain the Objective Function Value

Call `fmincon` with the `fval` output to obtain the value of the objective function at the solution.

The “Bound Constraints” on page 16-69 example shows two solutions. Which is better? Run the example requesting the `fval` output as well as the solution.

```
fun = @(x)1+x(1)./(1+x(2)) - 3*x(1).*x(2) + x(2).*(1+x(1));
lb = [0,0];
ub = [1,2];
A = [];
b = [];
Aeq = [];
beq = [];
x0 = [0.5,1];
[x,fval] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
```

Local minimum found that satisfies the constraints.

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

<stopping criteria details>

```
x =
    1.0000    2.0000

fval =
   -0.6667
```

Run the problem using a different starting point `x0`.

```
x0 = x0/5;
[x2,fval2] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
```

Local minimum found that satisfies the constraints.

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

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

<stopping criteria details>

```
x2 =
    1.0e-06 *
    0.4000    0.4000

fval2 =
    1.0000
```

This solution has an objective function value `fval2 = 1`, which is higher than the first value `fval = -0.6667`. The first solution `x` has a lower local minimum objective function value.

### Examine Solution Using Extra Outputs

To easily examine the quality of a solution, request the `exitflag` and output `outputs`.

Set up the problem of minimizing Rosenbrock's function on the unit disk,  $\|x\|^2 \leq 1$ . First create a function that represents the nonlinear constraint. Save this as a file named `unitdisk.m` on your MATLAB® path.

```
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: 6.9164e-06  
    algorithm: 'interior-point'  
    firstorderopt: 2.0934e-08  
    cgiterations: 4  
    message: '...'
```



- 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: 6.9164e-06  
        algorithm: 'interior-point'  
        firstorderopt: 2.0934e-08  
        cgiterations: 4  
        message: '...'  
  
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 =
```

```
497.2876 -314.5576
-314.5576 200.2383
```

- 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 3-31.

## 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 2-28 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 2-29. For an example using Symbolic Math Toolbox to compute the gradient and Hessian, see “Symbolic Math Toolbox Calculates Gradients and Hessians” on page 6-115.

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 2-31.

```
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  $x_0$ ). For large problems, pass **A** as a sparse matrix.

**A** encodes the **M** linear inequalities

$$A*x \leq b,$$

where  $x$  is the column vector of **N** variables  $x(:)$ , and **b** is a column vector with **M** elements.

For example, to specify

$$\begin{array}{rclcl} x_1 & + & 2x_2 & \leq & 10 \\ 3x_1 & + & 4x_2 & \leq & 20 \\ 5x_1 & + & 6x_2 & \leq & 30, \end{array}$$

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  $A = \text{ones}(1,N)$  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

$$A*x \leq b,$$

where  $x$  is the column vector of **N** variables  $x(:)$ , and **A** is a matrix of size **M**-by-**N**.

For example, to specify

$$\begin{array}{rclcl}
 x_1 & + & 2x_2 & \leq & 10 \\
 3x_1 & + & 4x_2 & \leq & 20 \\
 5x_1 & + & 6x_2 & \leq & 30,
 \end{array}$$

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

### 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

$$\begin{array}{rclclcl}
 x_1 & + & 2x_2 & + & 3x_3 & = & 10 \\
 2x_1 & + & 4x_2 & + & x_3 & = & 20,
 \end{array}$$

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

$$\text{Aeq} * x = \text{beq},$$

where `x` is the column vector of `N` variables `x(:)`, and `Aeq` is a matrix of size `Me-by-N`.

For example, to specify

$$\begin{array}{rcccccc} x_1 & + & 2x_2 & + & 3x_3 & = & 10 \\ 2x_1 & + & 4x_2 & + & x_3 & = & 20, \end{array}$$

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

### **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

$$x(i) \geq lb(i) \quad \text{for all } i.$$

If `numel(lb) < numel(x0)`, then `lb` specifies that

$$x(i) \geq lb(i) \quad \text{for } 1 \leq i \leq \text{numel}(lb).$$

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

$$x(i) \leq ub(i) \quad \text{for all } i.$$

If `numel(ub) < numel(x0)`, then `ub` specifies that

$$x(i) \leq ub(i) \quad \text{for } 1 \leq i \leq \text{numel}(ub).$$

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
 
$$c(x) \leq 0 \quad \text{for all entries of } c.$$
- `ceq(x)` is the array of nonlinear equality constraints at `x`. `fmincon` attempts to satisfy
 
$$ceq(x) = 0 \quad \text{for all entries of } ceq.$$

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 2-48.

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 15-8 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 2-82.

### **All Algorithms**

Algorithm	<p>Choose the optimization algorithm:</p> <ul style="list-style-type: none"><li>• 'interior-point' (default)</li><li>• 'trust-region-reflective'</li><li>• 'sqp'</li><li>• 'sqp-legacy' (optimoptions only)</li><li>• 'active-set'</li></ul> <p>For information on choosing the algorithm, see “Choosing the Algorithm” on page 2-8.</p> <p>The trust-region-reflective algorithm requires:</p> <ul style="list-style-type: none"><li>• A gradient to be supplied in the objective function</li><li>• SpecifyObjectiveGradient to be set to true</li><li>• Either bound constraints or linear equality constraints, but not both</li></ul> <p>If you select the 'trust-region-reflective' algorithm and these conditions are not all satisfied, fmincon throws an error.</p> <p>The 'active-set', 'sqp-legacy', and 'sqp' algorithms are not large-scale. See “Large-Scale vs. Medium-Scale Algorithms” on page 2-14.</p>
CheckGradients	<p>Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. Choices are false (default) or true.</p> <p>For optimset, the name is DerivativeCheck and the values are 'on' or 'off'. See “Current and Legacy Option Name Tables” on page 15-31.</p>
ConstraintTolerance	<p>Tolerance on the constraint violation, a positive scalar. The default is 1e-6. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For optimset, the name is TolCon. See “Current and Legacy Option Name Tables” on page 15-31.</p>

---

<i>Diagnostics</i>	Display diagnostic information about the function to be minimized or solved. Choices are 'off' (default) or 'on'.
<i>DiffMaxChange</i>	Maximum change in variables for finite-difference gradients (a positive scalar). The default is Inf.
<i>DiffMinChange</i>	Minimum change in variables for finite-difference gradients (a positive scalar). The default is 0.
<i>Display</i>	Level of display (see "Iterative Display" on page 3-17): <ul style="list-style-type: none"><li>• 'off' or 'none' displays no output.</li><li>• 'iter' displays output at each iteration, and gives the default exit message.</li><li>• 'iter-detailed' displays output at each iteration, and gives the technical exit message.</li><li>• 'notify' displays output only if the function does not converge, and gives the default exit message.</li><li>• 'notify-detailed' displays output only if the function does not converge, and gives the technical exit message.</li><li>• 'final' (default) displays only the final output, and gives the default exit message.</li><li>• 'final-detailed' displays only the final output, and gives the technical exit message.</li></ul>

<code>FiniteDifferenceStepSize</code>	<p>Scalar or vector step size factor for finite differences. When you set <code>FiniteDifferenceStepSize</code> to a vector <code>v</code>, the forward finite differences <code>delta</code> are</p> $\text{delta} = v.*\text{sign}'(x).*\max(\text{abs}(x),\text{TypicalX});$ <p>where <math>\text{sign}'(x) = \text{sign}(x)</math> except <math>\text{sign}'(0) = 1</math>. Central finite differences are</p> $\text{delta} = v.*\max(\text{abs}(x),\text{TypicalX});$ <p>Scalar <code>FiniteDifferenceStepSize</code> expands to a vector. The default is <code>sqrt(eps)</code> for forward finite differences, and <code>eps^(1/3)</code> for central finite differences.</p> <p>For <code>optimset</code>, the name is <code>FinDiffRelStep</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>FiniteDifferenceType</code>	<p>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 <code>FiniteDifferenceType</code> only when <code>CheckGradients</code> is set to <code>true</code>.</p> <p><code>fmincon</code> 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 <code>HonorBounds</code> option is set to <code>false</code>.</p> <p>For <code>optimset</code>, the name is <code>FinDiffType</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>FunValCheck</code>	<p>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.</p>

---

MaxFunctionEvaluations	<p>Maximum number of function evaluations allowed, a positive integer. The default value for all algorithms except <code>interior-point</code> is <math>100 \times \text{numberOfVariables}</math>; for the <code>interior-point</code> algorithm the default is 3000. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxFunEvals</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
MaxIterations	<p>Maximum number of iterations allowed, a positive integer. The default value for all algorithms except <code>interior-point</code> is 400; for the <code>interior-point</code> algorithm the default is 1000. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxIter</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
OptimalityTolerance	<p>Termination tolerance on the first-order optimality (a positive scalar). The default is <math>1e-6</math>. See “First-Order Optimality Measure” on page 3-12.</p> <p>For <code>optimset</code>, the name is <code>TolFun</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
OutputFcn	<p>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 (<code>[]</code>). See “Output Function Syntax” on page 15-37.</p>

**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
- `'optimplotconstrviolation'` plots the maximum constraint violation
- `'optimplotstepsize'` plots the step size
- `'optimplotfirstorderopt'` plots the first-order optimality measure

For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.

For `optimset`, the name is `PlotFcns`. See “Current and Legacy Option Name Tables” on page 15-31.

**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 Name Tables” on page 15-31.

SpecifyObjectiveGradient	<p>Gradient for the objective function defined by the user. See the description of <code>fun</code> to see how to define the gradient in <code>fun</code>. The default, <code>false</code>, causes <code>fmincon</code> to estimate gradients using finite differences. Set to <code>true</code> to have <code>fmincon</code> use a user-defined gradient of the objective function. To use the <code>'trust-region-reflective'</code> algorithm, you must provide the gradient, and set <code>SpecifyObjectiveGradient</code> to <code>true</code>.</p> <p>For <code>optimset</code>, the name is <code>GradObj</code> and the values are <code>'on'</code> or <code>'off'</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
StepTolerance	<p>Termination tolerance on <code>x</code>, a positive scalar. The default value for all algorithms except <code>'interior-point'</code> is <code>1e-6</code>; for the <code>'interior-point'</code> algorithm, the default is <code>1e-10</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolX</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
TypicalX	<p>Typical <code>x</code> values. The number of elements in <code>TypicalX</code> is equal to the number of elements in <code>x0</code>, the starting point. The default value is <code>ones(numberofvariables,1)</code>. <code>fmincon</code> uses <code>TypicalX</code> for scaling finite differences for gradient estimation.</p> <p>The <code>'trust-region-reflective'</code> algorithm uses <code>TypicalX</code> only for the <code>CheckGradients</code> option.</p>
UseParallel	<p>When <code>true</code>, <code>fmincon</code> estimates gradients in parallel. Disable by setting to the default, <code>false</code>. <code>trust-region-reflective</code> requires a gradient in the objective, so <code>UseParallel</code> does not apply. See “Parallel Computing”.</p>
<b>Trust-Region-Reflective Algorithm</b>	
FunctionTolerance	<p>Termination tolerance on the function value, a positive scalar. The default is <code>1e-6</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolFun</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>

**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 Name Tables” on page 15-31.

**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

$$W = \text{hmfun}(\text{Hinfo}, Y)$$

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

$$[f, g, \text{Hinfo}] = \text{fun}(x)$$

$Y$  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 2-70.

---

**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 6-110 for an example.

For `optimset`, the name is `HessMult`. See “Current and Legacy Option Name Tables” on page 15-31.



<i>HessPattern</i>	<p>Sparsity pattern of the Hessian for finite differencing. Set <math>\text{HessPattern}(i, j) = 1</math> when you can have <math>\partial^2 \text{fun} / \partial x(i) \partial x(j) \neq 0</math>. Otherwise, set <math>\text{HessPattern}(i, j) = 0</math>.</p> <p>Use <i>HessPattern</i> when it is inconvenient to compute the Hessian matrix <i>H</i> in <i>fun</i>, but you can determine (say, by inspection) when the <i>i</i>th component of the gradient of <i>fun</i> depends on <math>x(j)</math>. <i>fmincon</i> can approximate <i>H</i> via sparse finite differences (of the gradient) if you provide the sparsity structure of <i>H</i> as the value for <i>HessPattern</i>. In other words, provide the locations of the nonzeros.</p> <p>When the structure is unknown, do not set <i>HessPattern</i>. The default behavior is as if <i>HessPattern</i> is a dense matrix of ones. Then <i>fmincon</i> 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.</p>
<i>MaxPCGIter</i>	<p>Maximum number of preconditioned conjugate gradient (PCG) iterations, a positive scalar. The default is <math>\max(1, \text{floor}(\text{numberOfVariables}/2))</math> for bound-constrained problems, and is <i>numberOfVariables</i> for equality-constrained problems. For more information, see “Preconditioned Conjugate Gradient Method” on page 6-24.</p>
<i>PrecondBandWidth</i>	<p>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 <i>PrecondBandWidth</i> to <i>Inf</i> 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.</p>
<i>SubproblemAlgorithm</i>	<p>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 6-22.</p>
<i>TolPCG</i>	<p>Termination tolerance on the PCG iteration, a positive scalar. The default is 0.1.</p>

**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 2-84.
- For `optimset`, the name is `TolFun`. See “Current and Legacy Option Name Tables” on page 15-31.
- MaxSQPIter** Maximum number of SQP iterations allowed, a positive integer. The default is  $10 \cdot \max(\text{numberOfVariables}, \text{numberOfInequalities} + \text{numberOfBounds})$ .
- RelLineSrchBnd** Relative bound (a real nonnegative scalar value) on the line search step length. The total displacement in  $x$  satisfies  $|\Delta x(i)| \leq \text{relLineSrchBnd} \cdot \max(|x(i)|, |\text{typical}x(i)|)$ . This option provides control over the magnitude of the displacements in  $x$  for cases in which the solver takes steps that are considered too large. The default is no bounds (`[]`).
- RelLineSrchBndDuration** Number of iterations for which the bound specified in `RelLineSrchBnd` should be active (default is 1).
- TolConSQP** Termination tolerance on inner iteration SQP constraint violation, a positive scalar. The default is 1e-6.

**Interior-Point Algorithm**

**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 Name Tables” on page 15-31.

**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 Name Tables” on page 15-31.

**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 Name Tables” on page 15-31.

<code>HonorBounds</code>	<p>The default <code>true</code> ensures that bound constraints are satisfied at every iteration. Disable by setting to <code>false</code>.</p> <p>For <code>optimset</code>, the name is <code>AlwaysHonorConstraints</code> and the values are <code>'bounds'</code> or <code>'none'</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>InitBarrierParam</code>	<p>Initial barrier value, a positive scalar. Sometimes it might help to try a value above the default <code>0.1</code>, especially if the objective or constraint functions are large.</p>
<code>InitTrustRegionRadius</code>	<p>Initial radius of the trust region, a positive scalar. On badly scaled problems it might help to choose a value smaller than the default <math>\sqrt{n}</math>, where <math>n</math> is the number of variables.</p>
<code>MaxProjCGIter</code>	<p>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 <code>2*(numberOfVariables - numberOfEqualities)</code>.</p>
<code>ObjectiveLimit</code>	<p>A tolerance (stopping criterion) that is a scalar. If the objective function value goes below <code>ObjectiveLimit</code> and the iterate is feasible, the iterations halt, because the problem is presumably unbounded. The default value is <code>-1e20</code>.</p>
<code>ScaleProblem</code>	<p><code>true</code> causes the algorithm to normalize all constraints and the objective function. Disable by setting to the default <code>false</code>.</p> <p>For <code>optimset</code>, the values are <code>'obj-and-constr'</code> or <code>'none'</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>SubproblemAlgorithm</code>	<p>Determines how the iteration step is calculated. The default, <code>'factorization'</code>, is usually faster than <code>'cg'</code> (conjugate gradient), though <code>'cg'</code> might be faster for large problems with dense Hessians. See “fmincon Interior Point Algorithm” on page 6-37.</p>
<code>TolProjCG</code>	<p>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 <code>0.01</code>.</p>

*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 Name Tables” on page 15-31.

```
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
<code>objective</code>	Objective function
<code>x0</code>	Initial point for $x$
<code>Aineq</code>	Matrix for linear inequality constraints
<code>bineq</code>	Vector for linear inequality constraints
<code>Aeq</code>	Matrix for linear equality constraints
<code>beq</code>	Vector for linear equality constraints
<code>lb</code>	Vector of lower bounds
<code>ub</code>	Vector of upper bounds
<code>nonlcon</code>	Nonlinear constraint function
<code>solver</code>	<code>'fmincon'</code>

Field Name	Entry
options	Options created with <code>optoptions</code>

You must supply at least the `objective`, `x0`, `solver`, and `options` fields in the problem structure.

The simplest way to obtain a problem structure is to export the problem from the Optimization app.

Data Types: `struct`

## Output Arguments

### **x — Solution**

real vector | real array

Solution, returned as a real vector or real array. The size of `x` is the same as the size of `x0`. Typically, `x` is a local solution to the problem when `exitflag` is positive. For information on the quality of the solution, see “When the Solver Succeeds” on page 4-22.

### **fval — Objective function value at solution**

real number

Objective function value at the solution, returned as a real number. Generally, `fval = fun(x)`.

### **exitflag — Reason `fmincon` stopped**

integer

Reason `fmincon` stopped, returned as an integer.

All Algorithms:

- 1 First-order optimality measure was less than `options.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  $2 * \text{options.StepTolerance}$  and maximum constraint violation was less than `options.ConstraintTolerance`.
- 5 Magnitude of directional derivative in search direction was less than  $2 * \text{options.OptimalityTolerance}$  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

<code>funcCount</code>	Number of function evaluations
<code>lssteplength</code>	Size of line search step relative to search direction (active-set and sqp algorithms only)
<code>constrviolation</code>	Maximum of constraint functions
<code>stepsize</code>	Length of last displacement in $x$ (not in active-set algorithm)
<code>algorithm</code>	Optimization algorithm used
<code>cgiterations</code>	Total number of PCG iterations (trust-region-reflective and interior-point algorithms)
<code>firstorderopt</code>	Measure of first-order optimality
<code>message</code>	Exit message

**lambda — Lagrange multipliers at the solution**

structure

Lagrange multipliers at the solution, returned as a structure with fields:

<code>lower</code>	Lower bounds corresponding to <code>lb</code>
<code>upper</code>	Upper bounds corresponding to <code>ub</code>
<code>ineqlin</code>	Linear inequalities corresponding to <code>A</code> and <code>b</code>
<code>eqlin</code>	Linear equalities corresponding to <code>Aeq</code> and <code>beq</code>
<code>ineqnonlin</code>	Nonlinear inequalities corresponding to the <code>c</code> in <code>nonlcon</code>
<code>eqnonlin</code>	Nonlinear equalities corresponding to the <code>ceq</code> in <code>nonlcon</code>

**grad — Gradient at the solution**

real vector

Gradient at the solution, returned as a real vector. `grad` gives the gradient of `fun` at the point `x(:)`.

**hessian — Approximate Hessian**

real matrix

Approximate Hessian, returned as a real matrix. For the meaning of `hessian`, see “Hessian” on page 3-30.



## 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 2-28.
  - 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.

## Definitions

### 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,

$$\nabla_{xx}^2 L(x, \lambda) = \nabla^2 f(x) + \sum \lambda_i \nabla^2 c_i(x) + \sum \lambda_i \nabla^2 ceq_i(x). \quad (16-1)$$

For details of how to supply a Hessian to the `trust-region-reflective` or `interior-point` algorithms, see “Including Hessians” on page 2-28.

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 2-32:

- `'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 2-31.

## Algorithms

### Choosing the Algorithm

For help choosing the algorithm, see “`fmincon` Algorithms” on page 2-8. 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 6-37. 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 16-105. “fmincon SQP Algorithm” on page 6-36 describes the main differences. In summary, these differences are:

- “Strict Feasibility With Respect to Bounds” on page 6-36
- “Robustness to Non-Double Results” on page 6-36
- “Refactored Linear Algebra Routines” on page 6-36
- “Reformulated Feasibility Routines” on page 6-36

## Active-Set Optimization

fmincon uses a sequential quadratic programming (SQP) method. In this method, the function solves a quadratic programming (QP) subproblem at each iteration. fmincon updates an estimate of the Hessian of the Lagrangian at each iteration using the BFGS formula (see fminunc and references [7] and [8]).

fmincon performs a line search using a merit function similar to that proposed by [6], [7], and [8]. The QP subproblem is solved using an active set strategy similar to that described in [5]. “fmincon Active Set Algorithm” on page 6-27 describes this algorithm in detail.

See also “SQP Implementation” on page 6-30 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 6-22.

## 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

### 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 14-6.

### See Also

fminbnd | fminsearch | fminunc | optimoptions | optimtool

### Topics

“Solver-Based Nonlinear Optimization”

“Solver-Based Optimization Problem Setup”

“Constrained Nonlinear Optimization Algorithms” on page 6-22

**Introduced before R2006a**

## fminimax

Solve minimax constraint problem

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

$$\min_x \max_i F_i(x) \text{ such that } \begin{cases} c(x) \leq 0 \\ ceq(x) = 0 \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub \end{cases}$$

where  $b$  and  $beq$  are vectors,  $A$  and  $Aeq$  are matrices, and  $c(x)$ ,  $ceq(x)$ , and  $F(x)$  are functions that return vectors.  $F(x)$ ,  $c(x)$ , and  $ceq(x)$  can be nonlinear functions.

$x$ ,  $lb$ , and  $ub$  can be passed as vectors or matrices; see “Matrix Arguments” on page 2-40.

You can also solve max-min problems with fminimax, using the identity

$$\max_x \min_i F_i(x) = - \min_x \max_i (-F_i(x)).$$

You can solve problems of the form

$$\min_x \max_i |F_i(x)|$$

by using the `AbsoluteMaxObjectiveCount` option; see “Solve Minimax Problem Using Absolute Value of One Objective” on page 16-116.

## 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

`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 2-70 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  $A*x \leq b$ .

`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  $lb \leq x \leq ub$ . 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 2-42.

---

---

**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  $c(x) \leq 0$  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`, where `problem` is a structure described in `problem`. Create the problem structure by exporting a problem from the Optimization app, as described in “Exporting Your Work” on page 5-11.

`[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`.

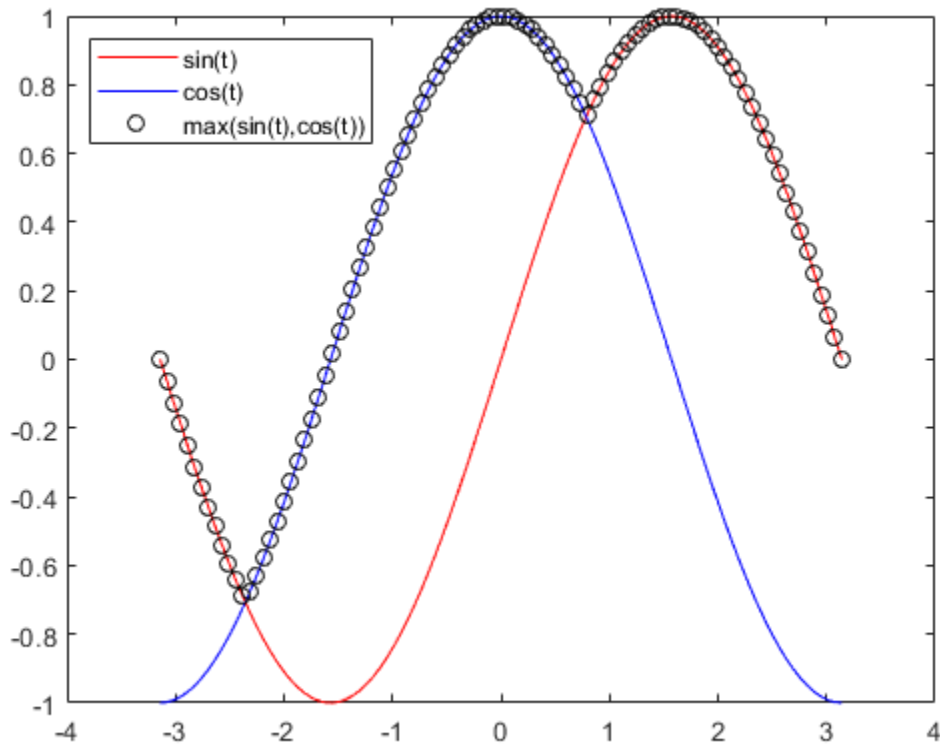
## Examples

### 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 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 8-8.

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];
```

Find the minimax point subject to the inequality  $x(1) + 3*x(2) \leq -4$ .

```
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 = 1x2
    -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 8-8.

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];
```

Set bounds that  $-2 \leq x(1) \leq 2$  and  $-1 \leq x(2) \leq 1$  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.

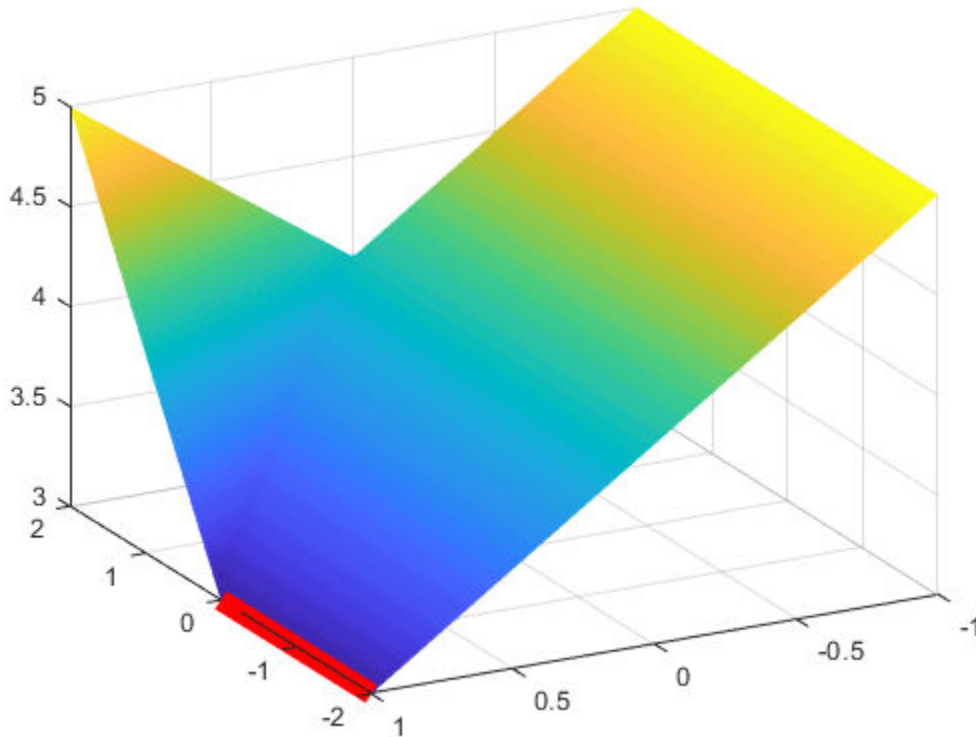
fminimax stopped because the size of the current search direction is less than twice the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.

```
x = 1x2
```

```
-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 8-8.

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  $\|x\|^2 \leq 1$ .

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  $x_0 = [0, 0]$ .

```
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 = 1x2
```

```
   -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)
```

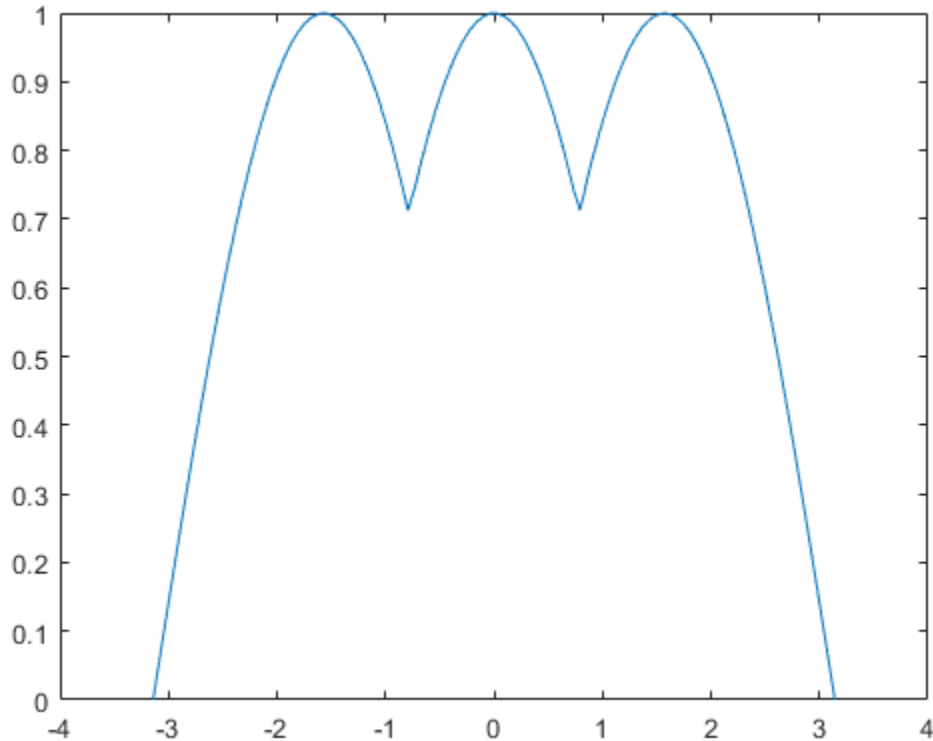
Local minimum possible. Constraints satisfied.

`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 16-110.

### 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 8-8.

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];

```

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 = 1x2
```

```
    -2.5000    2.2500
```

```
fval = 1x3
```

```
    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 8-8.

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];
```

Find the minimax point subject to the inequality  $x(1) + 3*x(2) \leq -4$ .

```
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)
```

Iter	F-count	Objective value	Max constraint	Line search steplength	Directional derivative	Procedure
0	4	0	6			
1	9	5	0	1	0.981	
2	14	4.889	0	1	-0.302	Hessian modi
3	19	3.4	8.132e-09	1	-0.302	Hessian modi

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 = 1x2
    -5.8000    0.6000
```

```
fval = 1x3
```

```

-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” (MATLAB)).

To minimize the worst-case absolute values of some elements of the vector  $F(x)$  (that is,  $\min\{\max \text{abs}\{F(x)\}\}$ ), 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 16-116.

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 `j`th column of `G` is the gradient of the `j`th 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

$$A*x \leq b,$$

where `x` is the column vector of `N` variables `x(:)`, and `b` is a column vector with `M` elements.

For example, to specify

$$\begin{array}{rclcl} x_1 & + & 2x_2 & \leq & 10 \\ 3x_1 & + & 4x_2 & \leq & 20 \\ 5x_1 & + & 6x_2 & \leq & 30, \end{array}$$

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

$$A*x \leq b,$$

where **x** is the column vector of N variables **x(:)**, and **A** is a matrix of size M-by-N.

For example, to specify

$$\begin{array}{rclcl} x_1 & + & 2x_2 & \leq & 10 \\ 3x_1 & + & 4x_2 & \leq & 20 \\ 5x_1 & + & 6x_2 & \leq & 30, \end{array}$$

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

### **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

$$\text{Aeq} * x = \text{beq},$$

where `x` is the column vector of `N` variables `x(:)`, and `Aeq` is a matrix of size `Me`-by-`N`.

For example, to specify

$$\begin{array}{rcccccc} x_1 & + & 2x_2 & + & 3x_3 & = & 10 \\ 2x_1 & + & 4x_2 & + & x_3 & = & 20, \end{array}$$

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

### **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

$$x(i) \geq lb(i) \quad \text{for all } i.$$

If `numel(lb) < numel(x0)`, then `lb` specifies that

$$x(i) \geq lb(i) \quad \text{for } 1 \leq i \leq \text{numel}(lb).$$

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

$$x(i) \leq ub(i) \quad \text{for all } i.$$

If `numel(ub) < numel(x0)`, then `ub` specifies that

$$x(i) \leq ub(i) \quad \text{for } 1 \leq i \leq \text{numel}(ub).$$

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  $c(x) \leq 0$  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 2-48 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  $x_0$ , 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 2-70 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 2-82.

For details about options that have different names for `optimset`, see “Current and Legacy Option Name Tables” on page 15-31.

Option	Description
<code>AbsoluteMaxObjectiveCount</code>	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 16-116.  For <code>optimset</code> , the name is <code>MinAbsMax</code> .
<code>ConstraintTolerance</code>	Termination tolerance on the constraint violation (a positive scalar). The default is <code>1e-6</code> . See “Tolerances and Stopping Criteria” on page 2-84.  For <code>optimset</code> , the name is <code>TolCon</code> .
<i>Diagnostics</i>	Display of diagnostic information about the function to be minimized or solved. The choices are 'on' or 'off' (the default).
<i>DiffMaxChange</i>	Maximum change in variables for finite-difference gradients (a positive scalar). The default is <code>Inf</code> .
<i>DiffMinChange</i>	Minimum change in variables for finite-difference gradients (a positive scalar). The default is <code>0</code> .

Option	Description
Display	<p>Level of display (see “Iterative Display” on page 3-17):</p> <ul style="list-style-type: none"> <li>• 'off' or 'none' displays no output.</li> <li>• 'iter' displays output at each iteration, and gives the default exit message.</li> <li>• 'iter-detailed' displays output at each iteration, and gives the technical exit message.</li> <li>• 'notify' displays output only if the function does not converge, and gives the default exit message.</li> <li>• 'notify-detailed' displays output only if the function does not converge, and gives the technical exit message.</li> <li>• 'final' (default) displays only the final output, and gives the default exit message.</li> <li>• 'final-detailed' displays only the final output, and gives the technical exit message.</li> </ul>
FiniteDifferenceStepSize	<p>Scalar or vector step size factor for finite differences. When you set <code>FiniteDifferenceStepSize</code> to a vector <code>v</code>, the forward finite differences <code>delta</code> are</p> $\text{delta} = v .* \text{sign}'(x) .* \max(\text{abs}(x), \text{TypicalX});$ <p>where <math>\text{sign}'(x) = \text{sign}(x)</math> except <math>\text{sign}'(0) = 1</math>. Central finite differences are</p> $\text{delta} = v .* \max(\text{abs}(x), \text{TypicalX});$ <p>Scalar <code>FiniteDifferenceStepSize</code> expands to a vector. The default is <math>\sqrt{\text{eps}}</math> for forward finite differences, and <math>\text{eps}^{1/3}</math> for central finite differences.</p> <p>For <code>optimset</code>, the name is <code>FinDiffRelStep</code>.</p>

Option	Description
FiniteDifferenceType	<p>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.</p> <p>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.</p>
FunctionTolerance	<p>For <code>optimset</code>, the name is <code>FinDiffType</code>.</p> <p>Termination tolerance on the function value (a positive scalar). The default is <code>1e-6</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p>
<i>FunValCheck</i>	<p>For <code>optimset</code>, the name is <code>TolFun</code>.</p> <p>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 <code>complex</code>, <code>Inf</code>, or <code>NaN</code>. The default 'off' displays no error.</p>
MaxFunctionEvaluations	<p>Maximum number of function evaluations allowed (a positive integer). The default is <code>100*numberOfVariables</code>. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p>
MaxIterations	<p>For <code>optimset</code>, the name is <code>MaxFunEvals</code>.</p> <p>Maximum number of iterations allowed (a positive integer). The default is <code>400</code>. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p>
	<p>For <code>optimset</code>, the name is <code>MaxIter</code>.</p>

---

<b>Option</b>	<b>Description</b>
<i>MaxSQPIter</i>	Maximum number of SQP iterations allowed (a positive integer). The default is $10 * \max(\text{numberOfVariables}, \text{numberOfInequalities} + \text{numberOfBounds})$ .
<i>MeritFunction</i>	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.
<i>OptimalityTolerance</i>	Termination tolerance on the first-order optimality (a positive scalar). The default is $1e-6$ . See “First-Order Optimality Measure” on page 3-12.  For optimset, the name is TolFun.
<i>OutputFcn</i>	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 Syntax” on page 15-37.

Option	Description
PlotFcn	<p>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 ( []).</p> <ul style="list-style-type: none"> <li>• 'optimplotx' plots the current point.</li> <li>• 'optimplotfunccount' plots the function count.</li> <li>• 'optimplotfval' plots the objective function values.</li> <li>• 'optimplotconstrviolation' plots the maximum constraint violation.</li> <li>• 'optimplotstepsize' plots the step size.</li> </ul> <p>For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.</p> <p>For <code>optimset</code>, the name is <code>PlotFcns</code>.</p>
<i>RelLineSrchBnd</i>	<p>Relative bound (a real nonnegative scalar value) on the line search step length such that the total displacement in <math>x</math> satisfies <math> \Delta x(i)  \leq \text{relLineSrchBnd} \cdot \max( x(i) ,  typicalx(i) )</math>. This option provides control over the magnitude of the displacements in <math>x</math> when the solver takes steps that are too large. The default is none ( []).</p>
<i>RelLineSrchBndDuration</i>	<p>Number of iterations for which the bound specified in <code>RelLineSrchBnd</code> should be active. The default is 1.</p>

Option	Description
SpecifyConstraintGradient	<p>Gradient for nonlinear constraint functions defined by the user. When this option is set to <code>true</code>, <code>fminimax</code> expects the constraint function to have four outputs, as described in <code>nonlcon</code>. When this option is set to <code>false</code> (the default), <code>fminimax</code> estimates gradients of the nonlinear constraints using finite differences.</p> <p>For <code>optimset</code>, the name is <code>GradConstr</code> and the values are <code>'on'</code> or <code>'off'</code>.</p>
SpecifyObjectiveGradient	<p>Gradient for the objective function defined by the user. Refer to the description of <code>fun</code> to see how to define the gradient. Set this option to <code>true</code> to have <code>fminimax</code> use a user-defined gradient of the objective function. The default, <code>false</code>, causes <code>fminimax</code> to estimate gradients using finite differences.</p> <p>For <code>optimset</code>, the name is <code>GradObj</code> and the values are <code>'on'</code> or <code>'off'</code>.</p>
StepTolerance	<p>Termination tolerance on <math>x</math> (a positive scalar). The default is <math>1e-6</math>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolX</code>.</p>
TolConSQP	<p>Termination tolerance on the inner iteration SQP constraint violation (a positive scalar). The default is <math>1e-6</math>.</p>
TypicalX	<p>Typical <math>x</math> values. The number of elements in <code>TypicalX</code> is equal to the number of elements in <code>x0</code>, the starting point. The default value is <code>ones(numberofvariables,1)</code>. The <code>fminimax</code> function uses <code>TypicalX</code> for scaling finite differences for gradient estimation.</p>

Option	Description
UseParallel	Option for using parallel computing. When this option is set to <code>true</code> , <code>fminimax</code> estimates gradients in parallel. The default is <code>false</code> . 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
<code>objective</code>	Objective function <code>fun</code>
<code>x0</code>	Initial point for <code>x</code>
<code>Aineq</code>	Matrix for linear inequality constraints
<code>bineq</code>	Vector for linear inequality constraints
<code>Aeq</code>	Matrix for linear equality constraints
<code>beq</code>	Vector for linear equality constraints
<code>lb</code>	Vector of lower bounds
<code>ub</code>	Vector of upper bounds
<code>nonlcon</code>	Nonlinear constraint function
<code>solver</code>	' <code>fminimax</code> '
<code>options</code>	Options created with <code>optimoptions</code>

You must supply at least the `objective`, `x0`, `solver`, and `options` fields in the problem structure.

The simplest way to obtain a problem structure is to export the problem from the Optimization app.

Data Types: `struct`



## Output Arguments

### **x — Solution**

real vector | real array

Solution, returned as a real vector or real array. The size of `x` is the same as the size of `x0`. Typically, `x` is a local solution to the problem when `exitflag` is positive. For information on the quality of the solution, see “When the Solver Succeeds” on page 4-22.

### **fval — Objective function 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 <code>x</code>
4	Magnitude of the search direction was less than the specified tolerance, and the constraint violation was less than <code>options.ConstraintTolerance</code>
5	Magnitude of the directional derivative was less than the specified tolerance, and the constraint violation was less than <code>options.ConstraintTolerance</code>
0	Number of iterations exceeded <code>options.MaxIterations</code> or the number of function evaluations exceeded <code>options.MaxFunctionEvaluations</code>
-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.

<code>iterations</code>	Number of iterations taken
<code>funcCount</code>	Number of function evaluations
<code>lssteplength</code>	Size of the line search step relative to the search direction
<code>constrviolation</code>	Maximum of the constraint functions
<code>stepsize</code>	Length of the last displacement in $x$
<code>algorithm</code>	Optimization algorithm used
<code>firstorderopt</code>	Measure of first-order optimality
<code>message</code>	Exit message

**lambda — Lagrange multipliers at solution**

structure

Lagrange multipliers at the solution, returned as a structure with the fields in this table.

<code>lower</code>	Lower bounds corresponding to <code>lb</code>
<code>upper</code>	Upper bounds corresponding to <code>ub</code>
<code>ineqlin</code>	Linear inequalities corresponding to <code>A</code> and <code>b</code>
<code>eqlin</code>	Linear equalities corresponding to <code>Aeq</code> and <code>beq</code>
<code>ineqnonlin</code>	Nonlinear inequalities corresponding to the <code>c</code> in <code>nonlcon</code>
<code>eqnonlin</code>	Nonlinear equalities corresponding to the <code>ceq</code> in <code>nonlcon</code>

## 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.

## 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 14-6.

### See Also

fgoalattain | optimoptions

### Topics

“Create Function Handle” (MATLAB)

“Multiobjective Optimization”

**Introduced before R2006a**

## fminsearch

Find minimum of unconstrained multivariable function using derivative-free method

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 2-40.

## 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

`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`, where `problem` is a structure. Create `problem` by exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11.

`[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  $x_0 = [-1.2, 1]$  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);
```

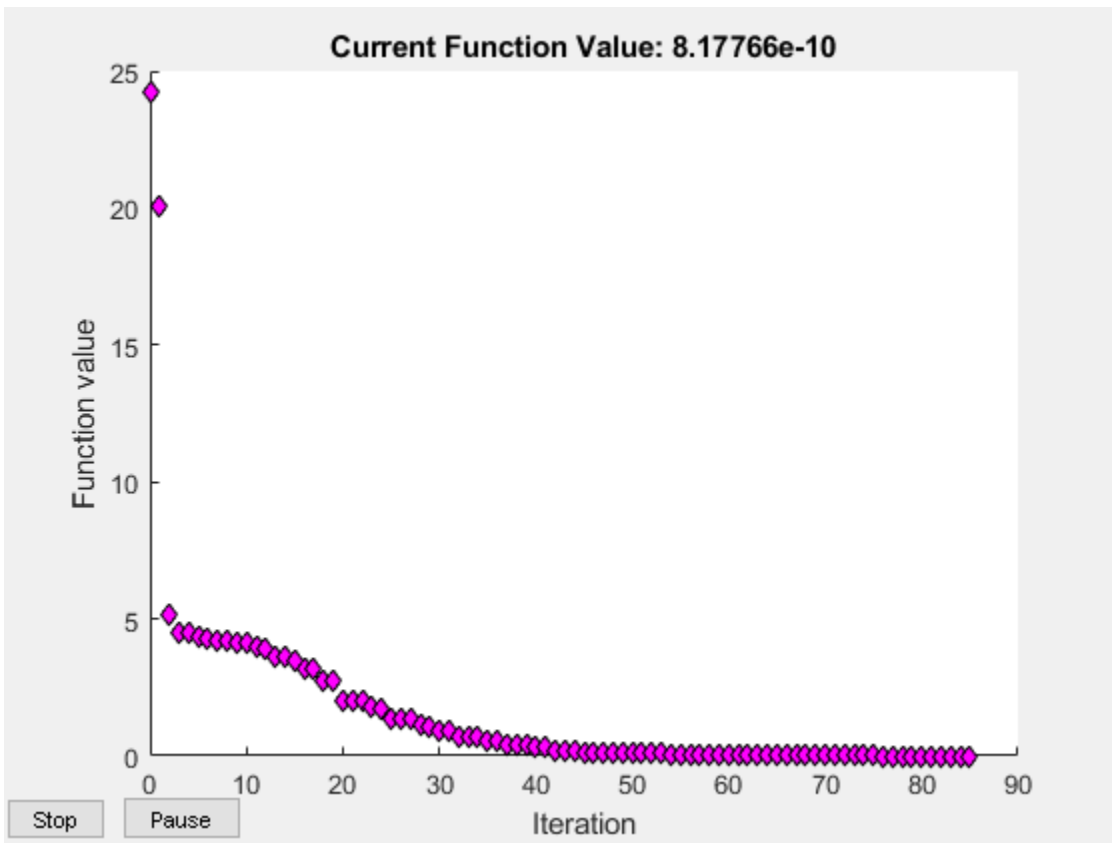
Set the objective function to Rosenbrock's function,

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

The function is minimized at the point  $x = [1, 1]$  with minimum value 0.

Set the start point to  $x_0 = [-1.2, 1]$  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 = 1x2
```

```
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  $x_0 = [0.25, -0.25]$  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  $x_0 = [-1, 1.9]$ .

```
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” (MATLAB).

### 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  $x_0$ . 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

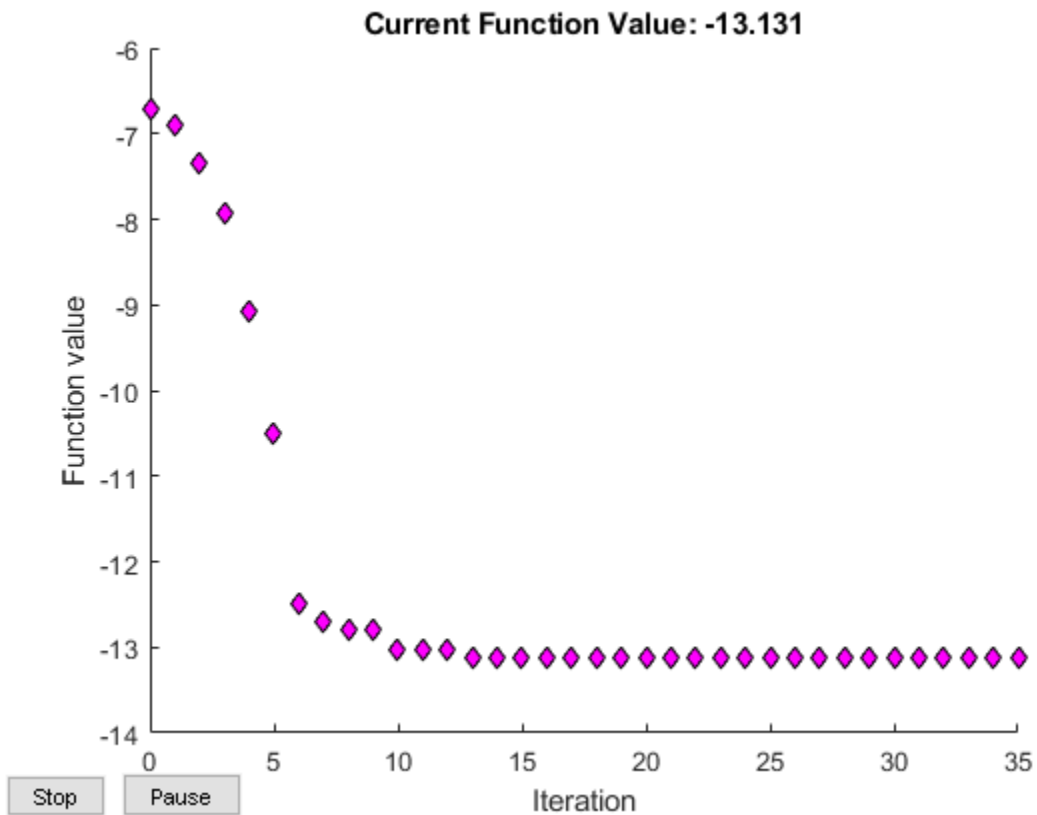
exitflag =

1

output =

struct with fields:

```
iterations: 35
funcCount: 69
algorithm: 'Nelder-Mead simplex direct search'
message: 'Optimization terminated:...'
```



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 15-8 for detailed information.

<code>Display</code>	Level of display (see “Iterative Display” on page 3-17): <ul style="list-style-type: none"><li>• <code>'notify'</code> (default) displays output only if the function does not converge.</li><li>• <code>'final'</code> displays just the final output.</li><li>• <code>'off'</code> or <code>'none'</code> displays no output.</li><li>• <code>'iter'</code> displays output at each iteration.</li></ul>
<code>FunValCheck</code>	Check whether objective function values are valid. <code>'on'</code> displays an error when the objective function returns a value that is <code>complex</code> or <code>NaN</code> . The default <code>'off'</code> displays no error.
<code>MaxFunEvals</code>	Maximum number of function evaluations allowed, a positive integer. The default is <code>200*numberOfVariables</code> . See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.
<code>MaxIter</code>	Maximum number of iterations allowed, a positive integer. The default value is <code>200*numberOfVariables</code> . See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.
<code>OutputFcn</code>	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 <code>none ([ ])</code> . See “Output Function Syntax” on page 15-37.

<code>PlotFcns</code>	<p>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 (<code>[]</code>):</p> <ul style="list-style-type: none"> <li>• <code>@optimplotx</code> plots the current point.</li> <li>• <code>@optimplotfunccount</code> plots the function count.</li> <li>• <code>@optimplotfval</code> plots the function value.</li> </ul> <p>For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.</p>
<code>TolFun</code>	<p>Termination tolerance on the function value, a positive scalar. The default is <math>1e-4</math>. See “Tolerances and Stopping Criteria” on page 2-84. Unlike other solvers, <code>fminsearch</code> stops when it satisfies <i>both</i> <code>TolFun</code> and <code>TolX</code>.</p>
<code>TolX</code>	<p>Termination tolerance on <code>x</code>, a positive scalar. The default value is <math>1e-4</math>. See “Tolerances and Stopping Criteria” on page 2-84. Unlike other solvers, <code>fminsearch</code> stops when it satisfies <i>both</i> <code>TolFun</code> and <code>TolX</code>.</p>

Example: `options = optimset('Display','iter')`

Data Types: `struct`

### **problem** — Problem structure

Problem structure, specified as a structure with the following fields.

Field Name	Entry
<code>objective</code>	Objective function
<code>x0</code>	Initial point for <code>x</code>
<code>solver</code>	' <code>fminsearch</code> '
<code>options</code>	Options structure such as returned by <code>optimset</code>

The simplest way to obtain a `problem` structure is to export the problem from the Optimization app.

Data Types: `struct`

## Output Arguments

### **x — Solution**

real vector | real array

Solution, returned as a real vector or real array. The size of `x` is the same as the size of `x0`. Typically, `x` is a local solution to the problem when `exitflag` is positive. For information on the quality of the solution, see “When the Solver Succeeds” on page 4-22.

### **fval — Objective function value at solution**

real number

Objective function value at the solution, returned as a real number. Generally, `fval = fun(x)`.

### **exitflag — Reason fminsearch stopped**

integer

Reason `fminsearch` stopped, returned as an integer.

1	The function converged to a solution <code>x</code> .
0	Number of iterations exceeded <code>options.MaxIter</code> or number of function evaluations exceeded <code>options.MaxFunEvals</code> .
-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:

<code>iterations</code>	Number of iterations
<code>funcCount</code>	Number of function evaluations
<code>algorithm</code>	'Nelder-Mead simplex direct search'
<code>message</code>	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

$$\min_x \|f(x)\|_2^2 = \min_x (f_1(x)^2 + f_2(x)^2 + \dots + f_n(x)^2)$$

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 6-11. The algorithm is not guaranteed to converge to a local minimum.

## 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

`fminbnd` | `fminunc` | `optimset` | `optimtool`

### Topics

“Create Function Handle” (MATLAB)

“Anonymous Functions” (MATLAB)

**Introduced before R2006a**

## fminunc

Find minimum of unconstrained multivariable function

Nonlinear programming solver.

Finds the minimum of a problem specified by

$$\min_x f(x)$$

where  $f(x)$  is a function that returns a scalar.

$x$  is a vector or a matrix; see “Matrix Arguments” on page 2-40.

## Syntax

```
x = fminunc(fun,x0)
x = fminunc(fun,x0,options)
x = fminunc(problem)
[x,fval] = fminunc(____)
[x,fval,exitflag,output] = fminunc(____)
[x,fval,exitflag,output,grad,hessian] = fminunc(____)
```

## Description

$x = \text{fminunc}(\text{fun},x0)$  starts at the point  $x0$  and attempts to find a local minimum  $x$  of the function described in  $\text{fun}$ . The point  $x0$  can be a scalar, vector, or matrix.

---

**Note** “Passing Extra Parameters” on page 2-70 explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

$\text{fminunc}$  is for nonlinear problems without constraints. If your problem has constraints, generally use  $\text{fmincon}$ . See “Optimization Decision Table” on page 2-6.

---

`x = fminunc(fun,x0,options)` minimizes `fun` with the optimization options specified in `options`. Use `optimoptions` to set these options.

`x = fminunc(problem)` finds the minimum for `problem`, where `problem` is a structure described in “Input Arguments” on page 16-159. Create the `problem` structure by exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11.

`[x,fval] = fminunc( ___ )`, for any syntax, returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag,output] = fminunc( ___ )` additionally returns a value `exitflag` that describes the exit condition of `fminunc`, and a structure `output` with information about the optimization process.

`[x,fval,exitflag,output,grad,hessian] = fminunc( ___ )` additionally returns:

- `grad` — Gradient of `fun` at the solution `x`.
- `hessian` — Hessian of `fun` at the solution `x`. See “fminunc Hessian” on page 3-30.

## Examples

### Minimize a Polynomial

Minimize the function  $f(x) = 3x_1^2 + 2x_1x_2 + x_2^2 - 4x_1 + 5x_2$ .

Write an anonymous function that calculates the objective.

```
fun = @(x)3*x(1)^2 + 2*x(1)*x(2) + x(2)^2 - 4*x(1) + 5*x(2);
```

Call `fminunc` to find a minimum of `fun` near `[1,1]`.

```
x0 = [1,1];
[x,fval] = fminunc(fun,x0);
```

After a few iterations, `fminunc` returns the solution, `x`, and the value of the function at `x`, `fval`.

```
x,fval
```

```
x =  
    2.2500   -4.7500  
  
fval =  
   -16.3750
```

### Supply the Gradient

`fminunc` can be faster and more reliable when you provide derivatives.

Write an objective function that returns the gradient as well as the function value. Use the conditionalized form described in “Including Gradients and Hessians” on page 2-25. The objective function is Rosenbrock’s function,

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

which has gradient

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

```
function [f,g] = rosenbrockwithgrad(x)  
% Calculate objective f  
f = 100*(x(2) - x(1)^2)^2 + (1-x(1))^2;  
  
if nargin > 1 % gradient required  
    g = [-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));  
        200*(x(2)-x(1)^2)];  
end
```

Save this code as a file on your MATLAB path, named `rosenbrockwithgrad.m`.

Create options to use the objective function’s gradient. Also, set the algorithm to `'trust-region'`.

```
options = optimoptions('fminunc','Algorithm','trust-region','SpecifyObjectiveGradient')
```

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 default value of the function tolerance.

<stopping criteria details>

x =

```
1.0000    1.0000
```

### Use a Problem Structure

Solve the same problem as in “Supply the Gradient” on page 16-154 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 2-25. The objective function is Rosenbrock’s function,

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

which has gradient

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

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

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

Save this code as a file on your MATLAB path, named `rosenbrockwithgrad.m`.

Create options to use the objective function's gradient. Also, set the algorithm to 'trust-region'.

```
options = optimoptions('fminunc','Algorithm','trust-region','SpecifyObjectiveGradient')
```

Create a problem structure including the initial point  $x_0 = [-1, 2]$ .

```
problem.options = options;  
problem.x0 = [-1,2];  
problem.objective = @rosenbrockwithgrad;  
problem.solver = 'fminunc';
```

Solve the problem.

```
x = fminunc(problem)
```

```
Local minimum found.
```

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

```
<stopping criteria details>
```

```
x =
```

```
1.0000    1.0000
```

### Obtain the Optimal Objective Function Value

Find both the location of the minimum of a nonlinear function and the value of the function at that minimum.

The objective function is

$$f(x) = x(1)e^{-\|x\|_2^2} + \|x\|_2^2/20.$$

```
fun = @(x)x(1)*exp(-(x(1)^2 + x(2)^2)) + (x(1)^2 + x(2)^2)/20;
```

Find the location and objective function value of the minimizer starting at  $x_0 = [1, 2]$ .

```
x0 = [1,2];
[x,fval] = fminunc(fun,x0)
```

Local minimum found.

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

<stopping criteria details>

x =

```
-0.6691    0.0000
```

fval =

```
-0.4052
```

### Examine the Solution Process

Choose fminunc options and outputs to examine the solution process.

Set options to obtain iterative display and use the 'quasi-newton' algorithm.

```
options = optimoptions(@fminunc,'Display','iter','Algorithm','quasi-newton');
```

The objective function is

$$f(x) = x(1)e^{-\|x\|_2^2} + \|x\|_2^2/20.$$

```
fun = @(x)x(1)*exp(-(x(1)^2 + x(2)^2)) + (x(1)^2 + x(2)^2)/20;
```

Start the minimization at  $x_0 = [1,2]$ , and obtain outputs that enable you to examine the solution quality and process.

```
x0 = [1,2];
[x,fval,exitflag,output] = fminunc(fun,x0,options)
```

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

$x = 1 \times 2$

```
-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 2-28 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 2-31.

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 15-8 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 2-82.

**All Algorithms**

<code>Algorithm</code>	Choose the <code>fminunc</code> algorithm. Choices are 'quasi-newton' (default) or 'trust-region'.  The 'trust-region' algorithm requires you to provide the gradient (see the description of <code>fun</code> ), or else <code>fminunc</code> uses the 'quasi-newton' algorithm. For information on choosing the algorithm, see “Choosing the Algorithm” on page 2-8.
<code>CheckGradients</code>	Compare user-supplied derivatives (gradient of objective) to finite-differencing derivatives. Choices are <code>false</code> (default) or <code>true</code> .  For <code>optimset</code> , the name is <code>DerivativeCheck</code> and the values are 'on' or 'off'. See “Current and Legacy Option Name Tables” on page 15-31.
<i>Diagnostics</i>	Display diagnostic information about the function to be minimized or solved. Choices are 'off' (default) or 'on'.

<i>DiffMaxChange</i>	Maximum change in variables for finite-difference gradients (a positive scalar). The default is Inf.
<i>DiffMinChange</i>	Minimum change in variables for finite-difference gradients (a positive scalar). The default is 0.
<i>Display</i>	Level of display (see “Iterative Display” on page 3-17): <ul style="list-style-type: none"><li>• 'off' or 'none' displays no output.</li><li>• 'iter' displays output at each iteration, and gives the default exit message.</li><li>• 'iter-detailed' displays output at each iteration, and gives the technical exit message.</li><li>• 'notify' displays output only if the function does not converge, and gives the default exit message.</li><li>• 'notify-detailed' displays output only if the function does not converge, and gives the technical exit message.</li><li>• 'final' (default) displays only the final output, and gives the default exit message.</li><li>• 'final-detailed' displays only the final output, and gives the technical exit message.</li></ul>

<code>FiniteDifferenceStepSize</code>	<p>Scalar or vector step size factor for finite differences. When you set <code>FiniteDifferenceStepSize</code> to a vector <code>v</code>, the forward finite differences <code>delta</code> are</p> $\text{delta} = v .* \text{sign}'(x) .* \max(\text{abs}(x), \text{TypicalX});$ <p>where <math>\text{sign}'(x) = \text{sign}(x)</math> except <math>\text{sign}'(0) = 1</math>. Central finite differences are</p> $\text{delta} = v .* \max(\text{abs}(x), \text{TypicalX});$ <p>Scalar <code>FiniteDifferenceStepSize</code> expands to a vector. The default is <code>sqrt(eps)</code> for forward finite differences, and <code>eps^(1/3)</code> for central finite differences.</p> <p>The trust-region algorithm uses <code>FiniteDifferenceStepSize</code> only when <code>CheckGradients</code> is set to <code>true</code>.</p> <p>For <code>optimset</code>, the name is <code>FinDiffRelStep</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>FiniteDifferenceType</code>	<p>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 <code>FiniteDifferenceType</code> only when <code>CheckGradients</code> is set to <code>true</code>.</p> <p>For <code>optimset</code>, the name is <code>FinDiffType</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>FunValCheck</code>	<p>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.</p>

---

MaxFunctionEvaluations	<p>Maximum number of function evaluations allowed, a positive integer. The default value is <math>100 \times \text{numberOfVariables}</math>. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxFunEvals</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
MaxIterations	<p>Maximum number of iterations allowed, a positive integer. The default value is 400. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxIter</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
OptimalityTolerance	<p>Termination tolerance on the first-order optimality (a positive scalar). The default is <math>1e-6</math>. See “First-Order Optimality Measure” on page 3-12.</p> <p>For <code>optimset</code>, the name is <code>TolFun</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
OutputFcn	<p>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 (<code>[]</code>). See “Output Function Syntax” on page 15-37.</p>

<code>PlotFcn</code>	<p>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 (<code>[]</code>):</p> <ul style="list-style-type: none"><li>• <code>'optimplotx'</code> plots the current point.</li><li>• <code>'optimplotfunccount'</code> plots the function count.</li><li>• <code>'optimplotfval'</code> plots the function value.</li><li>• <code>'optimplotstepsize'</code> plots the step size.</li><li>• <code>'optimplotfirstorderopt'</code> plots the first-order optimality measure.</li></ul> <p>For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.</p> <p>For <code>optimset</code>, the name is <code>PlotFcns</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>SpecifyObjectiveGradient</code>	<p>Gradient for the objective function defined by the user. See the description of <code>fun</code> to see how to define the gradient in <code>fun</code>. Set to <code>true</code> to have <code>fminunc</code> use a user-defined gradient of the objective function. The default <code>false</code> causes <code>fminunc</code> to estimate gradients using finite differences. You must provide the gradient, and set <code>SpecifyObjectiveGradient</code> to <code>true</code>, to use the trust-region algorithm. This option is not required for the quasi-Newton algorithm.</p> <p>For <code>optimset</code>, the name is <code>GradObj</code> and the values are <code>'on'</code> or <code>'off'</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>StepTolerance</code>	<p>Termination tolerance on <code>x</code>, a positive scalar. The default value is <code>1e-6</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolX</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>

---

TypicalX	<p>Typical <math>x</math> values. The number of elements in TypicalX is equal to the number of elements in <math>x_0</math>, the starting point. The default value is <code>ones(numberofvariables,1)</code>. <code>fminunc</code> uses TypicalX for scaling finite differences for gradient estimation.</p> <p>The trust-region algorithm uses TypicalX only for the CheckGradients option.</p>
<b>trust-region Algorithm</b>	
FunctionTolerance	<p>Termination tolerance on the function value, a positive scalar. The default is <code>1e-6</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolFun</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
HessianFcn	<p>If set to <code>[]</code> (default), <code>fminunc</code> approximates the Hessian using finite differences.</p> <p>If set to <code>'objective'</code>, <code>fminunc</code> uses a user-defined Hessian for the objective function. The Hessian is the third output of the objective function (see <code>fun</code>).</p> <p>For <code>optimset</code>, the name is <code>HessFcn</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>

**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

$$W = \text{hmfun}(\text{Hinfo}, Y)$$

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

$$[f, g, \text{Hinfo}] = \text{fun}(x)$$

$Y$  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 2-70.

---

**Note** To use the `HessianMultiplyFcn` option, `HessianFcn` must be set to `[]`.

---

For an example, see “Minimization with Dense Structured Hessian, Linear Equalities” on page 6-110.

For `optimset`, the name is `HessMult`. See “Current and Legacy Option Name Tables” on page 15-31.



<i>HessPattern</i>	<p>Sparsity pattern of the Hessian for finite differencing. Set <math>\text{HessPattern}(i, j) = 1</math> when you can have <math>\partial^2 \text{fun} / \partial x(i) \partial x(j) \neq 0</math>. Otherwise, set <math>\text{HessPattern}(i, j) = 0</math>.</p> <p>Use <i>HessPattern</i> when it is inconvenient to compute the Hessian matrix <math>H</math> in <i>fun</i>, but you can determine (say, by inspection) when the <math>i</math>th component of the gradient of <i>fun</i> depends on <math>x(j)</math>. <i>fminunc</i> can approximate <math>H</math> via sparse finite differences (of the gradient) if you provide the sparsity structure of <math>H</math> as the value for <i>HessPattern</i>. In other words, provide the locations of the nonzeros.</p> <p>When the structure is unknown, do not set <i>HessPattern</i>. The default behavior is as if <i>HessPattern</i> is a dense matrix of ones. Then <i>fminunc</i> 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.</p>
<i>MaxPCGIter</i>	<p>Maximum number of preconditioned conjugate gradient (PCG) iterations, a positive scalar. The default is <math>\max(1, \text{floor}(\text{numberOfVariables}/2))</math>. For more information, see “Trust Region Algorithm” on page 16-171.</p>
<i>PrecondBandWidth</i>	<p>Upper bandwidth of preconditioner for PCG, a nonnegative integer. By default, <i>fminunc</i> uses diagonal preconditioning (upper bandwidth of 0). For some problems, increasing the bandwidth reduces the number of PCG iterations. Setting <i>PrecondBandWidth</i> to <i>Inf</i> 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.</p>
<i>SubproblemAlgorithm</i>	<p>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.</p>
<i>TolPCG</i>	<p>Termination tolerance on the PCG iteration, a positive scalar. The default is 0.1.</p>
<b>quasi-newton Algorithm</b>	

- 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 16-170 and “Hessian Update” on page 6-9 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
<code>objective</code>	Objective function
<code>x0</code>	Initial point for <code>x</code>
<code>solver</code>	'fminunc'
<code>options</code>	Options created with <code>optimoptions</code>

The simplest way to obtain a `problem` structure is to export the problem from the Optimization app.

Data Types: `struct`

## Output Arguments

### **x — Solution**

real vector | real array

Solution, returned as a real vector or real array. The size of `x` is the same as the size of `x0`. Typically, `x` is a local solution to the problem when `exitflag` is positive. For information on the quality of the solution, see “When the Solver Succeeds” on page 4-22.

### **fval — Objective function value at solution**

real number

Objective function value at the solution, returned as a real number. Generally, `fval = fun(x)`.

### **exitflag — Reason fminunc stopped**

integer

Reason `fminunc` stopped, returned as an integer.

1	Magnitude of gradient is smaller than the <code>OptimalityTolerance</code> tolerance.
2	Change in <code>x</code> was smaller than the <code>StepTolerance</code> tolerance.
3	Change in the objective function value was less than the <code>FunctionTolerance</code> tolerance.
5	Predicted decrease in the objective function was less than the <code>FunctionTolerance</code> tolerance.
0	Number of iterations exceeded <code>MaxIterations</code> or number of function evaluations exceeded <code>MaxFunctionEvaluations</code> .
-1	Algorithm was terminated by the output function.
-3	Objective function at current iteration went below <code>ObjectiveLimit</code> .

### **output — Information about the optimization process**

structure

Information about the optimization process, returned as a structure with fields:

<code>iterations</code>	Number of iterations taken
<code>funcCount</code>	Number of function evaluations
<code>firstorderopt</code>	Measure of first-order optimality
<code>algorithm</code>	Optimization algorithm used
<code>cgiterations</code>	Total number of PCG iterations (' <code>trust-region</code> ' algorithm only)
<code>lssteplength</code>	Size of line search step relative to search direction (' <code>quasi-newton</code> ' algorithm only)
<code>stepsize</code>	Final displacement in $x$
<code>message</code>	Exit message

**grad — Gradient at the solution**

real vector

Gradient at the solution, returned as a real vector. `grad` gives the gradient of `fun` at the point `x(:)`.

**hessian — Approximate Hessian**

real matrix

Approximate Hessian, returned as a real matrix. For the meaning of `hessian`, see “Hessian” on page 3-30.

## 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 '`steepestdesc`' (and `Algorithm` to '`quasi-newton`'), although this setting is usually inefficient. See “`fminunc quasi-newton Algorithm`” on page 6-5.

## 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 6-4.

## 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 14-6.

### See Also

fmincon | fminsearch | optimoptions

### Topics

“Solver-Based Nonlinear Optimization”

“Solver-Based Optimization Problem Setup”

“Unconstrained Nonlinear Optimization Algorithms” on page 6-2

**Introduced before R2006a**

## fsemif

Find minimum of semi-infinitely constrained multivariable nonlinear function

### Equation

Finds the minimum of a problem specified by

$$\min_x f(x) \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub, \\ c(x) \leq 0, \\ ceq(x) = 0, \\ K_i(x, w_i) \leq 0, \quad 1 \leq i \leq n. \end{cases}$$

$b$  and  $beq$  are vectors,  $A$  and  $Aeq$  are matrices,  $c(x)$ ,  $ceq(x)$ , and  $K_i(x, w_i)$  are functions that return vectors, and  $f(x)$  is a function that returns a scalar.  $f(x)$ ,  $c(x)$ , and  $ceq(x)$  can be nonlinear functions. The vectors (or matrices)  $K_i(x, w_i) \leq 0$  are continuous functions of both  $x$  and an additional set of variables  $w_1, w_2, \dots, w_n$ . The variables  $w_1, w_2, \dots, w_n$  are vectors of, at most, length two.

$x$ ,  $lb$ , and  $ub$  can be passed as vectors or matrices; see “Matrix Arguments” on page 2-40.

### Syntax

```
x = fsemif(fun,x0,ntheta,semifcon)
x = fsemif(fun,x0,ntheta,semifcon,A,b)
x = fsemif(fun,x0,ntheta,semifcon,A,b,Aeq,beq)
x = fsemif(fun,x0,ntheta,semifcon,A,b,Aeq,beq,lb,ub)
x = fsemif(fun,x0,ntheta,semifcon,A,b,Aeq,beq,lb,ub,options)
x = fsemif(problem)
[x,fval] = fsemif(...)
[x,fval,exitflag] = fsemif(...)
[x,fval,exitflag,output] = fsemif(...)
[x,fval,exitflag,output,lambda] = fsemif(...)
```

## 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  $w_i \in \mathcal{R}^1$  (or  $w_i \in \mathcal{R}^2$ ). 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 2-70 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  $A*x \leq b$ .

`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  $lb \leq x \leq ub$ .

---

**Note** See “Iterations Can Violate Constraints” on page 2-42.

---

`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 16-175.

Create the `problem` structure by exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11.

`[x, fval] = fseminf(...)` returns the value of the objective function `fun` at the solution `x`.



`[x,fval,exitflag] = fseminf(...)` returns a value `exitflag` that describes the exit condition.

`[x,fval,exitflag,output] = fseminf(...)` returns a structure `output` that contains information about the optimization.

`[x,fval,exitflag,output,lambda] = fseminf(...)` returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`.

---

**Note** If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the output `fval` is `[]`.

---

## Input Arguments

“Function 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 16-179 provides the function-specific details for the `options` values.

**seminfcon**

The function that computes the vector of nonlinear inequality constraints, *c*, a vector of nonlinear equality constraints, *ceq*, and *ntheta* semi-infinite constraints (vectors or matrices) *K1*, *K2*, . . . , *Kntheta* evaluated over an interval *S* at the point *x*. The function *seminfcon* can be specified as a function handle.

```
x = fseminf(@myfun,x0,ntheta,@myinfcon)
```

where *myinfcon* is a MATLAB function such as

```
function [c,ceq,K1,K2,...,Kntheta,S] = myinfcon(x,S)
% Initial sampling interval
if isnan(S(1,1)),
    S = ...% S has ntheta rows and 2 columns
end
w1 = ...% Compute sample set
w2 = ...% Compute sample set
...
wntheta = ... % Compute sample set
K1 = ... % 1st semi-infinite constraint at x and w
K2 = ... % 2nd semi-infinite constraint at x and w
...
Kntheta = ...% Last semi-infinite constraint at x and w
c = ...      % Compute nonlinear inequalities at x
ceq = ...    % Compute the nonlinear equalities at x
```

*S* is a recommended sampling interval, which might or might not be used. Return *[]* for *c* and *ceq* if no such constraints exist.

The vectors or matrices *K1*, *K2*, . . . , *Kntheta* contain the semi-infinite constraints evaluated for a sampled set of values for the independent variables *w1*, *w2*, . . . , *wntheta*, respectively. The two-column matrix, *S*, contains a recommended sampling interval for values of *w1*, *w2*, . . . , *wntheta*, which are used to evaluate *K1*, *K2*, . . . , *Kntheta*. The *i*th row of *S* contains the recommended sampling interval for evaluating *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 6-145). 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 2-70 explains how to parametrize `semifcon`, if necessary. “Example of Creating Sampling Points” on page 6-42 contains an example of both one- and two-dimensional sampling points.

problem	<code>objective</code>	Objective function
	<code>x0</code>	Initial point for <code>x</code>
	<code>ntheta</code>	Number of semi-infinite constraints
	<code>semifcon</code>	Semi-infinite constraint function
	<code>Aineq</code>	Matrix for linear inequality constraints
	<code>bineq</code>	Vector for linear inequality constraints
	<code>Aeq</code>	Matrix for linear equality constraints
	<code>beq</code>	Vector for linear equality constraints
	<code>lb</code>	Vector of lower bounds
	<code>ub</code>	Vector of upper bounds
	<code>solver</code>	' <code>fseminf</code> '
	<code>options</code>	Options created with <code>optimoptions</code>

## 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`:

<code>exitflag</code>	Integer identifying the reason the algorithm terminated. The following lists the values of <code>exitflag</code> and the corresponding reasons the algorithm terminated.
1	Function converged to a solution <code>x</code> .
4	Magnitude of the search direction was less than the specified tolerance and constraint violation was less than <code>options.ConstraintTolerance</code> .
5	Magnitude of directional derivative was less than the specified tolerance and constraint violation was less than <code>options.ConstraintTolerance</code> .

	0	Number of iterations exceeded <code>options.MaxIterations</code> or number of function evaluations exceeded <code>options.MaxFunctionEvaluations</code> .
	-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 <code>lb</code>
	upper	Upper bounds <code>ub</code>
	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 15-8 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 2-82.

<code>CheckGradients</code>	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. The choices are <code>true</code> or the default <code>false</code> .  For <code>optimset</code> , the name is <code>DerivativeCheck</code> and the values are <code>'on'</code> or <code>'off'</code> . See “Current and Legacy Option Name Tables” on page 15-31.
<code>ConstraintTolerance</code>	Termination tolerance on the constraint violation, a positive scalar. The default is <code>1e-6</code> . See “Tolerances and Stopping Criteria” on page 2-84.  For <code>optimset</code> , the name is <code>TolCon</code> . See “Current and Legacy Option Name Tables” on page 15-31.
<i>Diagnostics</i>	Display diagnostic information about the function to be minimized or solved. The choices are <code>'on'</code> or the default <code>'off'</code> .
<i>DiffMaxChange</i>	Maximum change in variables for finite-difference gradients (a positive scalar). The default is <code>Inf</code> .
<i>DiffMinChange</i>	Minimum change in variables for finite-difference gradients (a positive scalar). The default is <code>0</code> .

**Display**

Level of display (see “Iterative Display” on page 3-17):

- '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

$$\text{delta} = v \cdot \text{sign}'(x) \cdot \max(\text{abs}(x), \text{TypicalX});$$

where  $\text{sign}'(x) = \text{sign}(x)$  except  $\text{sign}'(0) = 1$ . Central finite differences are

$$\text{delta} = v \cdot \max(\text{abs}(x), \text{TypicalX});$$

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 Name Tables” on page 15-31.

<code>FiniteDifferenceType</code>	<p>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.</p> <p>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.</p> <p>For <code>optimset</code>, the name is <code>FinDiffType</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>FunctionTolerance</code>	<p>Termination tolerance on the function value, a positive scalar. The default is <code>1e-4</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolFun</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<i>FunValCheck</i>	<p>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, <code>Inf</code>, or <code>NaN</code>. The default 'off' displays no error.</p>
<code>MaxFunctionEvaluations</code>	<p>Maximum number of function evaluations allowed, a positive integer. The default is <code>100*numberOfVariables</code>. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxFunEvals</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>



<b>MaxIterations</b>	Maximum number of iterations allowed, a positive integer. The default is <b>400</b> . See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.
	For <i>optimset</i> , the name is <b>MaxIter</b> . See “Current and Legacy Option Name Tables” on page 15-31.
<i>MaxSQPIter</i>	Maximum number of SQP iterations allowed, a positive integer. The default is $10 * \max(\text{numberOfVariables}, \text{numberOfInequalities} + \text{numberOfBounds})$ .
<b>OptimalityTolerance</b>	Termination tolerance on the first-order optimality (a positive scalar). The default is <b>1e-6</b> . See “First-Order Optimality Measure” on page 3-12.
	For <i>optimset</i> , the name is <b>TolFun</b> . See “Current and Legacy Option Name Tables” on page 15-31.
<b>OutputFcn</b>	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 <b>none ([])</b> . See “Output Function Syntax” on page 15-37.

**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.
- 'optimplotconstrviolation' plots the maximum constraint violation.
- 'optimplotstepsize' plots the step size.
- 'optimplotfirstorderopt' plots the first-order optimality measure.

For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.

For `optimset`, the name is `PlotFcns`. See “Current and Legacy Option Name Tables” on page 15-31.

***RelLineSrchBnd***

Relative bound (a real nonnegative scalar value) on the line search step length such that the total displacement in  $x$  satisfies  $|\Delta x(i)| \leq \text{relLineSrchBnd} \cdot \max(|x(i)|, |typicalx(i)|)$ . This option provides control over the magnitude of the displacements in  $x$  for cases in which the solver takes steps that `fseminf` considers too large. The default is no bounds ( []).

***RelLineSrchBndDuration***

Number of iterations for which the bound specified in `RelLineSrchBnd` should be active (default is 1)

<code>SpecifyObjectiveGradient</code>	<p>Gradient for the objective function defined by the user. See the preceding description of <code>fun</code> to see how to define the gradient in <code>fun</code>. Set to <code>true</code> to have <code>fseminf</code> use a user-defined gradient of the objective function. The default <code>false</code> causes <code>fseminf</code> to estimate gradients using finite differences.</p> <p>For <code>optimset</code>, the name is <code>GradObj</code> and the values are <code>'on'</code> or <code>'off'</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>StepTolerance</code>	<p>Termination tolerance on <code>x</code>, a positive scalar. The default value is <code>1e-4</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolX</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>TolConSQP</code>	<p>Termination tolerance on inner iteration SQP constraint violation, a positive scalar. The default is <code>1e-6</code>.</p>
<code>TypicalX</code>	<p>Typical <code>x</code> values. The number of elements in <code>TypicalX</code> is equal to the number of elements in <code>x0</code>, the starting point. The default value is <code>ones(numberofvariables,1)</code>. <code>fseminf</code> uses <code>TypicalX</code> for scaling finite differences for gradient estimation.</p>

## Notes

The optimization routine `fseminf` might vary the recommended sampling interval,  $S$ , set in `seminfcon`, during the computation because values other than the recommended interval might be more appropriate for efficiency or robustness. Also, the finite region  $w_i$ , over which  $K_i(x, w_i)$  is calculated, is allowed to vary during the optimization, provided that it does not result in significant changes in the number of local minima in  $K_i(x, w_i)$ .

## Examples

This example minimizes the function

$$(x - 1)^2,$$

subject to the constraints

$$g(x, t) = (x - 1/2) - (t - 1/2)^2 \leq 0 \quad \text{for all } 0 \leq t \leq 1.$$

The unconstrained objective function is minimized at  $x = 1$ . However, the constraint,

$$g(x, t) \leq 0 \quad \text{for all } 0 \leq t \leq 1,$$

implies  $x \leq 1/2$ . You can see this by noticing that  $(t - 1/2)^2 \geq 0$ , so

$$\max_t g(x, t) = (x - 1/2).$$

Therefore

$$\max_t g(x, t) \leq 0 \quad \text{when } x \leq 1/2.$$

To solve this problem using `fsemif`:

- 1 Write the objective function as an anonymous function:

```
objfun = @(x)(x-1)^2;
```

- 2 Write the semi-infinite constraint function, which includes the nonlinear constraints (`[]` in this case), initial sampling interval for  $t$  (0 to 1 in steps of 0.01 in this case), and the semi-infinite constraint function  $g(x, t)$ :

```
function [c, ceq, K1, s] = semifcon(x,s)
```

```
% No finite nonlinear inequality and equality constraints
```

```
c = [];
```

```
ceq = [];
```

```
% Sample set
```

```
if isnan(s)
```

```
    % Initial sampling interval
```

```
    s = [0.01 0];
```

```
end
```

```
t = 0:s(1):1;
```

```
% Evaluate the semi-infinite constraint
```

```
K1 = (x - 0.5) - (t - 0.5).^2;
```

- 3 Call `fseminf` with initial point 0.2, and view the result:

```
x = fseminf(objfun,0.2,1,@seminfcon)
```

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

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

```
Active inequalities (to within options.ConstraintTolerance = 1e-006):
   lower      upper   ineqlin  ineqnonlin
           1
```

```
x =
    0.5000
```

## Limitations

The function to be minimized, the constraints, and semi-infinite constraints, must be continuous functions of  $x$  and  $w$ . `fseminf` might only give local solutions.

When the problem is not feasible, `fseminf` attempts to minimize the maximum constraint value.

## 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 6-30. For more details on the `fseminf` algorithm, see “fseminf Problem Formulation and Algorithm” on page 6-41.

## See Also

`fmincon` | `optimoptions` | `optimtool`

## Topics

“Create Function Handle” (MATLAB)

“fseminf Problem Formulation and Algorithm” on page 6-41

“Multiobjective Optimization”

**Introduced before R2006a**

# fsolve

Solve system of nonlinear equations

Nonlinear system solver

Solves a problem specified by

$$F(x) = 0$$

for  $x$ , where  $F(x)$  is a function that returns a vector value.

$x$  is a vector or a matrix; see “Matrix Arguments” on page 2-40.

## Syntax

```
x = fsolve(fun,x0)
x = fsolve(fun,x0,options)
x = fsolve(problem)
[x,fval] = fsolve(____)
[x,fval,exitflag,output] = fsolve(____)
[x,fval,exitflag,output,jacobian] = fsolve(____)
```

## Description

`x = fsolve(fun,x0)` starts at `x0` and tries to solve the equations  $\text{fun}(x) = \mathbf{0}$ , an array of zeros.

`x = fsolve(fun,x0,options)` solves the equations with the optimization options specified in `options`. Use `optimoptions` to set these options.

`x = fsolve(problem)` solves `problem`, where `problem` is a structure described in “Input Arguments” on page 16-198. Create the `problem` structure by exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11.

`[x,fval] = fsolve(____)`, for any syntax, returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag,output] = fsolve( ___ )` additionally returns a value `exitflag` that describes the exit condition of `fsolve`, and a structure `output` with information about the optimization process.

`[x,fval,exitflag,output,jacobian] = fsolve( ___ )` returns the Jacobian of `fun` at the solution `x`.

## Examples

### Solution of 2-D Nonlinear System

This example shows how to solve two nonlinear equations in two variables. The equations are

$$\begin{aligned}e^{-e^{-(x_1+x_2)}} &= x_2(1+x_1^2) \\ x_1 \cos(x_2) + x_2 \sin(x_1) &= \frac{1}{2}.\end{aligned}$$

Convert the equations to the form  $F(x) = \mathbf{0}$ .

$$\begin{aligned}e^{-e^{-(x_1+x_2)}} - x_2(1+x_1^2) &= 0 \\ x_1 \cos(x_2) + x_2 \sin(x_1) - \frac{1}{2} &= 0.\end{aligned}$$

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

$$\begin{aligned} e^{-e^{-(x_1+x_2)}} &= x_2(1+x_1^2) \\ x_1 \cos(x_2) + x_2 \sin(x_1) &= \frac{1}{2}. \end{aligned}$$

Convert the equations to the form  $F(x) = \mathbf{0}$ .

$$\begin{aligned} e^{-e^{-(x_1+x_2)}} - x_2(1+x_1^2) &= 0 \\ x_1 \cos(x_2) + x_2 \sin(x_1) - \frac{1}{2} &= 0. \end{aligned}$$

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;
```

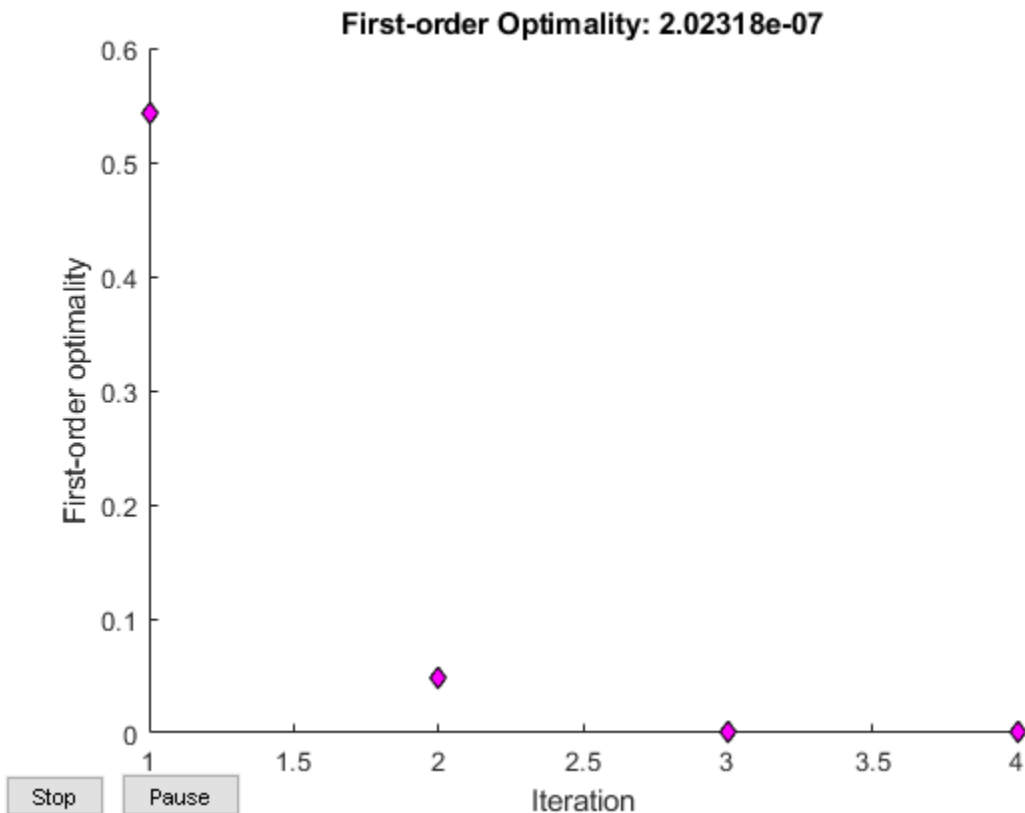
Save this code as a file named `root2d.m` on your MATLAB® path.

Solve the nonlinear system starting from the point `[0,0]` and observe the solution process.

```
fun = @root2d;  
x0 = [0,0];  
x = fsolve(fun,x0,options)
```

```
x =
```

```
    0.3532    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 16-191, 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',@optimplotfirstorder)
```

The equations in the nonlinear system are

$$\begin{aligned}e^{-e^{-(x_1+x_2)}} &= x_2(1+x_1^2) \\ x_1 \cos(x_2) + x_2 \sin(x_1) &= \frac{1}{2}.\end{aligned}$$

Convert the equations to the form  $F(x) = \mathbf{0}$ .

$$\begin{aligned}e^{-e^{-(x_1+x_2)}} - x_2(1+x_1^2) &= 0 \\ x_1 \cos(x_2) + x_2 \sin(x_1) - \frac{1}{2} &= 0.\end{aligned}$$

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.

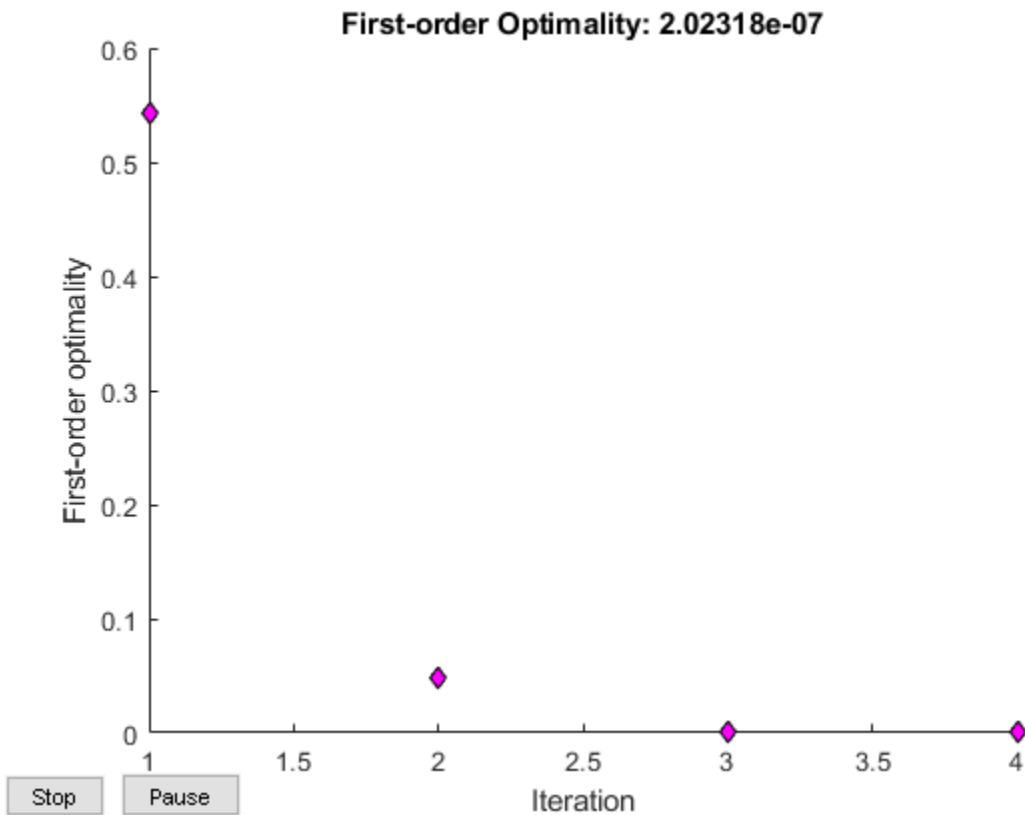
```
problem.objective = @root2d;
problem.x0 = [0,0];
problem.solver = 'fsolve';
```

Solve the problem.

```
x = fsolve(problem)
```

```
x =
```

```
    0.3532    0.6061
```



### Solution Process of Nonlinear System

This example returns the iterative display showing the solution process for the system of two equations and two unknowns

$$\begin{aligned}2x_1 - x_2 &= e^{-x_1} \\ -x_1 + 2x_2 &= e^{-x_2}.\end{aligned}$$

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  $x_0 = [-5 \ -5]$ .

First, write a file that computes F, the values of the equations at x.

```
function F = myfun(x)
F = [2*x(1) - x(2) - exp(-x(1));
     -x(1) + 2*x(2) - exp(-x(2))];
```

Save this function file as `myfun.m` on your MATLAB path.

Set up the initial point. Set options to return iterative display.

```
x0 = [-5;-5];
options = optimoptions('fsolve','Display','iter');
```

Call `fsolve`.

```
[x,fval] = fsolve(@myfun,x0,options)
```

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 =
    0.5671
```

```

0.5671

fval =
  1.0e-006 *
    -0.4059
    -0.4059

```

### Examine Matrix Equation Solution

Find a matrix  $X$  that satisfies

$$X * X * X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix},$$

starting at the point  $x = [1, 1; 1, 1]$ . Examine the `fsolve` outputs to see the solution quality and process.

Create an anonymous function that calculates the matrix equation.

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

Set options to turn off the display. Set the initial point  $x_0$ .

```
options = optimoptions('fsolve','Display','off');
x0 = ones(2);
```

Call `fsolve` and obtain information about the solution process.

```
[x,fval,exitflag,output] = fsolve(fun,x0,options);
x,fval,exitflag
```

```

x =
   -0.1291    0.8602
    1.2903    1.1612

```

```

fval =
  1.0e-09 *
   -0.1618    0.0778
    0.1160   -0.0474

```

```
exitflag =  
    1
```

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.

`fsolve` performed 35 function evaluations to find the solution, as you can see in the output structure.

```
output.funcCount  
  
ans =  
    35
```

## 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” (MATLAB)).

If the Jacobian can also be computed *and* the Jacobian option is 'on', set by

```
options = optimoptions('fsolve','SpecifyObjectiveGradient','on')
```

the function `fun` must return, in a second output argument, the Jacobian value  $J$ , a matrix, at  $x$ .

If `fun` returns a vector (matrix) of  $m$  components and  $x$  has length  $n$ , where  $n$  is the length of  $x_0$ , 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  $x_0$  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 15-8 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 2-82.

## **All Algorithms**

<code>Algorithm</code>	<p>Choose between <code>'trust-region-dogleg'</code> (default), <code>'trust-region'</code>, and <code>'levenberg-marquardt'</code>.</p> <p>The <code>Algorithm</code> 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 <code>F</code> returned by <code>fun</code>) must be at least as many as the length of <code>x</code>. Similarly, for the trust-region-dogleg algorithm, the number of equations must be the same as the length of <code>x</code>. <code>fsolve</code> uses the Levenberg-Marquardt algorithm when the selected algorithm is unavailable. For more information on choosing the algorithm, see “Choosing the Algorithm” on page 2-8.</p> <p>To set some algorithm options using <code>optimset</code> instead of <code>optimoptions</code>:</p> <ul style="list-style-type: none"><li>• <code>Algorithm</code> — Set the algorithm to <code>'trust-region-reflective'</code> instead of <code>'trust-region'</code>.</li><li>• <code>InitDamping</code> — Set the initial Levenberg-Marquardt parameter <math>\lambda</math> by setting <code>Algorithm</code> to a cell array such as <code>{'levenberg-marquardt', .005}</code>.</li></ul>
<code>CheckGradients</code>	<p>Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. The choices are <code>true</code> or the default <code>false</code>.</p> <p>For <code>optimset</code>, the name is <code>DerivativeCheck</code> and the values are <code>'on'</code> or <code>'off'</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>Diagnostics</code>	<p>Display diagnostic information about the function to be minimized or solved. The choices are <code>'on'</code> or the default <code>'off'</code>.</p>
<code>DiffMaxChange</code>	<p>Maximum change in variables for finite-difference gradients (a positive scalar). The default is <code>Inf</code>.</p>
<code>DiffMinChange</code>	<p>Minimum change in variables for finite-difference gradients (a positive scalar). The default is <code>0</code>.</p>

Display	Level of display (see “Iterative Display” on page 3-17):
	<ul style="list-style-type: none"> <li>• 'off' or 'none' displays no output.</li> <li>• 'iter' displays output at each iteration, and gives the default exit message.</li> <li>• 'iter-detailed' displays output at each iteration, and gives the technical exit message.</li> <li>• 'final' (default) displays just the final output, and gives the default exit message.</li> <li>• 'final-detailed' displays just the final output, and gives the technical exit message.</li> </ul>
FiniteDifferenceStepSize	<p>Scalar or vector step size factor for finite differences. When you set <code>FiniteDifferenceStepSize</code> to a vector <math>v</math>, the forward finite differences <math>\delta</math> are</p> $\delta = v \cdot \text{sign}'(x) \cdot \max(\text{abs}(x), \text{TypicalX});$ <p>where <math>\text{sign}'(x) = \text{sign}(x)</math> except <math>\text{sign}'(0) = 1</math>. Central finite differences are</p> $\delta = v \cdot \max(\text{abs}(x), \text{TypicalX});$ <p>Scalar <code>FiniteDifferenceStepSize</code> expands to a vector. The default is <math>\text{sqrt}(\text{eps})</math> for forward finite differences, and <math>\text{eps}^{(1/3)}</math> for central finite differences.</p> <p>For <code>optimset</code>, the name is <code>FinDiffRelStep</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>

<code>FiniteDifferenceType</code>	<p>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.</p> <p>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.</p> <p>For <code>optimset</code>, the name is <code>FinDiffType</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>FunctionTolerance</code>	<p>Termination tolerance on the function value, a positive scalar. The default is <code>1e-6</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolFun</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>FunValCheck</code>	<p>Check whether objective function values are valid. 'on' displays an error when the objective function returns a value that is complex, <code>Inf</code>, or <code>NaN</code>. The default, 'off', displays no error.</p>
<code>MaxFunctionEvaluations</code>	<p>Maximum number of function evaluations allowed, a positive integer. The default is <code>100*numberOfVariables</code>. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxFunEvals</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>MaxIterations</code>	<p>Maximum number of iterations allowed, a positive integer. The default is <code>400</code>. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxIter</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>

---

<b>OptimalityTolerance</b>	<p>Termination tolerance on the first-order optimality (a positive scalar). The default is <math>1e-6</math>. See “First-Order Optimality Measure” on page 3-12.</p> <p>Internally, the 'levenberg-marquardt' algorithm uses an optimality tolerance (stopping criterion) of <math>1e-4</math> times <b>FunctionTolerance</b> and does not use <b>OptimalityTolerance</b>.</p>
<b>OutputFcn</b>	<p>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 (<code>[]</code>). See “Output Function Syntax” on page 15-37.</p>
<b>PlotFcn</b>	<p>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 (<code>[]</code>):</p> <ul style="list-style-type: none"><li>• 'optimplotx' plots the current point.</li><li>• 'optimplotfunccount' plots the function count.</li><li>• 'optimplotfval' plots the function value.</li><li>• 'optimplotstepsize' plots the step size.</li><li>• 'optimplotfirstorderopt' plots the first-order optimality measure.</li></ul> <p>For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.</p> <p>For <code>optimset</code>, the name is <code>PlotFcns</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>

<code>SpecifyObjectiveGradient</code>	<p>If true, <code>fsolve</code> uses a user-defined Jacobian (defined in <code>fun</code>), or Jacobian information (when using <code>JacobianMultiplyFcn</code>), for the objective function. If false (default), <code>fsolve</code> approximates the Jacobian using finite differences.</p> <p>For <code>optimset</code>, the name is <code>Jacobian</code> and the values are 'on' or 'off'. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>StepTolerance</code>	<p>Termination tolerance on <code>x</code>, a positive scalar. The default is <code>1e-6</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolX</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>TypicalX</code>	<p>Typical <code>x</code> values. The number of elements in <code>TypicalX</code> is equal to the number of elements in <code>x0</code>, the starting point. The default value is <code>ones(numberofvariables,1)</code>. <code>fsolve</code> uses <code>TypicalX</code> for scaling finite differences for gradient estimation.</p> <p>The <code>trust-region-dogleg</code> algorithm uses <code>TypicalX</code> as the diagonal terms of a scaling matrix.</p>
<code>UseParallel</code>	<p>When true, <code>fsolve</code> estimates gradients in parallel. Disable by setting to the default, false. See “Parallel Computing”.</p>

**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

$$W = \text{jmfun}(Jinfo, Y, flag)$$

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

$$[F, Jinfo] = \text{fun}(x)$$

$Y$  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 2-70 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 6-110 for a similar example.

For `optimset`, the name is `JacobMult`. See “Current and Legacy Option Name Tables” on page 15-31.

- 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 `fsolve` Algorithm” on page 13-2.
- TolPCG* Termination tolerance on the PCG iteration, a positive scalar. The default is `0.1`.

### **Levenberg-Marquardt Algorithm**



<i>InitDamping</i>	Initial value of the Levenberg-Marquardt parameter, a positive scalar. Default is $1e-2$ . For details, see “Levenberg-Marquardt Method” on page 12-7.
<i>ScaleProblem</i>	'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
solver	'fsolve'
options	Options created with <code>optimoptions</code>

The simplest way of obtaining a problem structure is to export the problem from the Optimization app.

Data Types: `struct`

## **Output Arguments**

### **x — Solution**

real vector | real array

Solution, returned as a real vector or real array. The size of `x` is the same as the size of `x0`. Typically, `x` is a local solution to the problem when `exitflag` is positive. For information on the quality of the solution, see “When the Solver Succeeds” on page 4-22.

### **fval — Objective function value at the solution**

real vector

Objective function value at the solution, returned as a real vector. Generally, `fval = fun(x)`.

**exitflag — Reason fsolve stopped**

integer

Reason fsolve stopped, returned as an integer.

1	Equation solved. First-order optimality is small.
2	Equation solved. Change in x smaller than the specified tolerance.
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 <code>options.MaxIterations</code> or number of function evaluations exceeded <code>options.MaxFunctionEvaluations</code> .
-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:

<code>iterations</code>	Number of iterations taken
<code>funcCount</code>	Number of function evaluations
<code>algorithm</code>	Optimization algorithm used
<code>cgiterations</code>	Total number of PCG iterations ('trust-region' algorithm only)
<code>stepsize</code>	Final displacement in x (not in 'trust-region-dogleg')
<code>firstorderopt</code>	Measure of first-order optimality
<code>message</code>	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 16-209).

- By default `fsolve` chooses the trust-region dogleg algorithm. The algorithm is a variant of the Powell dogleg method described in [8]. It is similar in nature to the algorithm implemented in [7]. See “Trust-Region Dogleg Method” on page 13-5.
- 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 `fsolve` Algorithm” on page 13-2.

- The Levenberg-Marquardt method is described in references [4], [5], and [6]. See “Levenberg-Marquardt Method” on page 13-7.

## 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. Hillstom, *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

### 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 14-6.

## See Also

fzero | lsqcurvefit | lsqnonlin | optimoptions

## Topics

“Nonlinear Equations with Analytic Jacobian” on page 13-9

“Nonlinear Equations with Finite-Difference Jacobian” on page 13-12

“Nonlinear Equations with Jacobian” on page 13-14

“Nonlinear Equations with Jacobian Sparsity Pattern” on page 13-17

“Nonlinear Systems with Constraints” on page 13-20

“Solver-Based Optimization Problem Setup”

“Equation Solving Algorithms” on page 13-2

**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  $\text{fun}(x) = 0$ . This solution is where  $\text{fun}(x)$  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  $\text{fun}(x)$  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  $\pi$  by finding the zero of the sine function near 3.

```
fun = @sin; % function
x0 = 3; % initial point
x = fzero(fun,x0)
```

```
x = 3.1416
```

### Root Starting From an Interval

Find the zero of cosine between 1 and 2.

```
fun = @cos; % function
x0 = [1 2]; % initial interval
x = fzero(fun,x0)
```

```
x = 1.5708
```

Note that  $\cos(1)$  and  $\cos(2)$  differ in sign.

### Root of a Function Defined by a File

Find a zero of the function  $f(x) = x^3 - 2x - 5$ .

First, write a file called `f.m`.

```
function y = f(x)
y = x.^3-2*x-5;
```

Save `f.m` on your MATLAB path.

Find the zero of  $f(x)$  near 2.

```
fun = @f; % function
x0 = 2; % initial point
z = fzero(fun,x0)
```

```
z =
    2.0946
```

Since  $f(x)$  is a polynomial, you can find the same real zero, and a complex conjugate pair of zeros, using the `roots` command.

```
roots([1 0 -2 -5])

ans =
    2.0946
```

```
-1.0473 + 1.1359i  
-1.0473 - 1.1359i
```

### Root of Function with Extra Parameter

Find the root of a function that has an extra parameter.

```
myfun = @(x,c) cos(c*x); % parameterized function  
c = 2; % parameter  
fun = @(x) myfun(x,c); % function of x alone  
x = fzero(fun,0.1)  
  
x = 0.7854
```

### Nondefault Options

Plot the solution process by setting some plot functions.

Define the function and initial point.

```
fun = @(x)sin(cosh(x));  
x0 = 1;
```

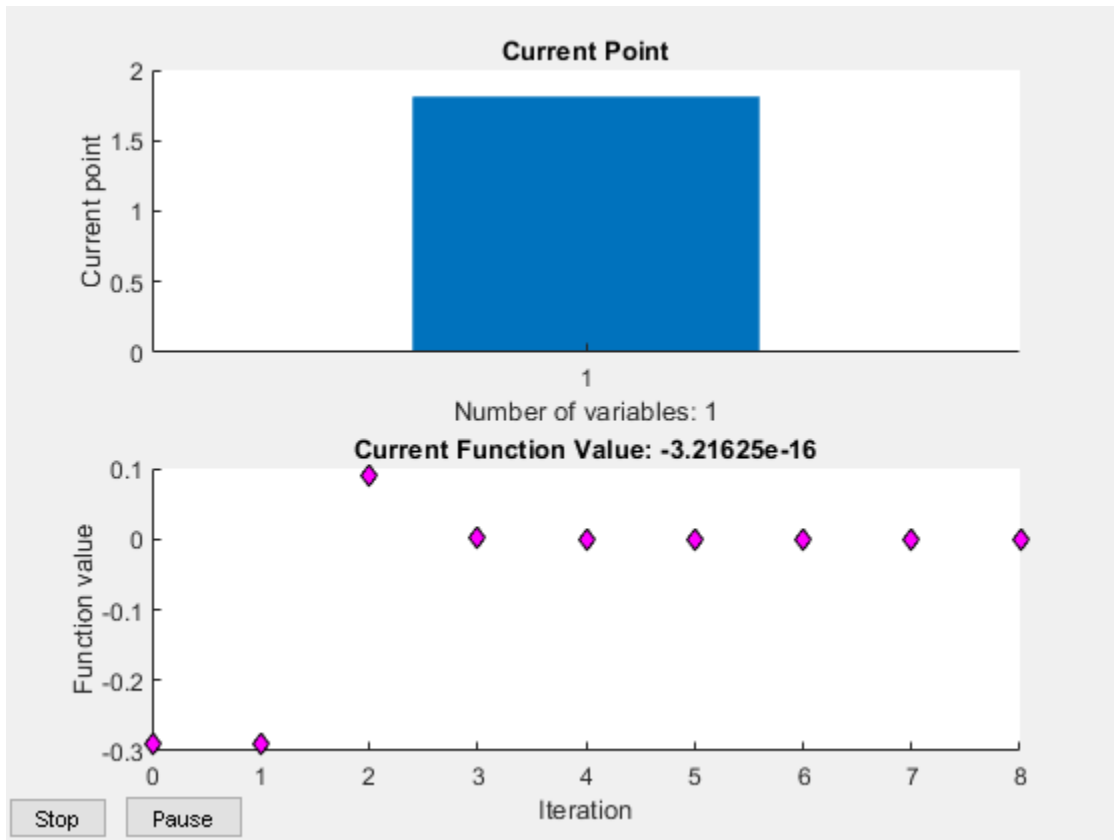
Examine the solution process by setting options that include plot functions.

```
options = optimset('PlotFcns',{@optimplotx,@optimplotfval});
```

Run `fzero` including options.

```
x = fzero(fun,x0,options)
```





$$x = 1.8115$$

### Solve Exported Problem

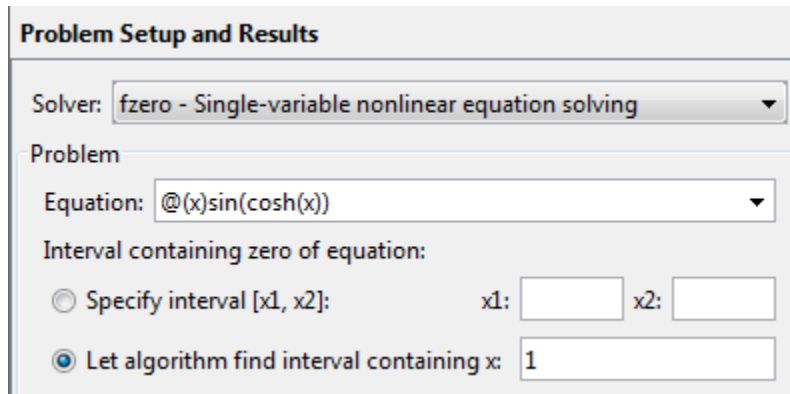
Solve a problem that is defined by an export from Optimization app.

Define a problem in Optimization app. Enter `optimtool('fzero')`, and fill in the problem as pictured.

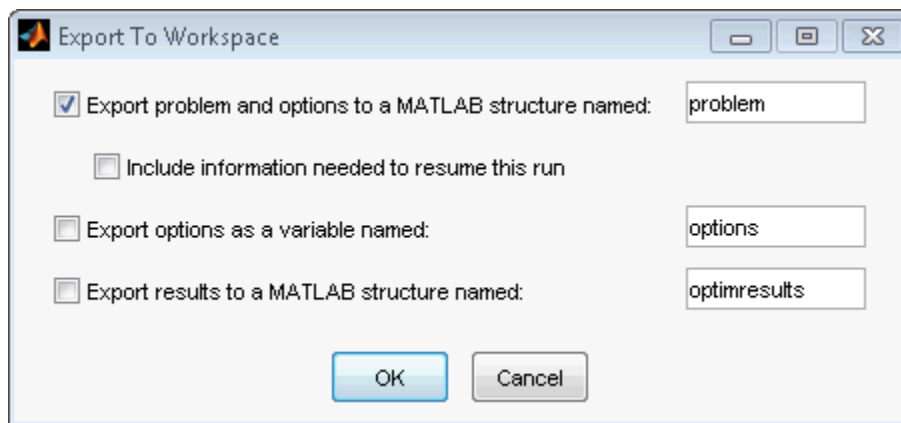
---

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

---



Select File > Export to Workspace, and export the problem as pictured to a variable named problem.



Enter the following at the command line.

```
x = fzero(problem)
```

```
x =
```

```
1.8115
```

## More Information from Solution

Find the point where  $\exp(-\exp(-x)) = x$ , and display information about the solution process.

```
fun = @(x) exp(-exp(-x)) - x; % function
x0 = [0 1]; % initial interval
options = optimset('Display','iter'); % show iterations
[x fval exitflag output] = fzero(fun,x0,options)
```

Func-count	x	f(x)	Procedure
2	1	-0.307799	initial
3	0.544459	0.0153522	interpolation
4	0.566101	0.00070708	interpolation
5	0.567143	-1.40255e-08	interpolation
6	0.567143	1.50013e-12	interpolation
7	0.567143	0	interpolation

Zero found in the interval [0, 1]

x = 0.5671

fval = 0

exitflag = 1

```
output = struct with fields:
    intervaliterations: 0
    iterations: 5
    funcCount: 7
    algorithm: 'bisection, interpolation'
    message: 'Zero found in the interval [0, 1]'
```

fval = 0 means  $\text{fun}(x) = 0$ , 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  $\text{fun}(x) = 0$ . To solve an equation  $\text{fun}(x) = c(x)$ , instead solve  $\text{fun2}(x) = \text{fun}(x) - c(x) = 0$ .

To include extra parameters in your function, see the example “Root of Function with Extra Parameter” on page 16-214 and the section “Passing Extra Parameters” on page 2-70.

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  $\pm\text{Inf}$ .

---

**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.

<code>Display</code>	Level of display (see “Iterative Display” on page 3-17): <ul style="list-style-type: none"> <li>• <code>'off'</code> displays no output.</li> <li>• <code>'iter'</code> displays output at each iteration.</li> <li>• <code>'final'</code> displays just the final output.</li> <li>• <code>'notify'</code> (default) displays output only if the function does not converge.</li> </ul>
<code>FunValCheck</code>	Check whether objective function values are valid. <ul style="list-style-type: none"> <li>• <code>'on'</code> displays an error when the objective function returns a value that is <code>complex</code>, <code>Inf</code>, or <code>NaN</code>.</li> <li>• The default, <code>'off'</code>, displays no error.</li> </ul>
<code>OutputFcn</code>	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 ( <code>[]</code> ). See “Output Function Syntax” on page 15-37.
<code>PlotFcns</code>	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 ( <code>[]</code> ). <ul style="list-style-type: none"> <li>• <code>@optimplotx</code> plots the current point.</li> <li>• <code>@optimplotfval</code> plots the function value.</li> </ul> <p>For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.</p>
<code>TolX</code>	Termination tolerance on <code>x</code> , a positive scalar. The default is <code>eps</code> , <code>2.2204e-16</code> . See “Tolerances and Stopping Criteria” on page 2-84.

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.

<code>objective</code>	Objective function
<code>x0</code>	Initial point for <code>x</code> , scalar or 2-D vector
<code>solver</code>	'fzero'
<code>options</code>	Options structure, typically created using <code>optimset</code>

You can generate `problem` by exporting from Optimization app. See “Importing and Exporting Your Work” on page 5-11 or “Solve Exported Problem” on page 16-215.

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 <code>x</code> .   |
| -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 | <code>fzero</code> 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:

<code>interval</code>	Number of iterations taken to find an interval containing a root
<code>s</code>	
<code>iterations</code>	Number of zero-finding iterations
<code>funcCount</code>	Number of function evaluations
<code>algorithm</code>	'bisection, interpolation'
<code>message</code>	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].

## 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

`fminbnd` | `fsolve` | `optimset` | `optimtool` | `roots`

### Topics

“Roots of Scalar Functions” (MATLAB)

“Passing Extra Parameters” on page 2-70

**Introduced before R2006a**



# infeasibility

**Package:** optim.problemdef

Constraint violation at a point

## Syntax

```
infeas = infeasibility(constr,pt)
```

## Description

`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

OptimizationConstraint object

Optimization constraint, specified as an `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 16-224.

## **See Also**

`OptimizationConstraint` | `evaluate`

## **Topics**

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

## intlinprog

Mixed-integer linear programming (MILP)

Mixed-integer linear programming solver.

Finds the minimum of a problem specified by

$$\min_x f^T x \text{ subject to } \begin{cases} x(\text{intcon}) \text{ are integers} \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub. \end{cases}$$

$f$ ,  $x$ ,  $\text{intcon}$ ,  $b$ ,  $beq$ ,  $lb$ , and  $ub$  are vectors, and  $A$  and  $Aeq$  are matrices.

You can specify  $f$ ,  $\text{intcon}$ ,  $lb$ , and  $ub$  as vectors or arrays. See “Matrix Arguments” on page 2-40.

---

**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.

---

## 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

$x = \text{intlinprog}(f, \text{intcon}, A, b)$  solves  $\min f^T x$  such that the components of  $x$  in  $\text{intcon}$  are integers, and  $A \cdot x \leq b$ .

`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  $lb \leq x \leq ub$ . 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  $A*x \leq b$  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 = 2x1
    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 explored	total time (s)	num int solution	integer fval	relative gap (%)
10000	0.80	0	-	-
18188	1.37	1	2.906000e+03	4.509804e+01
22039	1.68	2	2.073000e+03	2.270974e+01
24105	1.84	3	1.854000e+03	9.973046e+00
24531	1.88	3	1.854000e+03	1.347709e+00
24701	1.89	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 explored	total time (s)	num int solution	integer fval	relative gap (%)
3627	0.31	2	2.154000e+03	2.593968e+01
5844	0.46	3	1.854000e+03	1.180593e+01
6204	0.49	3	1.854000e+03	1.455526e+00
6400	0.50	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 = 3x1
```

```
    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 16-452.

### 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  $Ax \leq b$ .  $A$  has size  $M$ -by- $N$ , where  $M$  is the number of constraints and  $N = \text{numel}(f)$ . To save memory,  $A$  can be sparse.

Example:  $A = [4,3;2,0;4,-1];$  means three linear inequalities (three rows) for two decision variables (two columns).

Data Types: double

### **b** — Linear inequality constraint vector

real vector

Linear inequality constraint vector, specified as a vector of doubles.  $b$  represents the constant vector in the constraints  $A*x \leq b$ .  $b$  has length  $M$ , where  $A$  is  $M$ -by- $N$ .

Example: `[4,0]`

Data Types: `double`

### **Aeq — Linear equality constraint matrix**

`[]` (default) | real matrix

Linear equality constraint matrix, specified as a matrix of doubles.  $Aeq$  represents the linear coefficients in the constraints  $Aeq*x = beq$ .  $Aeq$  has size  $Meq$ -by- $N$ , where  $Meq$  is the number of constraints and  $N = \text{numel}(f)$ . 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  $lb \leq x \leq ub$ .

Internally, `intlinprog` converts an array  $lb$  to the vector  $lb(:)$ .

Example:  $lb = [0; -\text{Inf}; 4]$  means  $x(1) \geq 0$ ,  $x(3) \geq 4$ .

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  $lb \leq x \leq ub$ .

Internally, `intlinprog` converts an array `ub` to the vector `ub(:)`.

Example: `ub = [Inf;4;10]` means  $x(2) \leq 4$ ,  $x(3) \leq 10$ .

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 16-249.

`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 2-82.

Option	Description	Default
<i>AbsoluteGap Tolerance</i>	Nonnegative real. <code>intlinprog</code> stops if the difference between the internally calculated upper (U) and lower (L) bounds on the objective function is less than or equal to <code>AbsoluteGapTolerance</code> :  $U - L \leq \text{AbsoluteGapTolerance}$ .	0



Option	Description	Default
BranchRule	<p>Rule for choosing the component for branching:</p> <ul style="list-style-type: none"> <li>• 'maxpscost' — The fractional component with maximum pseudocost. See “Branch and Bound” on page 9-40.</li> <li>• 'strongpscost' — The fractional component with maximum pseudocost and a more accurate estimate of pseudocost than in 'maxpscost'. See “Branch and Bound” on page 9-40.</li> <li>• 'reliability' — The fractional component with maximum pseudocost and an even more accurate estimate of pseudocost than in 'strongpscost'. See “Branch and Bound” on page 9-40.</li> <li>• 'mostfractional' — The component whose fractional part is closest to 1/2.</li> <li>• 'maxfun' — The fractional component with a maximal corresponding component in the absolute value of the objective vector <math>f</math>.</li> </ul>	'reliability'
ConstraintTolerance	<p>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.</p>	1e-4
CutGeneration	<p>Level of cut generation (see “Cut Generation” on page 9-35):</p> <ul style="list-style-type: none"> <li>• 'none' — No cuts. Makes CutMaxIterations irrelevant.</li> <li>• 'basic' — Normal cut generation.</li> <li>• 'intermediate' — Use more cut types.</li> <li>• 'advanced' — Use most cut types.</li> </ul>	'basic'

Option	Description	Default
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 3-17): <ul style="list-style-type: none"> <li>• 'off' or 'none' — No iterative display</li> <li>• 'final' — Show final values only</li> <li>• 'iter' — Show iterative display</li> </ul>	'iter'
Heuristics	Algorithm for searching for feasible points (see “Heuristics for Finding Feasible Solutions” on page 9-36): <ul style="list-style-type: none"> <li>• 'basic'</li> <li>• 'intermediate'</li> <li>• 'advanced'</li> <li>• 'rss'</li> <li>• 'rins'</li> <li>• 'round'</li> <li>• 'diving'</li> <li>• 'rss-diving'</li> <li>• 'rins-diving'</li> <li>• 'round-diving'</li> <li>• 'none'</li> </ul>	'basic'
HeuristicsMaxNodes	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 9-36.	50

Option	Description	Default
IntegerPreprocess	Types of integer preprocessing (see “Mixed-Integer Program Preprocessing” on page 9-35): <ul style="list-style-type: none"> <li>• 'none' — Use very few integer preprocessing steps.</li> <li>• 'basic' — Use a moderate number of integer preprocessing steps.</li> <li>• 'advanced' — Use all available integer preprocessing steps.</li> </ul>	'basic'
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. <code>IntegerTolerance</code> is not a stopping criterion.	1e-5
LPMaxIterations	Strictly positive integer, the maximum number of simplex algorithm iterations per node during the branch-and-bound process.	$\max(3e4, 10 * (\text{numberOfEqualities} + \text{numberOfInequalities} + \text{numberOfVariables}))$ <p>In this expression, <code>numberOfEqualities</code> means the number of rows of <code>Aeq</code>, <code>numberOfInequalities</code> means the number of rows of <code>A</code>, and <code>numberOfVariables</code> means the number of elements of <code>f</code>.</p>

Option	Description	Default
<code>LPoptimalityTolerance</code>	Nonnegative real where reduced costs must exceed <code>LPoptimalityTolerance</code> for a variable to be taken into the basis.	<code>1e-7</code>
<code>LPPreprocess</code>	Type of preprocessing for the solution to the relaxed linear program (see “Linear Program Preprocessing” on page 9-34): <ul style="list-style-type: none"> <li>• <code>'none'</code> — No preprocessing.</li> <li>• <code>'basic'</code> — Use preprocessing.</li> </ul>	<code>'basic'</code>
<code>MaxNodes</code>	Strictly positive integer that is the maximum number of nodes <code>intlinprog</code> explores in its branch-and-bound process.	<code>1e7</code>
<code>MaxFeasiblePoints</code>	Strictly positive integer. <code>intlinprog</code> stops if it finds <code>MaxFeasiblePoints</code> integer feasible points.	<code>Inf</code>
<code>MaxTime</code>	Positive real that is the maximum time in seconds that <code>intlinprog</code> runs.	<code>7200</code>
<code>NodeSelection</code>	Choose the node to explore next. <ul style="list-style-type: none"> <li>• <code>'simplebestproj'</code> — Best projection. See “Branch and Bound” on page 9-40.</li> <li>• <code>'minobj'</code> — Explore the node with the minimum objective function.</li> <li>• <code>'mininfeas'</code> — Explore the node with the minimal sum of integer infeasibilities. See “Branch and Bound” on page 9-40.</li> </ul>	<code>'simplebestproj'</code>
<code>ObjectiveCutOff</code>	Real greater than <code>-Inf</code> . During the branch-and-bound calculation, <code>intlinprog</code> discards any node where the linear programming solution has an objective value exceeding <code>ObjectiveCutOff</code> .	<code>Inf</code>
<code>ObjectiveImprovementThreshold</code>	Nonnegative real. <code>intlinprog</code> changes the current feasible solution only when it locates another with an objective function value that is at least <code>ObjectiveImprovementThreshold</code> lower: $(\text{fold} - \text{fnew}) / (1 +  \text{fold} ) > \text{ObjectiveImprovementThreshold}$ .	<code>0</code>

Option	Description	Default
OutputFcn	<p>Specify one or more functions that an optimization function calls at events as 'savemilpsolutions', a function handle, or a cell array of function handles. For custom output functions, pass function handles.</p> <ul style="list-style-type: none"> <li>'savemilpsolutions' collects the integer-feasible points in the xIntSol matrix in your workspace, where each column is one integer feasible point.</li> </ul> <p>For information on writing a custom output function, see “intlinprog Output Function and Plot Function Syntax” on page 15-48.</p>	[]
PlotFcn	<p>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 ([]):</p> <ul style="list-style-type: none"> <li>'optimplotmilp' plots the internally-calculated upper and lower bounds on the objective value of the solution.</li> </ul> <p>For information on writing a custom plot function, see “intlinprog Output Function and Plot Function Syntax” on page 15-48.</p>	[]

Option	Description	Default
RelativeGapTolerance	<p>Real from 0 through 1. <code>intlinprog</code> 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 <code>RelativeGapTolerance</code>:</p> $(U - L) / (\text{abs}(U) + 1) \leq \text{RelativeGapTolerance}.$ <p><code>intlinprog</code> automatically modifies the tolerance for large L magnitudes:</p> $\text{tolerance} = \min(1/(1+ L ), \text{RelativeGapTolerance})$ <hr/> <p><b>Note</b> While you specify <code>RelativeGapTolerance</code> as a decimal number, the iterative display and output <code>.relativegap</code> report the gap in 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.</p>	1e-4
RootLPAlgorithm	<p>Algorithm for solving linear programs:</p> <ul style="list-style-type: none"> <li>• 'dual-simplex' — Dual simplex algorithm</li> <li>• 'primal-simplex' — Primal simplex algorithm</li> </ul>	'dual-simplex'

Option	Description	Default
RootLPMaxIterations	Nonnegative integer that is the maximum number of simplex algorithm iterations to solve the initial linear programming problem.	$\max(3e4, 10 * (\text{numberOfEqualities} + \text{numberOfInequalities} + \text{numberOfVariables}))$  In this expression, <code>numberOfEqualities</code> means the number of rows of <code>Aeq</code> , <code>numberOfInequalities</code> means the number of rows of <code>A</code> , and <code>numberOfVariables</code> means the number of elements of <code>f</code> .

Example: `options = optimoptions('intlinprog','MaxTime',120)`

### **problem** — Structure encapsulating inputs and options

structure

Structure encapsulating the inputs and options, specified with the following fields.

<code>f</code>	Vector representing objective $f' * x$ (required)
<code>intcon</code>	Vector indicating variables that take integer values (required)
<code>Aineq</code>	Matrix in linear inequality constraints $Aineq * x \leq bineq$
<code>bineq</code>	Vector in linear inequality constraints $Aineq * x \leq bineq$
<code>Aeq</code>	Matrix in linear equality constraints $Aeq * x = beq$
<code>beq</code>	Vector in linear equality constraints $Aeq * x = beq$
<code>lb</code>	Vector of lower bounds

<code>ub</code>	Vector of upper bounds
<code>x0</code>	Initial feasible point
<code>solver</code>	'intlinprog' (required)
<code>options</code>	Options created using <code>optimoptions</code> (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 <code>ConstraintTolerance</code> tolerance, but is not feasible with respect to the absolute tolerance.
2	<code>intlinprog</code> stopped prematurely. Integer feasible point found.
1	<code>intlinprog</code> converged to the solution <code>x</code> .
0	<code>intlinprog</code> stopped prematurely. No integer feasible point found.
-1	<code>intlinprog</code> 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.

<code>relativegap</code>	<p>Relative percentage difference between upper (U) and lower (L) bounds of the objective function that <code>intlinprog</code> calculates in its branch-and-bound algorithm.</p> $\text{relativegap} = 100 * (U - L) / (\text{abs}(U) + 1)$ <p>If <code>intcon = []</code>, <code>relativegap = []</code>.</p> <hr/> <p><b>Note</b> While you specify <code>RelativeGapTolerance</code> as a decimal number, the iterative display and <code>output.relativegap</code> report the gap in 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.</p> <hr/>
<code>absolutegap</code>	<p>Difference between upper and lower bounds of the objective function that <code>intlinprog</code> calculates in its branch-and-bound algorithm.</p> <p>If <code>intcon = []</code>, <code>absolutegap = []</code>.</p>
<code>numfeaspoints</code>	<p>Number of integer feasible points found.</p> <p>If <code>intcon = []</code>, <code>numfeaspoints = []</code>. Also, if the initial relaxed problem is infeasible, <code>numfeaspoints = []</code>.</p>
<code>numnodes</code>	<p>Number of nodes in branch-and-bound algorithm. If the solution was found during preprocessing or during the initial cuts, <code>numnodes = 0</code>.</p> <p>If <code>intcon = []</code>, <code>numnodes = []</code>.</p>
<code>constrviolation</code>	<p>Constraint violation that is positive for violated constraints.</p> $\text{constrviolation} = \max([0; \text{norm}(\text{Aeq} * x - \text{beq}, \text{inf}); (\text{lb} - x); (x - \text{ub}); (\text{Ai} * x - \text{bi})])$
<code>message</code>	<p>Exit message.</p>

## 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)
```

## 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

`linprog` | `mpsread` | `optimoptions` | `prob2struct`

## Topics

"Mixed-Integer Linear Programming Basics: Solver-Based" on page 9-48

"Factory, Warehouse, Sales Allocation Model: Solver-Based" on page 9-52

"Traveling Salesman Problem: Solver-Based" on page 9-64

"Solve Sudoku Puzzles Via Integer Programming: Solver-Based" on page 9-95

"Mixed-Integer Quadratic Programming Portfolio Optimization: Solver-Based" on page 9-86

"Optimal Dispatch of Power Generators: Solver-Based" on page 9-73

"Mixed-Integer Linear Programming Algorithms" on page 9-33

"Tuning Integer Linear Programming" on page 9-45

"Solver-Based Optimization Problem Setup"

## Introduced in R2014a

## linprog

Solve linear programming problems

Linear programming solver

Finds the minimum of a problem specified by

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

$f$ ,  $x$ ,  $b$ ,  $beq$ ,  $lb$ , and  $ub$  are vectors, and  $A$  and  $Aeq$  are matrices.

---

**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.

---

## 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

`x = linprog(f,A,b)` solves  $\min f' \cdot x$  such that  $A \cdot x \leq b$ .

`x = linprog(f,A,b,Aeq,beq)` includes equality constraints  $Aeq \cdot x = beq$ . Set  $A = []$  and  $b = []$  if no inequalities exist.

$x = \text{linprog}(f, A, b, Aeq, beq, lb, ub)$  defines a set of lower and upper bounds on the design variables,  $x$ , so that the solution is always in the range  $lb \leq x \leq ub$ . Set  $Aeq = []$  and  $beq = []$  if no equalities exist.

---

**Note** If the specified input bounds for a problem are inconsistent, the output `fval` is `[]`.

---

$x = \text{linprog}(f, A, b, Aeq, beq, lb, ub, options)$  minimizes with the optimization options specified by `options`. Use `optimoptions` to set these options.

$x = \text{linprog}(problem)$  finds the minimum for `problem`, where `problem` is a structure described in “Input Arguments” on page 16-265.

Create the `problem` structure by exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11. You can import a `problem` structure from an MPS file using `mpsread`. You can also create a `problem` structure from an `OptimizationProblem` object by using `prob2struct`.

$[x, fval] = \text{linprog}(\_\_\_)$ , for any input arguments, returns the value of the objective function `fun` at the solution  $x$ :  $fval = f' * x$ .

$[x, fval, exitflag, output] = \text{linprog}(\_\_\_)$  additionally returns a value `exitflag` that describes the exit condition, and a structure `output` that contains information about the optimization process.

$[x, fval, exitflag, output, lambda] = \text{linprog}(\_\_\_)$  additionally returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution  $x$ .

## Examples

### Linear Program, Linear Inequality Constraints

Solve a simple linear program defined by linear inequalities.

For this example, use these linear inequality constraints:

$$x(1) + x(2) \leq 2$$

$$x(1) + x(2)/4 \leq 1$$

$$x(1) - x(2) \leq 2$$

$$-x(1)/4 - x(2) \leq 1$$

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 2.$$

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1/4 \\ 1 & -1 \\ -1/4 & -1 \\ -1 & -1 \\ -1 & 1 \end{bmatrix};$$

$$b = [2 \ 1 \ 2 \ 1 \ -1 \ 2];$$

Use the objective function  $-x(1) - x(2)/3$ .

$$f = [-1 \ -1/3];$$

Solve the linear program.

$$x = \text{linprog}(f,A,b)$$

Optimal solution found.

$$x = 2 \times 1$$

$$\begin{array}{l} 0.6667 \\ 1.3333 \end{array}$$

### 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:

$$x(1) + x(2) \leq 2$$

$$x(1) + x(2)/4 \leq 1$$



$$x(1) - x(2) \leq 2$$

$$-x(1)/4 - x(2) \leq 1$$

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 2.$$

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1/4 \\ 1 & -1 \\ -1/4 & -1 \\ -1 & -1 \\ -1 & 1 \end{bmatrix};$$

$$b = [2 \ 1 \ 2 \ 1 \ -1 \ 2];$$

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.

$$x = \text{linprog}(f,A,b,Aeq,beq)$$

Optimal solution found.

$$x = 2 \times 1$$

$$\begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

### 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:

$$x(1) + x(2) \leq 2$$

$$x(1) + x(2)/4 \leq 1$$

$$x(1) - x(2) \leq 2$$

$$-x(1)/4 - x(2) \leq 1$$

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 2.$$

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1/4 \\ 1 & -1 \\ -1/4 & -1 \\ -1 & -1 \\ -1 & 1 \end{bmatrix};$$

$$b = [2 \ 1 \ 2 \ 1 \ -1 \ 2];$$

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  $-x(1) - x(2)/3$ .

$$f = [-1 \ -1/3];$$

Solve the linear program.

$$x = \text{linprog}(f, A, b, Aeq, beq, lb, ub)$$

Optimal solution found.

```
x = 2x1
    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:

$$x(1) + x(2) \leq 2$$

$$x(1) + x(2)/4 \leq 1$$

$$x(1) - x(2) \leq 2$$

$$-x(1)/4 - x(2) \leq 1$$

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 2.$$

```
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;
```

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

$$\max_x(x + y/3) \text{ subject to } \begin{cases} x + y \leq 2 \\ x + y/4 \leq 1 \\ x - y \leq 2 \\ x/4 + y \geq -1 \\ x + y \geq 1 \\ -x + y \leq 2 \\ x + y/4 = 1/2 \\ -1 \leq x \leq 1.5 \\ -1/2 \leq y \leq 1.25 \end{cases}$$

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 2-39.

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 16-452.

### 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:

$$x(1) + x(2) \leq 2$$

$$x(1) + x(2)/4 \leq 1$$

$$x(1) - x(2) \leq 2$$

$$-x(1)/4 - x(2) \leq 1$$

$$-x(1) - x(2) \leq -1$$

$$-x(1) + x(2) \leq 2.$$

```
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;
```

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 '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 16-260, 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

$$x_1 - x_2 + x_3 \leq 20$$

$$3x_1 + 2x_2 + 4x_3 \leq 42$$

$$3x_1 + 2x_2 \leq 30.$$

$$A = \begin{bmatrix} 1 & -1 & 1 \\ 3 & 2 & 4 \\ 3 & 2 & 0 \end{bmatrix};$$

$$b = [20; 42; 30];$$

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:

$A \cdot x$

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  $f' \cdot x$ . 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  $f(:)$ .

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

$A \cdot x \leq b$ ,

where  $x$  is the column vector of  $N$  variables  $x(:)$ , and  $b$  is a column vector with  $M$  elements.

For example, to specify

$$\begin{array}{rclcl} x_1 & + & 2x_2 & \leq & 10 \\ 3x_1 & + & 4x_2 & \leq & 20 \\ 5x_1 & + & 6x_2 & \leq & 30, \end{array}$$

give these constraints:

$$\begin{array}{l} A = [1,2;3,4;5,6]; \\ b = [10;20;30]; \end{array}$$

Example: To specify that the  $x$ -components add up to 1 or less, take  $A = \text{ones}(1,N)$  and  $b = 1$

Data Types: `double`

### **Aeq — Linear equality constraints**

real matrix

Linear equality constraints, specified as a real matrix. **Aeq** is an  $M_e$ -by- $N$  matrix, where  $M_e$  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  $M_e$  linear equalities

$$\text{Aeq} * x = \text{beq},$$

where  $x$  is the column vector of  $N$  variables  $x(:)$ , and **beq** is a column vector with  $M_e$  elements.

For example, to specify

$$\begin{array}{rclclcl} x_1 & + & 2x_2 & + & 3x_3 & = & 10 \\ 2x_1 & + & 4x_2 & + & x_3 & = & 20, \end{array}$$

give these constraints:

$$\begin{array}{l} \text{Aeq} = [1,2,3;2,4,1]; \\ \text{beq} = [10;20]; \end{array}$$

Example: To specify that the x-components sum to 1, take  $A_{eq} = \text{ones}(1, N)$  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

$$A*x \leq b,$$

where **x** is the column vector of N variables  $x(:)$ , and **A** is a matrix of size M-by-N.

For example, to specify

$$\begin{array}{rclcl} x_1 & + & 2x_2 & \leq & 10 \\ 3x_1 & + & 4x_2 & \leq & 20 \\ 5x_1 & + & 6x_2 & \leq & 30, \end{array}$$

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  $A = \text{ones}(1, N)$  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  $A_{eq}$  is a matrix of size  $M_{eq}$ -by- $N$ .

For example, to specify

$$\begin{array}{rcccccc} x_1 & + & 2x_2 & + & 3x_3 & = & 10 \\ 2x_1 & + & 4x_2 & + & x_3 & = & 20, \end{array}$$

enter these constraints:

```
Aeq = [1,2,3;2,4,1];  
beq = [10;20];
```

Example: To specify that the  $x$  components sum to 1, use  $A_{eq} = \text{ones}(1,N)$  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 that of  $lb$ , then  $lb$  specifies that

$$x(i) \geq lb(i) \quad \text{for all } i.$$

If  $\text{numel}(lb) < \text{numel}(f)$ , then  $lb$  specifies that

$$x(i) \geq lb(i) \quad \text{for } 1 \leq i \leq \text{numel}(lb).$$

In this case, solvers issue a warning.

Example: To specify that all  $x$ -components are positive,  $lb = \text{zeros}(\text{size}(f))$

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 that of  $ub$ , then  $ub$  specifies that

$$x(i) \leq ub(i) \quad \text{for all } i.$$

If  $\text{numel}(ub) < \text{numel}(f)$ , then  $ub$  specifies that

$$x(i) \leq ub(i) \quad \text{for} \quad 1 \leq i \leq \text{numel}(ub).$$

In this case, solvers issue a warning.

Example: To specify that all x-components are less than one, `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 15-8 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 2-82.

#### All Algorithms

<b>Algorithm</b>	Choose the optimization algorithm: <ul style="list-style-type: none"> <li>• 'dual-simplex' (default)</li> <li>• 'interior-point-legacy'</li> <li>• 'interior-point'</li> </ul> <p>For information on choosing the algorithm, see “Linear Programming Algorithms” on page 2-12.</p>
<i>Diagnostics</i>	Display diagnostic information about the function to be minimized or solved. Choose 'off' (default) or 'on'.
<b>Display</b>	Level of display (see “Iterative Display” on page 3-17): <ul style="list-style-type: none"> <li>• 'final' (default) displays just the final output.</li> <li>• 'off' or 'none' displays no output.</li> <li>• 'iter' displays output at each iteration.</li> </ul>

**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
- $10 * (\text{numberOfEqualities} + \text{numberOfInequalities} + \text{numberOfVariables})$  for the 'dual-simplex' algorithm

See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.

For `optimset`, the name is `MaxIter`. See “Current and Legacy Option Name Tables” on page 15-31.

**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 Name Tables” on page 15-31.

### **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 Name Tables” on page 15-31.

*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 Name Tables” on page 15-31.

**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
<b>f</b>	Linear objective function vector <b>f</b>
<b>Aineq</b>	Matrix for linear inequality constraints
<b>bineq</b>	Vector for linear inequality constraints
<b>Aeq</b>	Matrix for linear equality constraints
<b>beq</b>	Vector for linear equality constraints
<b>lb</b>	Vector of lower bounds
<b>ub</b>	Vector of upper bounds
<b>solver</b>	'linprog'
<b>options</b>	Options created with <b>optimoptions</b>

You must supply at least the **solver** field in the **problem** structure.

The simplest way to obtain a **problem** structure is to export the problem from the Optimization app.

Data Types: `struct`

## Output Arguments

### **x — Solution**

real vector | real array

Solution, returned as a real vector or real array. The size of `x` is the same as the size of `f`.

### **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 <code>ConstraintTolerance</code> tolerance, but is not feasible with respect to the absolute tolerance.
1	Function converged to a solution <code>x</code> .
0	Number of iterations exceeded <code>options.MaxIterations</code> or solution time in seconds exceeded <code>options.MaxTime</code> .
-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.

<code>iterations</code>	Number of iterations
<code>algorithm</code>	Optimization algorithm used
<code>cgiterations</code>	0 (interior-point algorithm only, included for backward compatibility)
<code>message</code>	Exit message
<code>constrviolation</code>	Maximum of constraint functions
<code>firstorderopt</code>	First-order optimality measure

### **Lambda — Lagrange multipliers at the solution**

structure

Lagrange multipliers at the solution, returned as a structure with these fields.

<code>lower</code>	Lower bounds corresponding to <code>lb</code>
<code>upper</code>	Upper bounds corresponding to <code>ub</code>
<code>ineqlin</code>	Linear inequalities corresponding to <code>A</code> and <code>b</code>
<code>eqlin</code>	Linear equalities corresponding to <code>Aeq</code> and <code>beq</code>

## **Algorithms**

### **Dual-Simplex Algorithm**

For a description, see “Dual-Simplex Algorithm” on page 9-11.

### **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 9-8.

The first stage of the algorithm might involve some preprocessing of the constraints (see “Interior-Point-Legacy Linear Programming” on page 9-8). Several conditions might cause

linprog to exit with an infeasibility message. In each case, linprog returns a negative exitflag, indicating to indicate failure.

- If a row of all zeros is detected in Aeq, but the corresponding element of beq is not zero, then the exit message is

Exiting due to infeasibility: An all-zero row in the constraint matrix does not have a zero in corresponding right-hand-side entry.

- If one of the elements of x is found not to be bounded below, then the exit message is

Exiting due to infeasibility: Objective f'\*x is unbounded below.

- If one of the rows of Aeq has only one nonzero element, then the associated value in x is called a *singleton* variable. In this case, the value of that component of x can be computed from Aeq and beq. If the value computed violates another constraint, then the exit message is

Exiting due to infeasibility: Singleton variables in equality constraints are not feasible.

- If the singleton variable can be solved for, but the solution violates the upper or lower bounds, then the exit message is

Exiting due to infeasibility: Singleton variables in the equality constraints are not within bounds.

---

**Note** The preprocessing steps are cumulative. For example, even if your constraint matrix does not have a row of all zeros to begin with, other preprocessing steps can cause such a row to occur.

---

When the preprocessing finishes, the iterative part of the algorithm begins until the stopping criteria are met. (For more information about residuals, the primal problem, the dual problem, and the related stopping criteria, see “Interior-Point-Legacy Linear Programming” on page 9-8.) If the residuals are growing instead of getting smaller, or the residuals are neither growing nor shrinking, one of the two following termination messages is displayed, respectively,

One or more of the residuals, duality gap, or total relative error has grown 100000 times greater than its minimum value so far:

OR

One or more of the residuals, duality gap, or total relative error has stalled:

After one of these messages is displayed, it is followed by one of the following messages indicating that the dual, the primal, or both appear to be infeasible.

- The dual appears to be infeasible (and the primal unbounded). (The primal residual  $<$  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  $>$   $\sqrt{\text{OptimalityTolerance}}$ . (The primal residual  $<$   $10 * \text{OptimalityTolerance}$ .)
- The primal appears to be infeasible (and the dual unbounded) since the primal residual  $>$   $\sqrt{\text{OptimalityTolerance}}$ . (The dual residual  $<$   $10 * \text{OptimalityTolerance}$ .)
- 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.

## References

- [1] Dantzig, G.B., A. Orden, and P. Wolfe. "Generalized Simplex Method for Minimizing a Linear Form Under Linear Inequality Restraints." *Pacific Journal Math.*, Vol. 5, 1955, pp. 183-195.
- [2] Mehrotra, S. "On the Implementation of a Primal-Dual Interior Point Method." *SIAM Journal on Optimization*, Vol. 2, 1992, pp. 575-601.
- [3] Zhang, Y., "Solving Large-Scale Linear Programs by Interior-Point Methods Under the MATLAB Environment." *Technical Report TR96-01*, Department of Mathematics

and Statistics, University of Maryland, Baltimore County, Baltimore, MD, July 1995.

## **See Also**

`intlinprog` | `mpsread` | `optimoptions` | `prob2struct` | `quadprog`

## **Topics**

“Set Up a Linear Program, Solver-Based” on page 1-23

“Typical Linear Programming Problem” on page 9-16

“Maximize Long-Term Investments Using Linear Programming: Problem-Based”

“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

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 2-40.

The `lsqcurvefit` function uses the same algorithm as `lsqnonlin`. `lsqcurvefit` simply provides a convenient interface for data-fitting problems.

Rather than compute the sum of squares, `lsqcurvefit` requires the user-defined function to compute the *vector*-valued function

$$F(x, xdata) = \begin{bmatrix} F(x, xdata(1)) \\ F(x, xdata(2)) \\ \vdots \\ F(x, xdata(k)) \end{bmatrix}.$$

## Syntax

```
x = lsqcurvefit(fun,x0,xdata,ydata)
x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub)
x = lsqcurvefit(fun,x0,xdata,ydata,lb,ub,options)
x = lsqcurvefit(problem)
[x,resnorm] = lsqcurvefit(____)
[x,resnorm,residual,exitflag,output] = lsqcurvefit(____)
```

```
[x, resnorm, residual, exitflag, output, lambda, jacobian] = lsqcurvefit(
___)
```

## Description

`x = lsqcurvefit(fun, x0, xdata, ydata)` starts at `x0` and finds coefficients `x` to best fit the nonlinear function `fun(x, xdata)` to the data `ydata` (in the least-squares sense). `ydata` must be the same size as the vector (or matrix) `F` returned by `fun`.

---

**Note** “Passing Extra Parameters” on page 2-70 explains how to pass extra parameters to the vector function `fun(x)`, if necessary.

---

`x = lsqcurvefit(fun, x0, xdata, ydata, lb, ub)` defines a set of lower and upper bounds on the design variables in `x`, so that the solution is always in the range  $lb \leq x \leq ub$ . You can fix the solution component `x(i)` by specifying `lb(i) = ub(i)`.

---

**Note** If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the outputs `resnorm` and `residual` are `[]`.

---

Components of `x0` that violate the bounds  $lb \leq x \leq ub$  are reset to the interior of the box defined by the bounds. Components that respect the bounds are not changed.

---

`x = lsqcurvefit(fun, x0, xdata, ydata, lb, ub, options)` minimizes with the optimization options specified in `options`. Use `optimoptions` to set these options. Pass empty matrices for `lb` and `ub` if no bounds exist.

`x = lsqcurvefit(problem)` finds the minimum for `problem`, where `problem` is a structure described in “Input Arguments” on page 16-287. Create the `problem` structure by exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11.

`[x, resnorm] = lsqcurvefit( ___ )`, for any input arguments, returns the value of the squared 2-norm of the residual at `x`: `sum((fun(x, xdata) - ydata).^2)`.

`[x, resnorm, residual, exitflag, output] = lsqcurvefit( ___ )` additionally returns the value of the residual `fun(x, xdata) - ydata` at the solution `x`, a value



`exitflag` that describes the exit condition, and a structure `output` that contains information about the optimization process.

`[x, resnorm, residual, exitflag, output, lambda, jacobian] = lsqcurvefit(____)` additionally returns a structure `lambda` whose fields contain the Lagrange multipliers at the solution `x`, and the Jacobian of `fun` at the solution `x`.

## Examples

### Simple Exponential Fit

Suppose that you have observation time data `xdata` and observed response data `ydata`, and you want to find parameters `x(1)` and `x(2)` to fit a model of the form

$$ydata = x(1)\exp(x(2)xdata).$$

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)
```

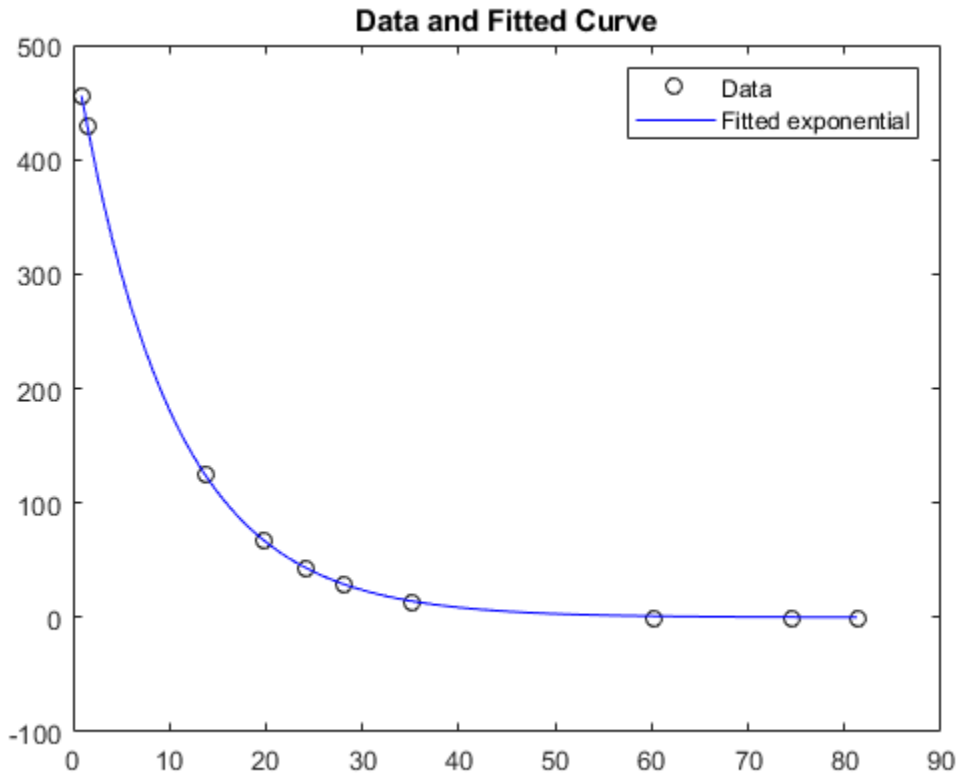
Local minimum possible.

`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  $\varepsilon$  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  $y = x(1)\exp(x(2)xdata)$  that best fits the data, with the parameters bounded as follows:

$$0 \leq x(1) \leq 3/4$$

$$-2 \leq x(2) \leq -1.$$

```
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.
```

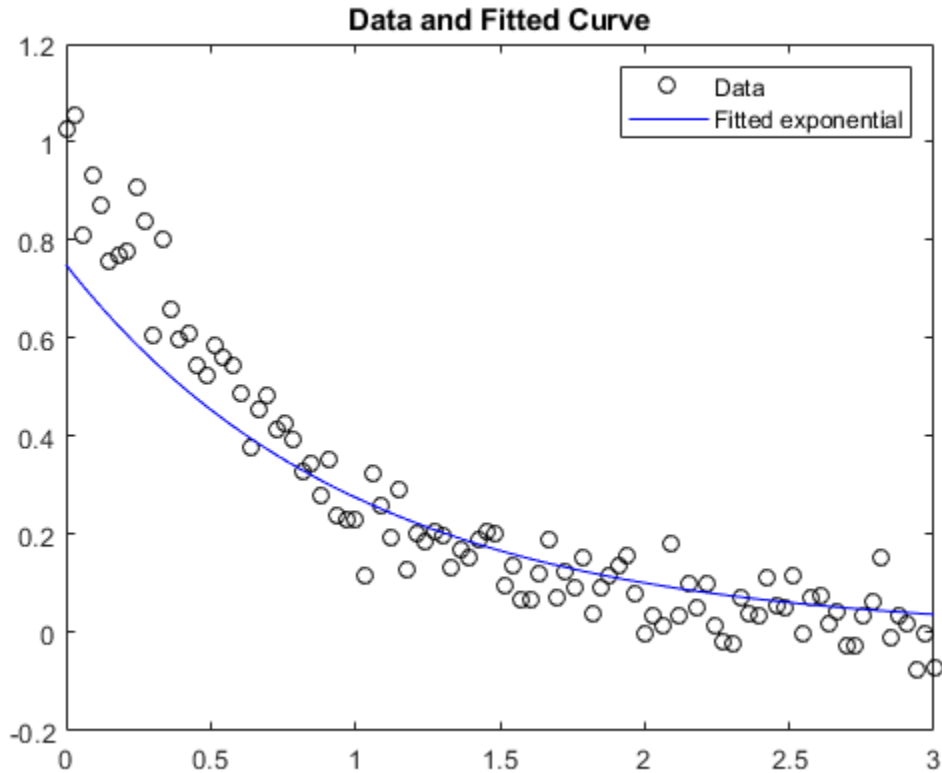
```
Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.
```

```
x = 1x2
```

```
    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  $x_0 = [100, -1]$ .

```
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 = 1x2
    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)
```

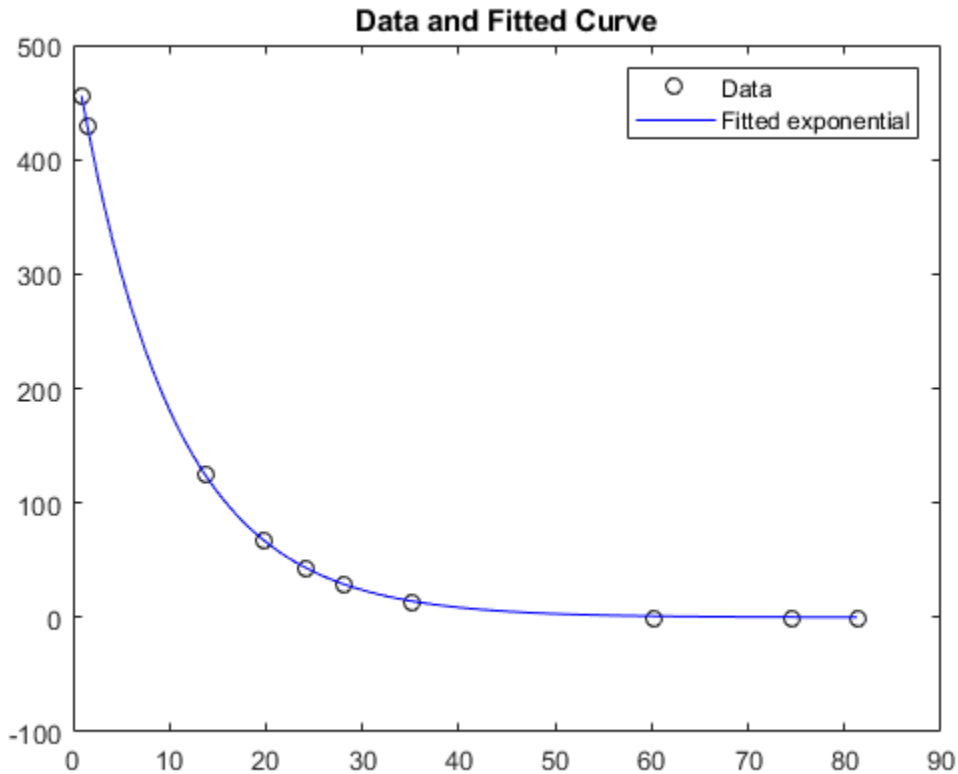
Local minimum possible.

lsqcurvefit stopped because the relative size of the current step is less than the value of the step size tolerance.

```
x = 1x2
    498.8309    -0.1013
```

The two algorithms converged to the same solution. Plot the data and the fitted exponential model.

```
times = linspace(xdata(1),xdata(end));  
plot(xdata,ydata,'ko',times,fun(x,times),'b-')  
legend('Data','Fitted exponential')  
title('Data and Fitted Curve')
```



### Compare Algorithms and Examine Solution Process

Compare the results of fitting with the default 'trust-region-reflective' algorithm and the 'levenberg-marquardt' algorithm. Examine the solution process to see which is more efficient in this case.

Suppose that you have observation time data `xdata` and observed response data `ydata`, and you want to find parameters  $x(1)$  and  $x(2)$  to fit a model of the form

$$ydata = x(1)\exp(x(2)xdata).$$

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  $x_0 = [100, -1]$ .

```
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 = 2.0632e-06
```

Yes, the solutions are equivalent.

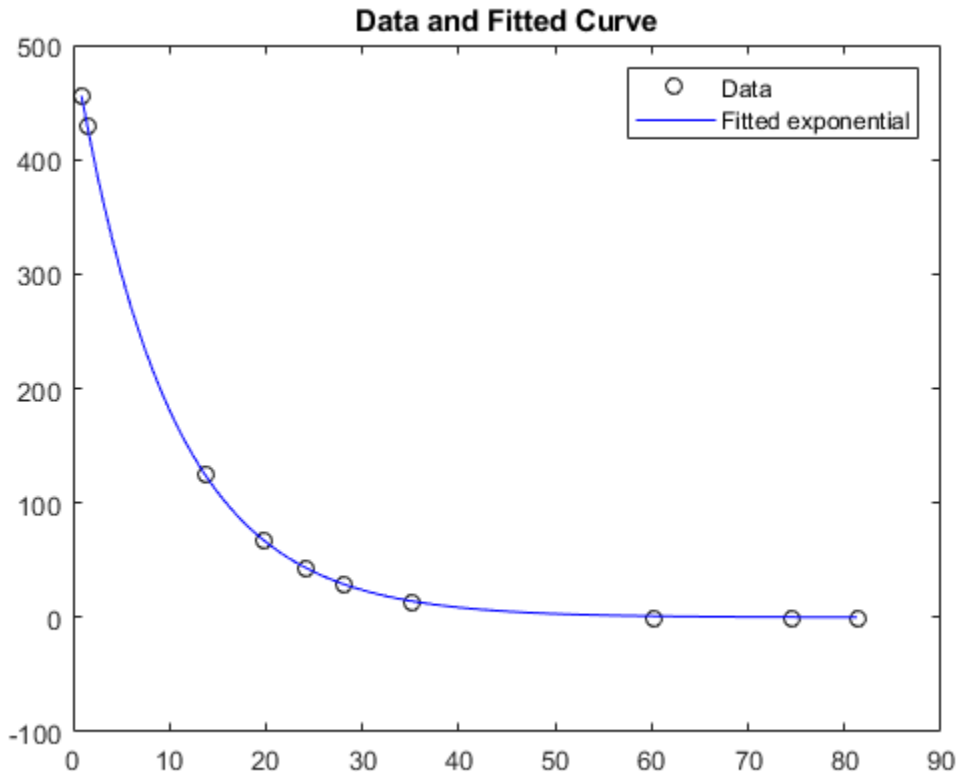
Which algorithm took fewer function evaluations to arrive at the solution?

```
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')
```



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
```

`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” (MATLAB)).

---

**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 Jacobian option is 'on', 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 2-34.

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

$$x(i) \geq lb(i) \quad \text{for all } i.$$

If `numel(lb) < numel(x0)`, then `lb` specifies that

$$x(i) \geq lb(i) \quad \text{for } 1 \leq i \leq \text{numel}(lb).$$

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

$$x(i) \leq ub(i) \quad \text{for all } i.$$

If `numel(ub) < numel(x0)`, then `ub` specifies that

$$x(i) \leq ub(i) \quad \text{for } 1 \leq i \leq \text{numel}(ub).$$

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 15-8 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 2-82.

#### **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  $x$ . The Levenberg-Marquardt algorithm does not handle bound constraints. For more information on choosing the algorithm, see “Choosing the Algorithm” on page 2-8.

*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 Name Tables” on page 15-31.

*Diagnostics*

Display diagnostic information about the function to be minimized or solved. Choices are 'off' (default) or 'on'.

<i>DiffMaxChange</i>	Maximum change in variables for finite-difference gradients (a positive scalar). The default is Inf.
<i>DiffMinChange</i>	Minimum change in variables for finite-difference gradients (a positive scalar). The default is 0.
<b>Display</b>	Level of display (see “Iterative Display” on page 3-17): <ul style="list-style-type: none"> <li>• 'off' or 'none' displays no output.</li> <li>• 'iter' displays output at each iteration, and gives the default exit message.</li> <li>• 'iter-detailed' displays output at each iteration, and gives the technical exit message.</li> <li>• 'final' (default) displays just the final output, and gives the default exit message.</li> <li>• 'final-detailed' displays just the final output, and gives the technical exit message.</li> </ul>
<b>FiniteDifferenceStepSize</b>	Scalar or vector step size factor for finite differences. When you set <code>FiniteDifferenceStepSize</code> to a vector <code>v</code> , the forward finite differences <code>delta</code> are $\text{delta} = v \cdot \text{sign}'(x) \cdot \max(\text{abs}(x), \text{TypicalX});$ where $\text{sign}'(x) = \text{sign}(x)$ except $\text{sign}'(0) = 1$ . Central finite differences are $\text{delta} = v \cdot \max(\text{abs}(x), \text{TypicalX});$ Scalar <code>FiniteDifferenceStepSize</code> expands to a vector. The default is <code>sqrt(eps)</code> for forward finite differences, and <code>eps^(1/3)</code> for central finite differences. <p>For <code>optimset</code>, the name is <code>FinDiffRelStep</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>

<code>FiniteDifferenceType</code>	<p>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.</p> <p>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.</p> <p>For <code>optimset</code>, the name is <code>FinDiffType</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>FunctionTolerance</code>	<p>Termination tolerance on the function value, a positive scalar. The default is <code>1e-6</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolFun</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>FunValCheck</code>	<p>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.</p>
<code>MaxFunctionEvaluations</code>	<p>Maximum number of function evaluations allowed, a positive integer. The default is <code>100*numberOfVariables</code>. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxFunEvals</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>MaxIterations</code>	<p>Maximum number of iterations allowed, a positive integer. The default is <code>400</code>. See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.</p> <p>For <code>optimset</code>, the name is <code>MaxIter</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>

**OptimalityTolerance** Termination tolerance on the first-order optimality (a positive scalar). The default is  $1e-6$ . See “First-Order Optimality Measure” on page 3-12.

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 Name Tables” on page 15-31.

**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 Syntax” on page 15-37.

**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.

For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.

For **optimset**, the name is **PlotFcns**. See “Current and Legacy Option Name Tables” on page 15-31.

SpecifyObjectiveGradient	<p>If <code>false</code> (default), the solver approximates the Jacobian using finite differences. If <code>true</code>, the solver uses a user-defined Jacobian (defined in <code>fun</code>), or Jacobian information (when using <code>JacobMult</code>), for the objective function.</p> <p>For <code>optimset</code>, the name is <code>Jacobian</code>, and the values are <code>'on'</code> or <code>'off'</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
StepTolerance	<p>Termination tolerance on <code>x</code>, a positive scalar. The default is <code>1e-6</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolX</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
TypicalX	<p>Typical <code>x</code> values. The number of elements in <code>TypicalX</code> is equal to the number of elements in <code>x0</code>, the starting point. The default value is <code>ones(numberofvariables,1)</code>. The solver uses <code>TypicalX</code> for scaling finite differences for gradient estimation.</p>
UseParallel	<p>When <code>true</code>, the solver estimates gradients in parallel. Disable by setting to the default, <code>false</code>. See “Parallel Computing”.</p>
<b>Trust-Region-Reflective Algorithm</b>	



**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

$$W = \text{jmfun}(\text{Jinfo}, Y, \text{flag})$$

where  $\text{Jinfo}$  contains the matrix used to compute  $J*Y$  (or  $J'*Y$ , or  $J'*(J*Y)$ ). The first argument  $\text{Jinfo}$  must be the same as the second argument returned by the objective function  $\text{fun}$ , for example, by

$$[F, \text{Jinfo}] = \text{fun}(x)$$

$Y$  is a matrix that has the same number of rows as there are dimensions in the problem.  $\text{flag}$  determines which product to compute:

- If  $\text{flag} == 0$  then  $W = J'*(J*Y)$ .
- If  $\text{flag} > 0$  then  $W = J*Y$ .
- If  $\text{flag} < 0$  then  $W = J'*Y$ .

In each case,  $J$  is not formed explicitly. The solver uses  $\text{Jinfo}$  to compute the preconditioner. See “Passing Extra Parameters” on page 2-70 for information on how to supply values for any additional parameters  $\text{jmfun}$  needs.

---

**Note** 'SpecifyObjectiveGradient' must be set to true for the solver to pass  $\text{Jinfo}$  from  $\text{fun}$  to  $\text{jmfun}$ .

---

See “Minimization with Dense Structured Hessian, Linear Equalities” on page 6-110 and “Jacobian Multiply Function with Linear Least Squares” on page 12-37 for similar examples.

For  $\text{optimset}$ , the name is  $\text{JacobMult}$ . See “Current and Legacy Option Name Tables” on page 15-31.

- 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)`. 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**

<i>InitDamping</i>	Initial value of the Levenberg-Marquardt parameter, a positive scalar. Default is $1e-2$ . For details, see “Levenberg-Marquardt Method” on page 12-7.
<i>ScaleProblem</i>	'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
$xdata$	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 <code>optimoptions</code>

You must supply at least the `objective`,  `$x0$` , `solver`,  `$xdata$` ,  `$ydata$` , and `options` fields in the problem structure.

The simplest way of obtaining a problem structure is to export the problem from the Optimization app.

Data Types: `struct`

## Output Arguments

### $x$ — Solution

real vector | real array

Solution, returned as a real vector or real array. The size of `x` is the same as the size of `x0`. Typically, `x` is a local solution to the problem when `exitflag` is positive. For information on the quality of the solution, see “When the Solver Succeeds” on page 4-22.

**resnorm — 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 <code>x</code> .
2	Change in <code>x</code> was less than the specified tolerance.
3	Change in the residual was less than the specified tolerance.
4	Magnitude of search direction was smaller than the specified tolerance.
0	Number of iterations exceeded <code>options.MaxIterations</code> or number of function evaluations exceeded <code>options.MaxFunctionEvaluations</code> .
-1	Output function terminated the algorithm.
-2	Problem is infeasible: the bounds <code>lb</code> and <code>ub</code> are inconsistent.

**output — Information about the optimization process**

structure

Information about the optimization process, returned as a structure with fields:

---

<code>firstorderopt</code>	Measure of first-order optimality
<code>iterations</code>	Number of iterations taken
<code>funcCount</code>	The number of function evaluations
<code>cgiterations</code>	Total number of PCG iterations (trust-region-reflective algorithm only)
<code>stepsize</code>	Final displacement in $x$
<code>algorithm</code>	Optimization algorithm used
<code>message</code>	Exit message

### **Lambda — Lagrange multipliers at the solution**

structure

Lagrange multipliers at the solution, returned as a structure with fields:

---

<code>lower</code>	Lower bounds <code>lb</code>
<code>upper</code>	Upper bounds <code>ub</code>

---

### **Jacobian — Jacobian at the solution**

real matrix

Jacobian at the solution, returned as a real matrix. `jacobian(i, j)` is the partial derivative of `fun(i)` with respect to `x(j)` at the solution  $x$ .

## **Limitations**

- The Levenberg-Marquardt algorithm does not handle bound constraints.
- The trust-region-reflective algorithm does not solve underdetermined systems; it requires that the number of equations, i.e., the row dimension of  $F$ , be at least as great as the number of variables. In the underdetermined case, `lsqcurvefit` uses the Levenberg-Marquardt algorithm.

Since the trust-region-reflective algorithm does not handle underdetermined systems and the Levenberg-Marquardt does not handle bound constraints, problems that have both of these characteristics cannot be solved by `lsqcurvefit`.

- `lsqcurvefit` can solve complex-valued problems directly with the `levenberg-marquardt` algorithm. However, this algorithm does not accept bound constraints. For

a complex problem with bound constraints, split the variables into real and imaginary parts, and use the `trust-region-reflective` algorithm. See “Fit a Model to Complex-Valued Data” on page 12-62.

- The preconditioner computation used in the preconditioned conjugate gradient part of the trust-region-reflective method forms  $J^T J$  (where  $J$  is the Jacobian matrix) before computing the preconditioner. Therefore, a row of  $J$  with many nonzeros, which results in a nearly dense product  $J^T J$ , can lead to a costly solution process for large problems.
- If components of  $x$  have no upper (or lower) bounds, `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 12-7.

## 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. Hillstom. *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

### 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 14-6.

## See Also

`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

Linear least-squares solver with bounds or linear constraints.

Solves least-squares curve fitting problems of the form

$$\min_x \frac{1}{2} \|C \cdot x - d\|_2^2 \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub. \end{cases}$$

---

**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.

---

## Syntax

```
x = lsqlin(C,d,A,b)
x = lsqlin(C,d,A,b,Aeq,beq,lb,ub)
x = lsqlin(C,d,A,b,Aeq,beq,lb,ub,x0,options)
x = lsqlin(problem)
[x,resnorm,residual,exitflag,output,lambda] = lsqlin(____)
```

## Description

`x = lsqlin(C,d,A,b)` solves the linear system  $C \cdot x = d$  in the least-squares sense, subject to  $A \cdot x \leq b$ .

`x = lsqlin(C,d,A,b,Aeq,beq,lb,ub)` adds linear equality constraints  $Aeq \cdot x = beq$  and bounds  $lb \leq x \leq ub$ . If you do not need certain constraints such as  $Aeq$  and  $beq$ , set them to `[]`. If  $x(i)$  is unbounded below, set  $lb(i) = -Inf$ , and if  $x(i)$  is unbounded above, set  $ub(i) = Inf$ .

`x = lsqlin(C,d,A,b,Aeq,beq,lb,ub,x0,options)` minimizes with an initial point `x0` and the optimization options specified in `options`. Use `optimoptions` to set these options. If you do not want to include an initial point, set the `x0` argument to `[]`.

`x = lsqlin(problem)` finds the minimum for `problem`, where `problem` is a structure. Create the `problem` structure by exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11. 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  $\text{resnorm} = \|C \cdot x - d\|_2^2$
- The residual  $\text{residual} = C \cdot x - d$
- 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  $\frac{1}{2}$  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  $C \cdot x - d$  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	2.000616e-10	2.036997e-08	1.370933e-02
4	-3.912129e-01	1.000866e-13	1.006816e-08	2.548273e-03
5	-3.948062e-01	2.220446e-16	2.955102e-09	4.295807e-04
6	-3.953277e-01	2.775558e-17	1.237758e-09	3.102850e-05
7	-3.953581e-01	2.775558e-17	1.645862e-10	1.138719e-07
8	-3.953582e-01	2.775558e-17	2.399608e-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 = 4x1
```

```

    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: [3×1 double]  
    eqlin: [0×1 double]  
    lower: [4×1 double]  
    upper: [4×1 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 constraint matrix**

real matrix

Linear inequality constraint matrix, specified as a matrix of doubles. A represents the linear coefficients in the constraints  $A*x \leq b$ . A has size Mineq-by-N, where Mineq is the number of constraints and N is the number of elements of  $x$ . To save memory, pass A as a sparse matrix.

Example:  $A = [4,3;2,0;4,-1]$ ; means three linear inequalities (three rows) for two decision variables (two columns).

Data Types: double

### **b — Linear inequality constraint vector**

real vector

Linear inequality constraint vector, specified as a vector of doubles. b represents the constant vector in the constraints  $A*x \leq b$ . b has length Mineq, where A is Mineq-by-N.

Example:  $[4,0]$

Data Types: double

### **Aeq — Linear equality constraint matrix**

[ ] (default) | real matrix

Linear equality constraint matrix, specified as a matrix of doubles. **Aeq** represents the linear coefficients in the constraints  $\text{Aeq} * \mathbf{x} = \text{beq}$ . **Aeq** has size **Meq**-by-**N**, where **Meq** is the number of constraints and **N** is the number of elements of **x**. To save memory, pass **Aeq** as a sparse matrix.

Example:  $\text{A} = [4, 3; 2, 0; 4, -1]$ ; 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  $\text{Aeq} * \mathbf{x} = \text{beq}$ . **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  $\text{lb} \leq \mathbf{x} \leq \text{ub}$ .

Internally, `lsqlin` converts an array **lb** to the vector `lb(:)`.

Example:  $\text{lb} = [0; -\text{Inf}; 4]$  means  $x(1) \geq 0$ ,  $x(3) \geq 4$ .

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  $\text{lb} \leq \mathbf{x} \leq \text{ub}$ .

Internally, `lsqlin` converts an array **ub** to the vector `ub(:)`.

Example: `ub = [Inf;4;10]` means  $x(2) \leq 4$ ,  $x(3) \leq 10$ .

Data Types: `double`

### **`x0` — Initial point**

`[]` (default) | real vector or array

Initial point for the solution process, specified as a vector or array of doubles. `x0` is used only by the `'trust-region-reflective'` algorithm. Optional.

If you do not provide an `x0` for the `'trust-region-reflective'` algorithm, `lsqlin` sets `x0` to the zero vector. If any component of this zero vector `x0` violates the bounds, `lsqlin` sets `x0` to a point in the interior of the box defined by the bounds.

Example: `x0 = [4;-3]`

Data Types: `double`

### **`options` — Options for `lsqlin`**

options created using `optimoptions` or the Optimization app

Options for `lsqlin`, specified as the output of the `optimoptions` function or the Optimization app.

Some options are absent from the `optimoptions` display. These options appear in italics in the following table. For details, see “View Options” on page 2-82.

**All Algorithms****Algorithm**

Choose the algorithm:

- 'interior-point' (default)
- 'trust-region-reflective'

The 'trust-region-reflective' algorithm allows *only* upper and lower bounds, meaning no linear inequalities or equalities. If you specify both the 'trust-region-reflective' and linear constraints, `lsqin` uses the 'interior-point' algorithm.

The 'trust-region-reflective' algorithm does not allow equal upper and lower bounds.

For more information on choosing the algorithm, see “Choosing the Algorithm” on page 2-8.

**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 200.

For `optimset`, the name is `MaxIter`. See “Current and Legacy Option Name Tables” on page 15-31.

## trust-region-reflective Algorithm Options

**FunctionTolerance** Termination tolerance on the function value, a positive scalar. The default is  $100 \times \text{eps}$ , about  $2.2204e-14$ .

For `optimset`, the name is `TolFun`. See “Current and Legacy Option Name Tables” on page 15-31.

**JacobianMultiplyFcn** Jacobian multiply function, specified as a function handle. For large-scale structured problems, this function should compute the Jacobian matrix product  $C \times Y$ ,  $C' \times Y$ , or  $C' \times (C \times Y)$  without actually forming  $C$ . Write the function in the form

$$W = \text{jmfun}(\text{Jinfo}, Y, \text{flag})$$

where `Jinfo` contains a matrix used to compute  $C \times Y$  (or  $C' \times Y$ , or  $C' \times (C \times Y)$ ).

`jmfun` must compute one of three different products, depending on the value of `flag` that `lsqlin` passes:

- If `flag == 0` then  $W = C' \times (C \times Y)$ .
- If `flag > 0` then  $W = C \times Y$ .
- If `flag < 0` then  $W = C' \times Y$ .

In each case, `jmfun` need not form  $C$  explicitly. `lsqlin` uses `Jinfo` to compute the preconditioner. See “Passing Extra Parameters” on page 2-70 for information on how to supply extra parameters if necessary.

See “Jacobian Multiply Function with Linear Least Squares” on page 12-37 for an example.

For `optimset`, the name is `JacobMult`. See “Current and Legacy Option Name Tables” on page 15-31.

*MaxPCGIter*

Maximum number of PCG (preconditioned conjugate gradient) iterations, a positive scalar. The default is  $\max(1, \text{floor}(\text{numberOfVariables}/2))$ . For more information, see “Trust-Region-Reflective Algorithm” on page 16-321.

OptimalityTolerance	Termination tolerance on the first-order optimality, a positive scalar. The default is <code>100*eps</code> , about <code>2.2204e-14</code> . See “First-Order Optimality Measure” on page 3-12.
	For <code>optimset</code> , the name is <code>TolFun</code> . See “Current and Legacy Option Name Tables” on page 15-31.
<i>PrecondBandWidth</i>	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 <code>PrecondBandWidth</code> to <code>Inf</code> 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 16-321.
SubproblemAlgorithm	Determines how the iteration step is calculated. The default, <code>'cg'</code> , takes a faster but less accurate step than <code>'factorization'</code> . See “Trust-Region-Reflective Least Squares” on page 12-3.
<i>TolPCG</i>	Termination tolerance on the PCG (preconditioned conjugate gradient) iteration, a positive scalar. The default is <code>0.1</code> .
TypicalX	Typical $x$ values. The number of elements in <code>TypicalX</code> is equal to the number of variables. The default value is <code>ones(numberofvariables,1)</code> . <code>lsqlin</code> uses <code>TypicalX</code> internally for scaling. <code>TypicalX</code> has an effect only when $x$ has unbounded components, and when a <code>TypicalX</code> value for an unbounded component is larger than 1.

## interior-point Algorithm Options

<code>ConstraintTolerance</code>	Tolerance on the constraint violation, a positive scalar. The default is $1e-8$ .  For <code>optimset</code> , the name is <code>TolCon</code> . See “Current and Legacy Option Name Tables” on page 15-31.
<code>LinearSolver</code>	Type of internal linear solver in algorithm: <ul style="list-style-type: none"> <li>• 'auto' (default) — Use 'sparse' if the C matrix is sparse, 'dense' otherwise.</li> <li>• 'sparse' — Use sparse linear algebra. See “Sparse Matrices” (MATLAB).</li> <li>• 'dense' — Use dense linear algebra.</li> </ul>
<code>OptimalityTolerance</code>	Termination tolerance on the first-order optimality, a positive scalar. The default is $1e-8$ . See “First-Order Optimality Measure” on page 3-12.  For <code>optimset</code> , the name is <code>TolFun</code> . See “Current and Legacy Option Name Tables” on page 15-31.
<code>StepTolerance</code>	Termination tolerance on $x$ , a positive scalar. The default is $1e-12$ .  For <code>optimset</code> , the name is <code>TolX</code> . See “Current and Legacy Option Name Tables” on page 15-31.

## problem — Optimization problem

structure

Optimization problem, specified as a structure with the following fields.

<code>C</code>	Matrix multiplier in the term $C*x - d$
<code>d</code>	Additive constant in the term $C*x - d$
<code>Aineq</code>	Matrix for linear inequality constraints
<code>bineq</code>	Vector for linear inequality constraints
<code>Aeq</code>	Matrix for linear equality constraints
<code>beq</code>	Vector for linear equality constraints

<code>lb</code>	Vector of lower bounds
<code>ub</code>	Vector of upper bounds
<code>x0</code>	Initial point for <code>x</code>
<code>solver</code>	'lsqlin'
<code>options</code>	Options created with <code>optimoptions</code>

Create the `problem` structure by exporting a problem from the Optimization app, as described in “Exporting Your Work” on page 5-11.

Data Types: `struct`

## Output Arguments

### **`x` — Solution**

real vector

Solution, returned as a vector that minimizes the norm of  $C*x - d$  subject to all bounds and linear constraints.

### **`resnorm` — Objective value**

real scalar

Objective value, returned as the scalar value  $\text{norm}(C*x - d)^2$ .

### **`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 <code>options.FunctionTolerance</code> . (trust-region-reflective algorithm)
2	Step size smaller than <code>options.StepTolerance</code> , constraints satisfied. (interior-point algorithm)
1	Function converged to a solution <code>x</code> .
0	Number of iterations exceeded <code>options.MaxIterations</code> .
-2	The problem is infeasible. Or, for the interior-point algorithm, step size smaller than <code>options.StepTolerance</code> , 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 `lsqin` 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.

<code>iterations</code>	Number of iterations the solver took.
<code>algorithm</code>	One of these algorithms: <ul style="list-style-type: none"> <li>'interior-point'</li> <li>'trust-region-reflective'</li> </ul>
<code>constrviolation</code>	Constraint violation that is positive for violated constraints (not returned for the 'trust-region-reflective' algorithm). <p><code>constrviolation = max([0;norm(Aeq*x-beq, inf);(lb-x);(x-ub);(A*x-b)])</code></p>
<code>message</code>	Exit message.

<code>firstorderopt</code>	First-order optimality at the solution. See “First-Order Optimality Measure” on page 3-12.
<code>linearsolver</code>	Type of internal linear solver, 'dense' or 'sparse' ('interior-point' algorithm only)
<code>cgiterations</code>	Number of conjugate gradient iterations the solver performed. Nonempty only for the 'trust-region-reflective' algorithm.

See “Output Structures” on page 3-26.

### **lambda — Lagrange multipliers**

structure

Lagrange multipliers, returned as a structure with the following fields.

<code>lower</code>	Lower bounds <code>lb</code>
<code>upper</code>	Upper bounds <code>ub</code>
<code>ineqlin</code>	Linear inequalities
<code>eqlin</code>	Linear equalities

See “Lagrange Multiplier Structures” on page 3-27.

## **Tips**

- For problems with no constraints, you can use `mldivide` (matrix left division). When you have no constraints, `lsqlin` returns  $x = C \backslash d$ .
- Because the problem being solved is always convex, `lsqlin` finds a global, although not necessarily unique, solution.
- Better numerical results are likely if you specify equalities explicitly, using `Aeq` and `beq`, instead of implicitly, using `lb` and `ub`.
- The `trust-region-reflective` algorithm does not allow equal upper and lower bounds. Use another algorithm for this case.
- If the specified input bounds for a problem are inconsistent, the output `x` is `x0` and the outputs `resnorm` and `residual` are `[]`.
- You can solve some large structured problems, including those where the `C` matrix is too large to fit in memory, using the `trust-region-reflective` algorithm with a

Jacobian multiply function. For information, see trust-region-reflective Algorithm Options.

## Algorithms

### Trust-Region-Reflective Algorithm

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 “Interior-Point Linear Least Squares” on page 12-6.

## References

- [1] Coleman, T. F. and Y. Li. “A Reflective Newton Method for Minimizing a Quadratic Function Subject to Bounds on Some of the Variables,” *SIAM Journal on Optimization*, Vol. 6, Number 4, pp. 1040-1058, 1996.
- [2] Gill, P. E., W. Murray, and M. H. Wright. *Practical Optimization*, Academic Press, London, UK, 1981.

## See Also

lsqnonneg | mldivide | optimtool | quadprog

## Topics

- “Linear Least Squares with Bound Constraints” on page 12-31
- “Optimization App with the lsqlin Solver” on page 12-33
- “Jacobian Multiply Function with Linear Least Squares” on page 12-37
- “Least-Squares (Model Fitting) Algorithms” on page 12-2

**Introduced before R2006a**

## lsqnonlin

Solve nonlinear least-squares (nonlinear data-fitting) problems

Nonlinear least-squares solver

Solves nonlinear least-squares curve fitting problems of the form

$$\min_x \|f(x)\|_2^2 = \min_x (f_1(x)^2 + f_2(x)^2 + \dots + f_n(x)^2)$$

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 2-40.

Rather than compute the value  $\|f(x)\|_2^2$  (the sum of squares), `lsqnonlin` requires the user-defined function to compute the *vector*-valued function

$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix}.$$

## Syntax

```
x = lsqnonlin(fun,x0)
x = lsqnonlin(fun,x0,lb,ub)
x = lsqnonlin(fun,x0,lb,ub,options)
x = lsqnonlin(problem)
[x,resnorm] = lsqnonlin(____)
[x,resnorm,residual,exitflag,output] = lsqnonlin(____)
[x,resnorm,residual,exitflag,output,lambda,jacobian] = lsqnonlin(____)
)
```

## Description

$x = \text{lsqnonlin}(\text{fun}, x_0)$  starts at the point  $x_0$  and finds a minimum of the sum of squares of the functions described in  $\text{fun}$ . The function  $\text{fun}$  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  $\text{fun}(x)$ .)

---

**Note** “Passing Extra Parameters” on page 2-70 explains how to pass extra parameters to the vector function  $\text{fun}(x)$ , if necessary.

---

$x = \text{lsqnonlin}(\text{fun}, x_0, \text{lb}, \text{ub})$  defines a set of lower and upper bounds on the design variables in  $x$ , so that the solution is always in the range  $\text{lb} \leq x \leq \text{ub}$ . You can fix the solution component  $x(i)$  by specifying  $\text{lb}(i) = \text{ub}(i)$ .

---

**Note** If the specified input bounds for a problem are inconsistent, the output  $x$  is  $x_0$  and the outputs  $\text{resnorm}$  and  $\text{residual}$  are  $[\ ]$ .

---

Components of  $x_0$  that violate the bounds  $\text{lb} \leq x \leq \text{ub}$  are reset to the interior of the box defined by the bounds. Components that respect the bounds are not changed.

---

$x = \text{lsqnonlin}(\text{fun}, x_0, \text{lb}, \text{ub}, \text{options})$  minimizes with the optimization options specified in  $\text{options}$ . Use  $\text{optimoptions}$  to set these options. Pass empty matrices for  $\text{lb}$  and  $\text{ub}$  if no bounds exist.

$x = \text{lsqnonlin}(\text{problem})$  finds the minimum for  $\text{problem}$ , where  $\text{problem}$  is a structure described in “Input Arguments” on page 16-334. Create the  $\text{problem}$  structure by exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11.

$[x, \text{resnorm}] = \text{lsqnonlin}(\text{___})$ , for any input arguments, returns the value of the squared 2-norm of the residual at  $x$ :  $\text{sum}(\text{fun}(x).^2)$ .

$[x, \text{resnorm}, \text{residual}, \text{exitflag}, \text{output}] = \text{lsqnonlin}(\text{___})$  additionally returns the value of the residual  $\text{fun}(x)$  at the solution  $x$ , a value  $\text{exitflag}$  that describes the exit condition, and a structure  $\text{output}$  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  $\varepsilon$  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  $x_0 = 4$ .

```
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.

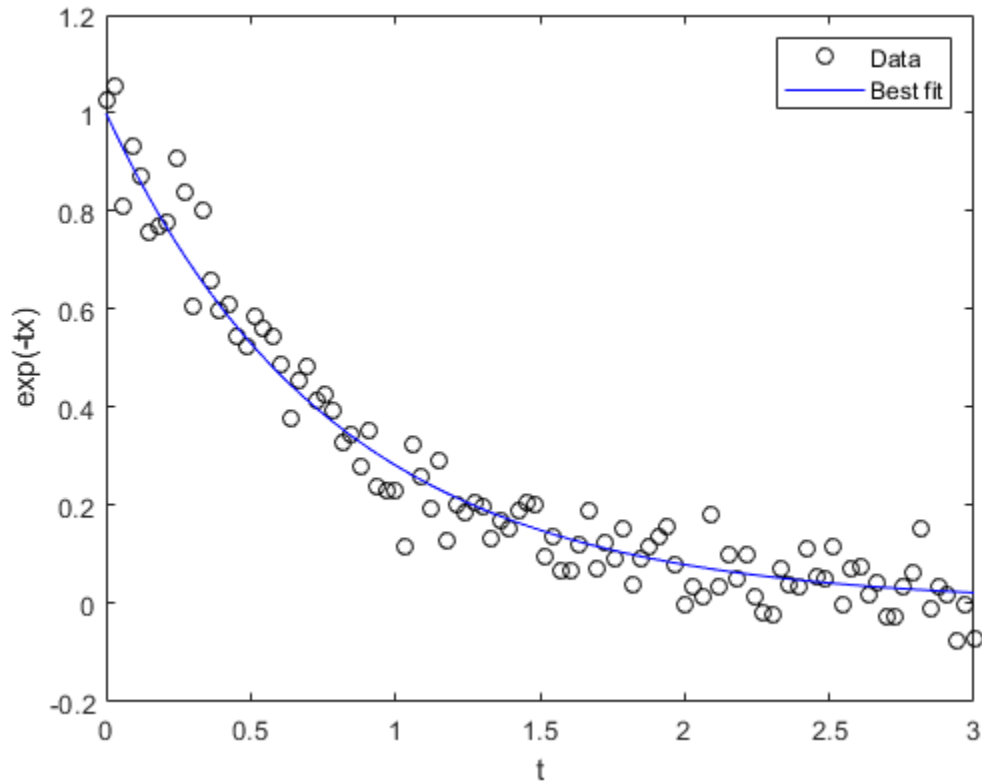
```
x = 1.2645
```

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)
```

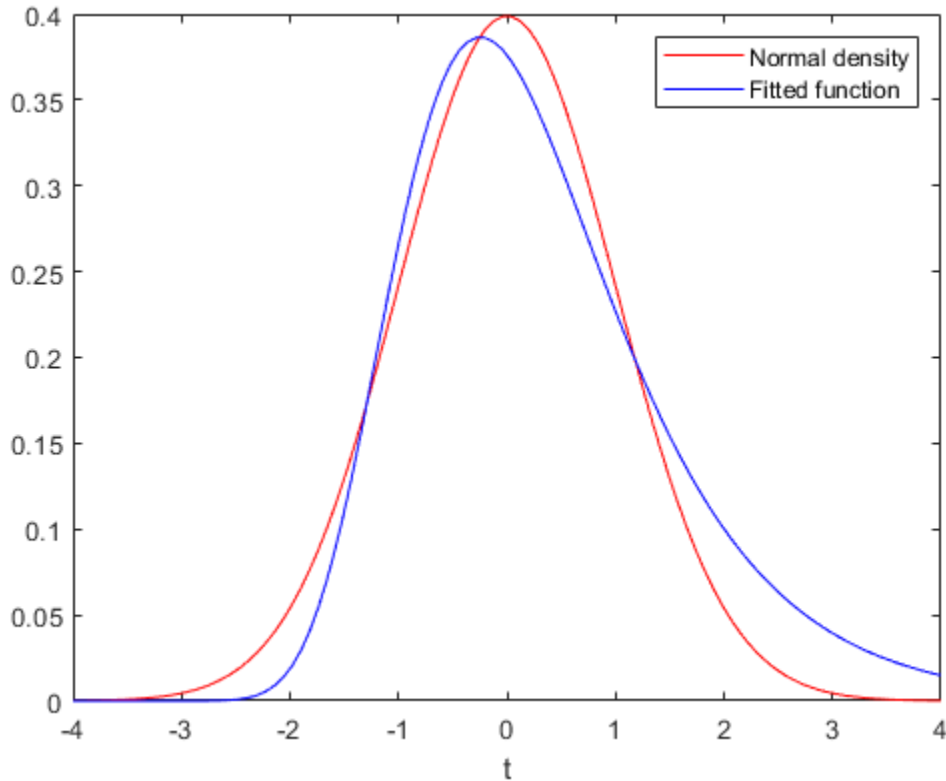
Local minimum possible.

`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  $x_0 = [100, -1]$ . 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.

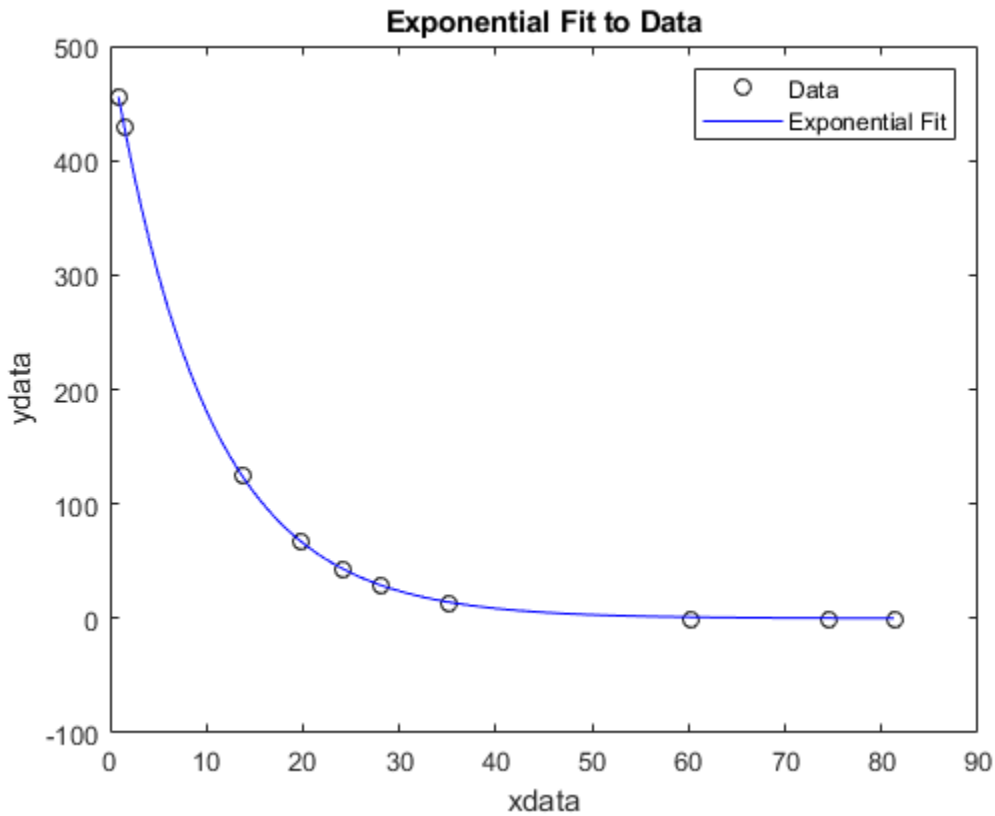
lsqnonlin stopped because the relative size of the current step is less than the value of the step size tolerance.

```
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
```



### Nonlinear Least Squares Solution and Residual Norm

Find the  $x$  that minimizes

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

and find the value of the minimal sum of squares.

Because `lsqnonlin` assumes that the sum of squares is *not* explicitly formed in the user-defined function, the function passed to `lsqnonlin` should instead compute the vector-valued function

$$F_k(x) = 2 + 2k - e^{kx_1} - e^{kx_2},$$

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

First, write a file to compute the 10-component vector  $F$ .

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

Find the minimizing point and the minimum value, starting at the point  $x_0 = [0.3, 0.4]$ .

```
x0 = [0.3, 0.4];
[x, resnorm] = lsqnonlin(@myfun, x0);
```

After about 24 function evaluations, this example gives the solution

```
x, resnorm
x =
    0.2578    0.2578

resnorm =
    124.3622
```

### Examine the Solution Process

Examine the solution process both as it occurs (by setting the `Display` option to `'iter'`) and 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  $x_0 = [100, -1]$ . 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);
```

Iteration	Func-count	f(x)	Norm of step	First-order 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	0.000462263	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);
```

Iteration	Func-count	Residual	First-Order optimality	Lambda	Norm of 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	0.00821621
10	45	243424	1.61e+04	10000	0.00777935
11	48	243322	8.8e+03	1000	0.0673933
12	51	242408	5.1e+03	100	0.675209
13	54	233628	1.05e+04	10	6.59804
14	57	169089	8.51e+04	1	54.6992
15	60	30814.7	1.54e+05	0.1	196.939
16	63	147.496	8e+03	0.01	129.795
17	66	9.51503	117	0.001	9.96069
18	69	9.50489	0.0714	0.0001	0.080486
19	72	9.50489	4.91e-05	1e-05	5.07033e-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: 4.9122e-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” (MATLAB)).

---

**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 Jacobian option is 'on', 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

$$x(i) \geq lb(i) \quad \text{for all } i.$$

If `numel(lb) < numel(x0)`, then `lb` specifies that

$$x(i) \geq lb(i) \quad \text{for } 1 \leq i \leq \text{numel}(lb).$$

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

$$x(i) \leq ub(i) \quad \text{for all } i.$$

If `numel(ub) < numel(x0)`, then `ub` specifies that

$$x(i) \leq ub(i) \quad \text{for } 1 \leq i \leq \text{numel}(ub).$$

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 15-8 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 2-82.

### All Algorithms

<code>Algorithm</code>	Choose between 'trust-region-reflective' (default) and 'levenberg-marquardt'.
	The <code>Algorithm</code> 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 <code>F</code> returned by <code>fun</code> ) must be at least as many as the length of <code>x</code> . The Levenberg-Marquardt algorithm does not handle bound constraints. For more information on choosing the algorithm, see “Choosing the Algorithm” on page 2-8.
<code>CheckGradients</code>	Compare user-supplied derivatives (gradients of objective or constraints) to finite-differencing derivatives. Choices are <code>false</code> (default) or <code>true</code> .
	For <code>optimset</code> , the name is <code>DerivativeCheck</code> and the values are 'on' or 'off'. See “Current and Legacy Option Name Tables” on page 15-31.
<i>Diagnositics</i>	Display diagnostic information about the function to be minimized or solved. Choices are 'off' (default) or 'on'.
<i>DiffMaxChange</i>	Maximum change in variables for finite-difference gradients (a positive scalar). The default is <code>Inf</code> .
<i>DiffMinChange</i>	Minimum change in variables for finite-difference gradients (a positive scalar). The default is <code>0</code> .

**Display**

Level of display (see “Iterative Display” on page 3-17):

- '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

$$\delta = v \cdot \text{sign}'(x) \cdot \max(\text{abs}(x), \text{TypicalX});$$

where  $\text{sign}'(x) = \text{sign}(x)$  except  $\text{sign}'(0) = 1$ . Central finite differences are

$$\delta = v \cdot \max(\text{abs}(x), \text{TypicalX});$$

Scalar `FiniteDifferenceStepSize` expands to a vector. The default is  $\sqrt{\text{eps}}$  for forward finite differences, and  $\text{eps}^{1/3}$  for central finite differences.

For `optimset`, the name is `FinDiffRelStep`. See “Current and Legacy Option Name Tables” on page 15-31.

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 <code>optimset</code> , the name is <code>FinDiffType</code> . See “Current and Legacy Option Name Tables” on page 15-31.
FunctionTolerance	Termination tolerance on the function value, a positive scalar. The default is <code>1e-6</code> . See “Tolerances and Stopping Criteria” on page 2-84.
	For <code>optimset</code> , the name is <code>TolFun</code> . See “Current and Legacy Option Name Tables” on page 15-31.
<i>FunValCheck</i>	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 <code>100*numberOfVariables</code> . See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.
	For <code>optimset</code> , the name is <code>MaxFunEvals</code> . See “Current and Legacy Option Name Tables” on page 15-31.
MaxIterations	Maximum number of iterations allowed, a positive integer. The default is <code>400</code> . See “Tolerances and Stopping Criteria” on page 2-84 and “Iterations and Function Counts” on page 3-10.
	For <code>optimset</code> , the name is <code>MaxIter</code> . See “Current and Legacy Option Name Tables” on page 15-31.

- OptimalityTolerance** Termination tolerance on the first-order optimality (a positive scalar). The default is  $1e-6$ . See “First-Order Optimality Measure” on page 3-12.
- 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 Name Tables” on page 15-31.
- 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 Syntax” on page 15-37.
- 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.
- For information on writing a custom plot function, see “Plot Function Syntax” on page 15-47.
- For **optimset**, the name is **PlotFcns**. See “Current and Legacy Option Name Tables” on page 15-31.

---

<code>SpecifyObjectiveGradient</code>	<p>If <code>false</code> (default), the solver approximates the Jacobian using finite differences. If <code>true</code>, the solver uses a user-defined Jacobian (defined in <code>fun</code>), or Jacobian information (when using <code>JacobMult</code>), for the objective function.</p> <p>For <code>optimset</code>, the name is <code>Jacobian</code>, and the values are <code>'on'</code> or <code>'off'</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>StepTolerance</code>	<p>Termination tolerance on <code>x</code>, a positive scalar. The default is <code>1e-6</code>. See “Tolerances and Stopping Criteria” on page 2-84.</p> <p>For <code>optimset</code>, the name is <code>TolX</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p>
<code>TypicalX</code>	<p>Typical <code>x</code> values. The number of elements in <code>TypicalX</code> is equal to the number of elements in <code>x0</code>, the starting point. The default value is <code>ones(numberofvariables,1)</code>. The solver uses <code>TypicalX</code> for scaling finite differences for gradient estimation.</p>
<code>UseParallel</code>	<p>When <code>true</code>, the solver estimates gradients in parallel. Disable by setting to the default, <code>false</code>. See “Parallel Computing”.</p>

### **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

$$W = \text{jmfun}(\text{Jinfo}, Y, \text{flag})$$

where  $\text{Jinfo}$  contains the matrix used to compute  $J*Y$  (or  $J'*Y$ , or  $J'*(J*Y)$ ). The first argument  $\text{Jinfo}$  must be the same as the second argument returned by the objective function  $\text{fun}$ , for example, by

$$[F, \text{Jinfo}] = \text{fun}(x)$$

$Y$  is a matrix that has the same number of rows as there are dimensions in the problem.  $\text{flag}$  determines which product to compute:

- If  $\text{flag} == 0$  then  $W = J'*(J*Y)$ .
- If  $\text{flag} > 0$  then  $W = J*Y$ .
- If  $\text{flag} < 0$  then  $W = J'*Y$ .

In each case,  $J$  is not formed explicitly. The solver uses  $\text{Jinfo}$  to compute the preconditioner. See “Passing Extra Parameters” on page 2-70 for information on how to supply values for any additional parameters  $\text{jmfun}$  needs.

---

**Note** 'SpecifyObjectiveGradient' must be set to true for the solver to pass  $\text{Jinfo}$  from  $\text{fun}$  to  $\text{jmfun}$ .

---

See “Minimization with Dense Structured Hessian, Linear Equalities” on page 6-110 and “Jacobian Multiply Function with Linear Least Squares” on page 12-37 for similar examples.

For  $\text{optimset}$ , the name is  $\text{JacobMult}$ . See “Current and Legacy Option Name Tables” on page 15-31.



<i>JacobPattern</i>	<p>Sparsity pattern of the Jacobian for finite differencing. Set <code>JacobPattern(i, j) = 1</code> when <code>fun(i)</code> depends on <code>x(j)</code>. Otherwise, set <code>JacobPattern(i, j) = 0</code>. In other words, <code>JacobPattern(i, j) = 1</code> when you can have <math>\partial \text{fun}(i) / \partial x(j) \neq 0</math>.</p> <p>Use <code>JacobPattern</code> when it is inconvenient to compute the Jacobian matrix <code>J</code> in <code>fun</code>, though you can determine (say, by inspection) when <code>fun(i)</code> depends on <code>x(j)</code>. The solver can approximate <code>J</code> via sparse finite differences when you give <code>JacobPattern</code>.</p> <p>If the structure is unknown, do not set <code>JacobPattern</code>. The default behavior is as if <code>JacobPattern</code> 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.</p>
<i>MaxPCGIter</i>	<p>Maximum number of PCG (preconditioned conjugate gradient) iterations, a positive scalar. The default is <code>max(1, numberOfVariables/2)</code>. For more information, see “Large Scale Nonlinear Least Squares” on page 12-5.</p>
<i>PrecondBandWidth</i>	<p>Upper bandwidth of preconditioner for PCG, a nonnegative integer. The default <code>PrecondBandWidth</code> is <code>Inf</code>, 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 <code>PrecondBandWidth</code> to <code>0</code> for diagonal preconditioning (upper bandwidth of 0). For some problems, an intermediate bandwidth reduces the number of PCG iterations.</p>
<i>SubproblemAlgorithm</i>	<p>Determines how the iteration step is calculated. The default, <code>'factorization'</code>, takes a slower but more accurate step than <code>'cg'</code>. See “Trust-Region-Reflective Least Squares” on page 12-3.</p>
<i>TolPCG</i>	<p>Termination tolerance on the PCG iteration, a positive scalar. The default is <code>0.1</code>.</p>

### **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 12-7.
- 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.

The simplest way of obtaining a problem structure is to export the problem from the Optimization app.

Data Types: struct

## Output Arguments

**x — Solution**

real vector | real array

Solution, returned as a real vector or real array. The size of x is the same as the size of x0. Typically, x is a local solution to the problem when exitflag is positive. For information on the quality of the solution, see “When the Solver Succeeds” on page 4-22.

**resnorm — 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 <code>x</code> .
2	Change in <code>x</code> was less than the specified tolerance.
3	Change in the residual was less than the specified tolerance.
4	Magnitude of search direction was smaller than the specified tolerance.
0	Number of iterations exceeded <code>options.MaxIterations</code> or number of function evaluations exceeded <code>options.MaxFunctionEvaluations</code> .
-1	Output function terminated the algorithm.
-2	Problem is infeasible: the bounds <code>lb</code> and <code>ub</code> are inconsistent.

**output — Information about the optimization process**

structure

Information about the optimization process, returned as a structure with fields:

<code>firstorderopt</code>	Measure of first-order optimality
<code>iterations</code>	Number of iterations taken

<code>funcCount</code>	The number of function evaluations
<code>cgiterations</code>	Total number of PCG iterations (trust-region-reflective algorithm only)
<code>stepsize</code>	Final displacement in $x$
<code>algorithm</code>	Optimization algorithm used
<code>message</code>	Exit message

### **Lambda — Lagrange multipliers at the solution**

structure

Lagrange multipliers at the solution, returned as a structure with fields:

---

<code>lower</code>	Lower bounds <code>lb</code>
<code>upper</code>	Upper bounds <code>ub</code>

---

### **Jacobian — Jacobian at the solution**

real matrix

Jacobian at the solution, returned as a real matrix. `jacobian(i,j)` is the partial derivative of `fun(i)` with respect to `x(j)` at the solution `x`.

## Limitations

- The Levenberg-Marquardt algorithm does not handle bound constraints.
- The trust-region-reflective algorithm does not solve underdetermined systems; it requires that the number of equations, i.e., the row dimension of  $F$ , be at least as great as the number of variables. In the underdetermined case, `lsqnonlin` uses the Levenberg-Marquardt algorithm.

Since the trust-region-reflective algorithm does not handle underdetermined systems and the Levenberg-Marquardt does not handle bound constraints, problems that have both of these characteristics cannot be solved by `lsqnonlin`.

- `lsqnonlin` can solve complex-valued problems directly with the `levenberg-marquardt` algorithm. However, this algorithm does not accept bound constraints. For a complex problem with bound constraints, split the variables into real and imaginary parts, and use the `trust-region-reflective` algorithm. See “Fit a Model to Complex-Valued Data” on page 12-62.

- The preconditioner computation used in the preconditioned conjugate gradient part of the trust-region-reflective method forms  $J^T J$  (where  $J$  is the Jacobian matrix) before computing the preconditioner. Therefore, a row of  $J$  with many nonzeros, which results in a nearly dense product  $J^T J$ , can lead to a costly solution process for large problems.
- If components of  $x$  have no upper (or lower) bounds, `lsqnonlin` prefers that the corresponding components of `ub` (or `lb`) be set to `inf` (or `-inf` for lower bounds) as opposed to an arbitrary but very large positive (or negative for lower bounds) number.

You can use the trust-region reflective algorithm in `lsqnonlin`, `lsqcurvefit`, and `fsolve` with small- to medium-scale problems without computing the Jacobian in `fun` or providing the Jacobian sparsity pattern. (This also applies to using `fmincon` or `fminunc` without computing the Hessian or supplying the Hessian sparsity pattern.) How small is small- to medium-scale? No absolute answer is available, as it depends on the amount of virtual memory in your computer system configuration.

Suppose your problem has  $m$  equations and  $n$  unknowns. If the command `J = sparse(ones(m,n))` causes an `Out of memory` error on your machine, then this is certainly too large a problem. If it does not result in an error, the problem might still be too large. You can find out only by running it and seeing if MATLAB runs within the amount of virtual memory available on your system.

## 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 12-7.

## 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. Hillstom. *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

### 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 14-6.

## See Also

`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

Solve nonnegative least-squares curve fitting problems of the form

$$\min_x \|C \cdot x - d\|_2^2, \text{ where } x \geq 0.$$

---

**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.

---

## 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

`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`, where `problem` is a structure. Create the `problem` argument by exporting a problem from Optimization app, as described in “Exporting Your Work” on page 5-11.

`[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  $x_{unc}$  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 = 2x1
    -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 15-8 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  $10*\text{eps}*\text{norm}(C,1)*\text{length}(C)$ . See “Tolerances and Stopping Criteria” on page 2-84.

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
<code>C</code>	Real matrix
<code>d</code>	Real vector
<code>solver</code>	'lsqnonneg'
<code>options</code>	Options structure such as returned by <code>optimset</code>

The simplest way to obtain a problem structure is to export the problem from the Optimization app.

Data Types: `struct`

## **Output Arguments**

### **x — Solution**

real vector

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  $\text{norm}(C*x - d)^2$ .

### **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 <code>options.MaxIter</code> .

### **output — Information about the optimization process**

structure

Information about the optimization process, returned as a structure with fields:

<code>iterations</code>	Number of iterations taken
<code>algorithm</code>	'active-set'
<code>message</code>	Exit message

### **lambda — Lagrange multipliers**

real vector

Lagrange multipliers, returned as a real vector. The entries satisfy the complementarity condition  $x'*\text{lambda} = 0$ . This means  $\text{lambda}(i) < 0$  when  $x(i)$  is approximately 0, and  $\text{lambda}(i)$  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_lsqli] = ...  
    lsqli(C,d,-eye(n,n),zeros(n,1));
```

The only difference is that the corresponding Lagrange multipliers have opposite signs:  $\lambda = -\lambda_{\text{lsqli}}$ .

## 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  $\lambda \leq 0$ .

## 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

lsqlin | mldivide | optimset | optimtool

**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`

### **lambda**in — 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

categorical variable

Reason the solver stopped, returned as a categorical variable. This table describes the exit flags for the `intlinprog` solver.

Exit Flag for <code>intlinprog</code>	Numeric Equivalent	Meaning
<code>OptimalWithPoorFeasibility</code>	3	The solution is feasible with respect to the relative <code>ConstraintTolerance</code> tolerance, but is not feasible with respect to the absolute tolerance.
<code>IntegerFeasible</code>	2	<code>intlinprog</code> stopped prematurely, and found an integer feasible point.
<code>OptimalSolution</code>	1	The solver converged to a solution $x$ .
<code>SolverLimitExceeded</code>	0	<code>intlinprog</code> exceeds one of the following tolerances: <ul style="list-style-type: none"> <li>• <code>LPMaxIterations</code></li> <li>• <code>MaxNodes</code></li> <li>• <code>MaxTime</code></li> <li>• <code>RootLPMaxIterations</code></li> </ul> See “Tolerances and Stopping Criteria” on page 2-84. <code>solve</code> also returns this exit flag when it runs out of memory at the root node.
<code>OutputFcnStop</code>	-1	<code>intlinprog</code> stopped by an output function or plot function.
<code>NoFeasiblePointFound</code>	-2	No feasible point found.
<code>Unbounded</code>	-3	The problem is unbounded.
<code>FeasibilityLost</code>	-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.
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)

Exit Flag for lsqlin	Numeric Equivalent	Meaning
StepSizeBelowTolerance	2	Step size smaller than <code>options.StepTolerance</code> , constraints satisfied. (interior-point algorithm)
OptimalSolution	1	The solver converged to a solution <code>x</code> .
SolverLimitExceeded	0	The number of iterations exceeds <code>options.MaxIterations</code> .
NoFeasiblePointFound	-2	The problem is infeasible. Or, for the interior-point algorithm, step size smaller than <code>options.StepTolerance</code> , but constraints are not satisfied.
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 <code>options.FunctionTolerance</code> . (trust-region-reflective algorithm)
StepSizeBelowTolerance	2	Step size smaller than <code>options.StepTolerance</code> , constraints satisfied. (interior-point-convex algorithm)
OptimalSolution	1	The solver converged to a solution <code>x</code> .



Exit Flag for quadprog	Numeric Equivalent	Meaning
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 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.

Exit Flag for <code>fminunc</code>	Numeric Equivalent	Meaning
<code>OutputFcnStop</code>	-1	Stopped by an output function or plot function.
<code>Unbounded</code>	-3	Objective function at current iteration is below <code>options.ObjectiveLimit</code> .

This table describes the exit flags for the `fmincon` solver.

Exit Flag for <code>fmincon</code>	Numeric Equivalent	Meaning
<code>NoDecreaseAlongSearchDirection</code>	5	Magnitude of directional derivative in search direction is less than <code>2*options.OptimalityTolerance</code> and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .
<code>SearchDirectionTooSmall</code>	4	Magnitude of the search direction is less than <code>2*options.StepTolerance</code> and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .
<code>FunctionChangeBelowTolerance</code>	3	Change in the objective function value is less than <code>options.FunctionTolerance</code> and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .
<code>StepSizeBelowTolerance</code>	2	Change in <code>x</code> is less than <code>options.StepTolerance</code> and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .
<code>OptimalSolution</code>	1	First-order optimality measure is less than <code>options.OptimalityTolerance</code> , and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .

Exit Flag for fmincon	Numeric Equivalent	Meaning
SolverLimitExceeded	0	Number of iterations exceeds <code>options.MaxIterations</code> or number of function evaluations exceeds <code>options.MaxFunctionEvaluations</code> .
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 <code>options.ObjectiveLimit</code> and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .

### 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:

- 'intlinprog' output
- 'linprog' output
- 'lsqlin' output
- 'lsqnonneg' output
- 'quadprog' output
- 'fminunc' output
- 'fmincon' 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. For the `intlinprog` solver, `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 10-28

“Include Derivatives in Problem-Based Workflow” on page 7-24

**Introduced in R2017b**

# mpsread

Read MPS file for LP and MILP optimization data

## Syntax

```
problem = mpsread(mpsfile)
```

## Description

`problem = mpsread(mpsfile)` reads data for linear programming (LP) and mixed-integer linear programming (MILP) problems. It returns the data in a structure that the `intlinprog` or `linprog` solvers accept.

## Examples

### Import and Run an MPS File

Load an mps file and solve the problem it describes.

Load the `eil33-2.mps` file from a public repository. View the problem type.

```
gunzip('http://miplib.zib.de/download/eil33-2.mps.gz')  
problem = mpsread('eil33-2.mps')
```

```
problem =
```

```
    f: [4516x1 double]  
  Aineq: [0x4516 double]  
  bineq: [0x1 double]  
   Aeq: [32x4516 double]  
  beq: [32x1 double]  
   lb: [4516x1 double]  
   ub: [4516x1 double]  
 intcon: [4516x1 double]
```

```
 solver: 'intlinprog'  
 options: [1x1 optim.options.Intlinprog]
```

Notice that `problem.intcon` is not empty, and `problem.solver` is 'intlinprog'. The problem is an integer linear programming problem.

Change the options to suppress iterative display and to generate a plot as the solver progresses.

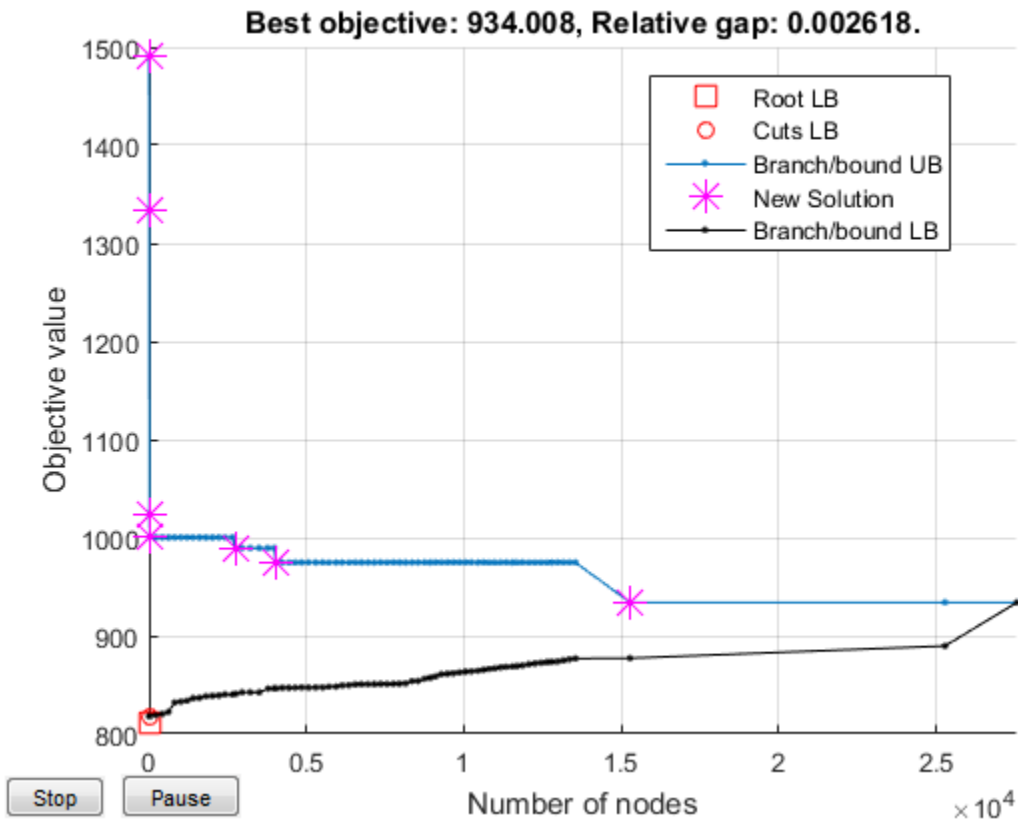
```
options = optimoptions('intlinprog','Display','final','PlotFcn',@optimplotmilp);  
problem.options = options;
```

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.RelativeGapTolerance = 0.0001` (the default value). The `intcon` variables integer within tolerance, `options.IntegerTolerance = 1e-05` (the default value).



## Input Arguments

**mpsfile** — Path to MPS file

character vector

Path to MPS file, specified as a character vector. `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

## Output Arguments

### **problem** — Problem structure

structure

Problem structure, returned as a structure with fields:

<code>f</code>	Vector representing objective $f'x$
<code>intcon</code>	Vector indicating variables that take integer values (empty for LP, nonempty for MILP)
<code>Aineq</code>	Matrix in linear inequality constraints $Aineq*x \leq bineq$
<code>bineq</code>	Vector in linear inequality constraints $Aineq*x \leq bineq$
<code>Aeq</code>	Matrix in linear equality constraints $Aeq*x = beq$
<code>beq</code>	Vector in linear equality constraints $Aeq*x = beq$
<code>lb</code>	Vector of lower bounds
<code>ub</code>	Vector of upper bounds
<code>solver</code>	'intlinprog' (if <code>intcon</code> is nonempty), or 'linprog' (if <code>intcon</code> is empty)
<code>options</code>	Default options, as returned by the command <code>optimoptions(solver)</code>

`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

`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 the cell array of character vectors or string vectors `cstr`.

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

showconstr(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`:

```

showconstr(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.

```
showconstr(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 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

arrays | string array

Names for indexing, specified as a cell array of character arrays or as a string array.

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 10-32.

## See Also

[OptimizationConstraint](#) | [OptimizationExpression](#) | [OptimizationProblem](#) | [OptimizationVariable](#) | [optimexpr](#)

## Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

# 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

`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 `nj` and names `cstrj`, creates an empty array of optimization expressions with dimensions equal to the integers `nj` or the lengths of the entries of `cstrj`.

## 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 showexpr.
```

### Create Optimization Expressions Indexed by Strings

Create a string array of color names, and an optimization expression that is indexed by the color names.

```
strexpr = ["red","green","blue","yellow"];  
expr = optimexpr(strexpr)
```

```
expr =  
  1x4 OptimizationExpression array with properties:  
  
  IndexNames: {{}} {1x4 cell}  
  Variables: [1x1 struct] containing 0 OptimizationVariables  
  
See expression formulation with showexpr.
```

You can use a cell array of character vectors instead of strings to get the same effect.

```
strexpr = {'red','green','blue','yellow'};  
expr = optimexpr(strexpr)
```

```
expr =  
  1x4 OptimizationExpression array with properties:  
  
  IndexNames: {{}} {1x4 cell}  
  Variables: [1x1 struct] containing 0 OptimizationVariables  
  
See expression formulation with showexpr.
```

If `strexpr` 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 showexpr.

```

### 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 showexpr.

```

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 showexpr.

```

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;  
showexpr(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 10-32.

## See Also

OptimizationExpression | optimconstr | showexpr | writeexpr

## Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

# OptimizationConstraint

Optimization constraints

## Description

An `OptimizationConstraint` object contains constraint expressions in terms of `OptimizationVariable` objects. Each constraint expression uses one of these comparison operators: `==`, `<=`, or `>=`.

A single expression can represent an array of constraints. For example, you can express the constraints that each row of a matrix variable `x` sums to one in this single expression.

```
constrsum = sum(x,2) == 1
```

## Creation

Create constraints using optimization expressions with one of these comparison operators: `==`, `<=`, or `>=`.

Include constraints in the `Constraints` property by using dot notation.

```
prob = optimproblem;  
x = optimvar(x,4,6);  
SumToOne = sum(x,2) == 1;  
prob.Constraints.SumToOne = SumToOne;
```

You can also create an empty optimization constraint by using `optimconstr`. Typically, you then fill the expression in a loop. For examples, see the `optimconstr` function reference page.

## Properties

### IndexNames — Index names

' ' (default) | cell array of strings | cell array of cells containing 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 10-22.

Data Types: `cell`

## Object Functions

`infeasibility`    Constraint violation at a point  
`showconstr`      Display optimization constraint  
`writeconstr`     Save optimization constraint description

## Examples

### Create Constraint Array

Create a 4-by-6 optimization variable matrix named `x`.

```
x = optimvar('x',4,6);
```

Create the constraints that each row of `x` sums to one.

```
constrsum = sum(x,2) == 1
```

```
constrsum =
```

```
4x1 Linear OptimizationConstraint array with properties:
```

```
  IndexNames: {} {}
```

```
  Variables: [1x1 struct] containing 1 OptimizationVariable
```

See constraint formulation with `showconstr`.

View the constraints.

```
showconstr(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 constraints in an optimization problem, use dot notation.

```
prob = optimproblem;  
prob.Constraints.constrsum = constrsum;
```

## See Also

`OptimizationExpression` | `OptimizationProblem` | `OptimizationVariable` | `infeasibility` | `optimconstr`

## Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

# OptimizationExpression

Objective function or constraints

## Description

An `OptimizationExpression` is an arithmetic expression in terms of optimization variables for an objective function or for comparison in constraints.

## 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 “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 16-397 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 16-396 and the `optimexpr` function reference page.

## Properties

### IndexNames — Index names

' ' (default) | cell array of strings | cell array of cells containing 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 10-22.

Data Types: `cell`

### Variables — Optimization variables in object

structure of `OptimizationVariable` objects

This property is read-only.

Optimization variables in the object, returned as a structure of `OptimizationVariable` objects.

Data Types: `struct`

## Object Functions

`evaluate` Evaluate optimization expression  
`showexpr` Display optimization expression  
`writelnexpr` Save optimization expression 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;  
showexpr(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 `showexpr`.

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 `showvar`.  
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 showexpr.
```

### 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
showexpr(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)x^2/10 + exp(-exp(-x));
fun = fcn2optimexpr(f,x)
```

```
fun =
  Nonlinear OptimizationExpression

  anonymousFunction1(x)
```

where:

```
anonymousFunction1 = @(x)x^2/10+exp(-exp(-x));
```

Find the point that minimizes fun starting from the point  $x_0 = 0$ .

```
x0 = struct('x',0);
prob = optimproblem('Objective',fun);
[sol,fval] = solve(prob,x0)
```

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:  
  x: -0.9595
```

```
fval = 0.1656
```

## Definitions

### Arithmetic Operations

For the list of supported operations on optimization expressions, see “Supported Operations on Optimization Variables and Expressions” on page 10-42.

### See Also

[OptimizationConstraint](#) | [OptimizationVariable](#) | [evaluate](#) | [fcn2optimexpr](#) | [optimexpr](#) | [showexpr](#) | [writeexpr](#)

### Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

“Optimization Expressions” on page 10-4

### Introduced in R2017b



# 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`.

## Creation

Create an `OptimizationProblem` object by using `optimproblem`.

## 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, returned 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 | structure containing `OptimizationConstraint` objects

Optimization constraints, specified as an `OptimizationConstraint` object or as a structure containing `OptimizationConstraint` 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

writeproblem Save optimization problem description  
 showproblem Display optimization problem  
 solve Solve optimization problem  
 prob2struct Convert optimization problem to solver form

## 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.

```
showproblem(prob)
```

```
OptimizationProblem :

max :
    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);
```

Optimal solution found.

```
sol.x
```

```
ans = 2×1
```

```
25.0000
```

```
15.0000
```

## See Also

[OptimizationConstraint](#) | [OptimizationExpression](#) | [OptimizationVariable](#) | [optimproblem](#) | [showproblem](#) | [solve](#) | [writeproblem](#)

## Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

# OptimizationVariable

Variable for optimization

## Description

An `OptimizationVariable` object contains variables for optimization expressions. These expressions represent the objective function or constraints. Variables are symbolic in nature, and can be arrays of any size.

## 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 `showbounds`, `showconstr`, `showexpr`, `showproblem`, or `showvar`. 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 cells containing 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 10-22.

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

<code>showbounds</code>	Display variable bounds
<code>showvar</code>	Display optimization variable
<code>writebounds</code>	Save description of variable bounds
<code>writevar</code>	Save optimization variable description

## 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 `showvar`.  
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 showvar.
  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 showvar.
```



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 showvar.
  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 showvar.  
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 showvar.  
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 `showvar`.  
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 `showvar`.  
See bounds with `showbounds`.

## Definitions

### Arithmetic Operations

For the list of supported operations on optimization variables, see “Supported Operations on Optimization Variables and Expressions” on page 10-42.

### Tips

- `OptimizationVariable` objects have *handle* copy behavior. See “Handle Object Behavior” (MATLAB) and “Comparison of Handle and Value Classes” (MATLAB). 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](#)

### Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

“Supported Operations on Optimization Variables and Expressions” on page 10-42

**Introduced in R2017b**

# optimoptions

Create optimization options

## Syntax

```
options = optimoptions(SolverName)
options = optimoptions(SolverName,Name,Value)

options = optimoptions(olddoptions,Name,Value)
options = optimoptions(SolverName,olddoptions)

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(olddoptions,Name,Value)` returns a copy of `olddoptions` with the named parameters altered with the specified values.

`options = optimoptions(SolverName,olddoptions)` returns default options for the `SolverName` solver, and copies the applicable options in `olddoptions` to `options`.

`options = optimoptions(prob)` returns a set of default options for the `prob` optimization 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-refle
```

```
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-r
```

```
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-r
```

```
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)'

    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: '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);
```

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	2.664535e-15	2.586177e-01
2	-3.405808e+00	0.000000e+00	1.776357e-15	2.244054e-03
3	-3.438788e+00	0.000000e+00	7.797582e-16	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 = 3x1
    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** — Optimization problem

`OptimizationProblem` object

Optimization problem, specified as an `OptimizationProblem` object. Create `prob` using the “Problem-Based Workflow” on page 10-2.

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 16-179
- `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 16-471
- `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

### App

You can set and modify options using the Optimization app (`optimtool`). See “Importing and Exporting Your Work” on page 5-11.

---

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

---

### See Also

`optimset` | `optimtool` | `resetoptions`

### Topics

“Set Options”

“Optimization App” on page 5-2

**Introduced in R2013a**



# optimproblem

Create optimization problem

## Syntax

```
prob = optimproblem
prob = optimproblem(Name, Value)
```

## Description

`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.

```
showproblem(prob)
```

```
OptimizationProblem :  
  
max :  
    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);
```

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 the `Description` property.

Example: `prob = optimproblem('Description','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 linear constraints. However, you can have a feasibility problem, which has no objective function, or you can have a problem with no linear constraints. Solve a complete problem by calling `solve`.

## See Also

OptimizationProblem | optimvar | solve

## Topics

“Problem-Based Optimization Setup”

“Problem-Based 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(olddopts,'param1',value1,...)
options = optimset(olddopts,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(olddopts,'param1',value1,...)` creates a copy of `olddopts`, modifying the specified options with the specified values.

`options = optimset(olddopts,newopts)` combines an existing options structure, `olddopts`, with a new options structure, `newopts`. Any options in `newopts` with nonempty values overwrite the corresponding old options in `olddopts`.

## Options

For more information about individual options, including their default values, see the reference pages for the optimization functions. “Optimization Options Reference” on page 15-8 provides descriptions of optimization options and which functions use them. `optimset` uses different names for some options than `optimoptions`. See “Current and Legacy Option Name Tables” on page 15-31.

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.

You can also see the default values of all relevant options in the Optimization app. To see the options:

- 1 Start the Optimization app, e.g., with the `optimtool` command.
- 2 Choose the solver from the **Solver** menu.
- 3 Choose the algorithm, if applicable, from the **Algorithm** menu.
- 4 Read off the default values within the **Options** pane.

## Examples

This statement creates an optimization options structure called `options` in which the `Display` option is set to `'iter'` and the `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

Select solver and optimization options, run problems

optimtool warns that it will be removed in a future release.

---

**Note** The Optimization app warns that it will be removed in a future release. For alternatives, see “Optimization App Alternatives” on page 5-15.

---

## Syntax

```
optimtool
optimtool(optstruct)
optimtool('solver')
```

## Description

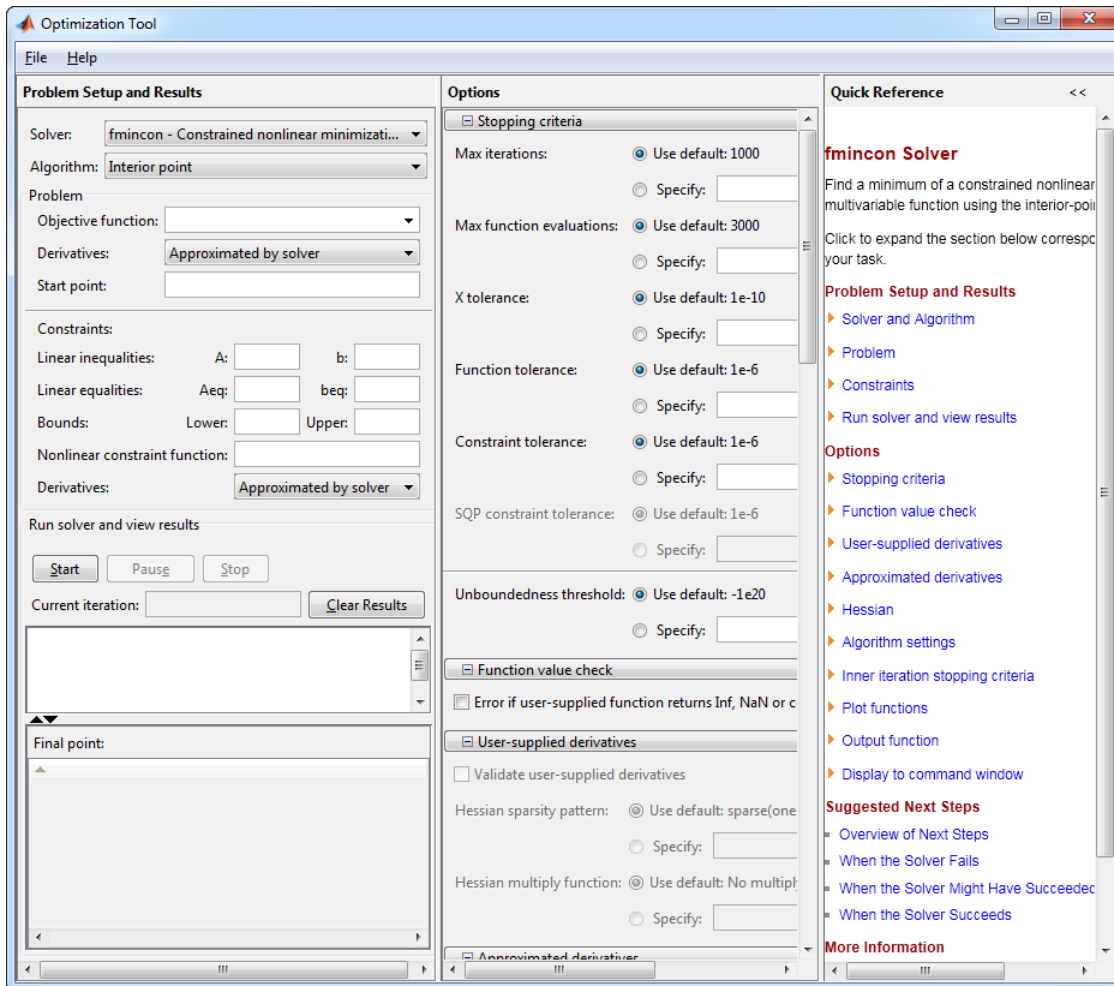
optimtool opens the Optimization app. Use the Optimization app to select a solver, optimization options, and run problems. See “Optimization App” on page 5-2 for a complete description of the Optimization app.

The Optimization app can be used to run any Optimization Toolbox solver except `intlinprog`, and any Global Optimization Toolbox solver except `GlobalSearch` and `MultiStart`. Results can be exported to a file or to the MATLAB workspace as a structure.

`optimtool(optstruct)` starts the Optimization app and loads `optstruct`. `optstruct` can be either optimization options or an optimization problem structure. Create optimization options with the `optimoptions` or `optimset` function, or by using the export option from the Optimization app. Create a problem structure by exporting the problem from the Optimization app to the MATLAB workspace. If you have Global Optimization Toolbox, you can create a problem structure for `fmincon`, `fminunc`, `lsqnonlin`, or `lsqcurvefit` using the `createOptimProblem` function.

`optimtool('solver')` starts the Optimization app with the specified solver, identified as a 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 intlinprog, GlobalSearch, and MultiStart.



**See Also**  
 optimoptions | optimset

**Topics**

“Optimization App” on page 5-2

“Solve a Constrained Nonlinear Problem, Solver-Based” on page 1-13

**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

`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 `cstrk`.

`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 `showvar`.  
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 showvar.
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 showvar.
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 showvar.  
  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 `showvar`.  
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 `showvar`.  
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 `showvar`.  
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 `showvar`.  
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” (MATLAB) and “Comparison of Handle and Value Classes” (MATLAB). 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 Workflow” on page 10-2

“Named Index for Optimization Variables” on page 10-22

**Introduced in R2017b**

## prob2struct

**Package:** optim.problemdef

Convert optimization problem to solver form

### Syntax

```
problem = prob2struct(prob)
problem = prob2struct(prob,x0)
problem = prob2struct( ____,Name,Value)
```

### Description

`problem = prob2struct(prob)` returns an optimization problem structure suitable for solving problems using `linprog`, `intlinprog`, `quadprog`, `lsqlin`, `fmincon`, or `fminunc`. 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 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 10-45.

```
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 = 3.6035e-11
```

## Input Arguments

### **prob** — Optimization problem

OptimizationProblem object

Optimization problem, specified as an `OptimizationProblem` object. Create a problem by using `optimproblem`.

```
Example: prob = optimproblem; prob.Objective = obj;
prob.Constraints.cons1 = cons1;
```

### **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 10-49.

```
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')
```

### **ConstraintFunctionName — Name of nonlinear constraint function file**

'generatedConstraints' (default) | file name

Name of the nonlinear constraint function file created by `prob2struct`, 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 `ConstraintFilename`, 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

### **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

### **ObjectiveFunctionName — Name of objective function file**

'generatedObjective' (default) | file name

Name of the objective function file created by `prob2struct`, 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 `ObjectiveFilename`, 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

### **Options — Optimization options**

`optimoptions(prob)` (default) | options object created by `optimoptions`

Optimization options, specified as the comma-separated pair consisting of 'Options' and an options object created by `optimoptions`. Create options for the appropriate solver; see `problem`.

Example: `optimoptions('fmincon','PlotFcn','optimplotfval')`

## Output Arguments

### problem — Problem structure

intlinprog problem structure | linprog problem structure | quadprog problem structure | lsqlin problem structure | fmincon problem structure | fminunc problem structure

Problem structure, returned as an intlinprog problem structure, linprog problem structure, quadprog problem structure, lsqlin problem structure, fmincon problem structure, or fminunc problem structure.

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
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
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

**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.

---

For linear and quadratic 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 2-39. 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` and, if appropriate, `ConstraintFunctionName` name-value pair arguments. 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

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.

## See Also

`OptimizationProblem` | `varindex`

## Topics

“Problem-Based Workflow” on page 10-2

“Include Derivatives in Problem-Based Workflow” on page 7-24

“Output Function for Problem-Based Optimization” on page 7-31

**Introduced in R2017b**

## quadprog

Quadratic programming

Solver for quadratic objective functions with linear constraints.

quadprog finds a minimum for a problem specified by

$$\min_x \frac{1}{2}x^T H x + f^T x \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub. \end{cases}$$

$H$ ,  $A$ , 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 2-40.

---

**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.

---

## 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

$x = \text{quadprog}(H, f)$  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  $x = H \setminus (-f)$ .

$x = \text{quadprog}(H, f, A, b)$  minimizes  $1/2*x'*H*x + f'*x$  subject to the restrictions  $A*x \leq b$ . The input  $A$  is a matrix of doubles, and  $b$  is a vector of doubles.

$x = \text{quadprog}(H, f, A, b, Aeq, beq)$  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  $b = []$ .

$x = \text{quadprog}(H, f, A, b, Aeq, beq, lb, ub)$  solves the preceding problem subject to the additional restrictions  $lb \leq x \leq ub$ . 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  $lb \leq x \leq ub$  to the interior of the box defined by the bounds. `quadprog` does not change components that respect the bounds.

---

$x = \text{quadprog}(H, f, A, b, Aeq, beq, lb, ub, x0)$  solves the preceding problem starting from the vector  $x0$ . If no bounds exist, set  $lb = []$  and  $ub = []$ . Some `quadprog` algorithms ignore  $x0$ ; see  $x0$ .

$x = \text{quadprog}(H, f, A, b, Aeq, beq, lb, ub, x0, options)$  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  $x0 = []$ .

$x = \text{quadprog}(problem)$  returns the minimum for `problem`, where `problem` is a structure described in "Description" on page 16-455. Create `problem` by exporting a problem using the Optimization app; see "Exporting Your Work" on page 5-11. Alternatively, create a problem structure from an `OptimizationProblem` object by using `prob2struct`.

$[x, fval] = \text{quadprog}(\text{___})$ , for any input variables, also returns `fval`, the value of the objective function at  $x$ :

$$fval = 0.5*x'*H*x + f'*x$$

$[x, fval, \text{exitflag}, \text{output}] = \text{quadprog}(\text{___})$  also returns `exitflag`, an integer that describes the exit condition of `quadprog`, and `output`, a structure that contains information about the optimization.

$[x, fval, \text{exitflag}, \text{output}, \text{lambda}] = \text{quadprog}(\text{___})$  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

$$f(x) = \frac{1}{2}x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2$$

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

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

where

$$H = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}$$
$$f = \begin{bmatrix} -2 \\ -6 \end{bmatrix},$$

subject to the linear 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

$$\frac{1}{2}x^T Hx + f^T x$$

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

$$0 \leq x \leq 1, \sum x = 1/2.$$

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 = 3x1
    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	3.064216e-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 Workflow” on page 10-2. Create an optimization problem equivalent to “Quadratic Program with Linear Constraints” on page 16-456.

```
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

$$0 \leq x \leq 1,$$

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

$$A*x \leq b,$$

where **x** is the column vector of N variables **x(:)**, and **b** is a column vector with M elements.

For example, to specify

$$\begin{array}{rclcl} x_1 & + & 2x_2 & \leq & 10 \\ 3x_1 & + & 4x_2 & \leq & 20 \\ 5x_1 & + & 6x_2 & \leq & 30, \end{array}$$

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

$$A*x \leq b,$$

where **x** is the column vector of N variables **x(:)**, and **A** is a matrix of size M-by-N.

For example, to specify

$$x_1 + 2x_2 \leq 10$$

$$\begin{array}{rclcl} 3x_1 & + & 4x_2 & \leq & 20 \\ 5x_1 & + & 6x_2 & \leq & 30, \end{array}$$

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

### **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

$$\text{Aeq} * x = \text{beq},$$

where `x` is the column vector of `N` variables `x(:)`, and `beq` is a column vector with `Me` elements.

For example, to specify

$$\begin{array}{rclclcl} x_1 & + & 2x_2 & + & 3x_3 & = & 10 \\ 2x_1 & + & 4x_2 & + & x_3 & = & 20, \end{array}$$

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

$$\text{Aeq} * x = \text{beq},$$

where `x` is the column vector of `N` variables `x(:)`, and `Aeq` is a matrix of size `Me-by-N`.

For example, to specify

$$\begin{array}{rcccccc} x_1 & + & 2x_2 & + & 3x_3 & = & 10 \\ 2x_1 & + & 4x_2 & + & x_3 & = & 20, \end{array}$$

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

### **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

$$x(i) \geq lb(i) \quad \text{for all } i.$$

If `numel(lb) < numel(x0)`, then `lb` specifies that

$$x(i) \geq lb(i) \quad \text{for } 1 \leq i \leq \text{numel}(lb).$$

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

$$x(i) \leq ub(i) \quad \text{for all } i.$$

If `numel(ub) < numel(x0)`, then `ub` specifies that

$$x(i) \leq ub(i) \quad \text{for } 1 \leq i \leq \text{numel}(ub).$$

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. This input is optional. `x0` applies only to the 'trust-region-reflective' algorithm when there are only bound constraints.

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 2-82.



## All Algorithms

### Algorithm

Choose the algorithm:

- 'interior-point-convex' (default)
- 'trust-region-reflective'

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. For details, see “Choosing the Algorithm” on page 2-8.

### 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 3-17):

- 'off' or 'none' displays no output.
- 'final' displays only the final output (default).

The 'interior-point-convex' algorithm allows 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  $2 * (\text{numberOfVariables} - \text{numberOfEqualities})$ .
- For all other algorithms and problems, the default value is 200.

For `optimset`, the option name is `MaxIter`. See “Current and Legacy Option Name Tables” on page 15-31.

- 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  $100*\text{eps}$ , about  $2.2204e-14$ .
  - For 'interior-point-convex' algorithms, the default value is  $1e-8$ .
- See “Tolerances and Stopping Criteria” on page 2-84.
- For `optimset`, the option name is `TolFun`. See “Current and Legacy Option Name Tables” on page 15-31.
- StepTolerance** Termination tolerance on  $x$ ; a positive scalar.
- For 'trust-region-reflective', the default value is  $100*\text{eps}$ , about  $2.2204e-14$ .
  - For 'interior-point-convex', the default value is  $1e-12$ .
- For `optimset`, the option name is `TolX`. See “Current and Legacy Option Name Tables” on page 15-31.

**'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 $100*\text{eps}$ , and linear equality-constrained problems use $1\text{e-}6$ . See “Tolerances and Stopping Criteria” on page 2-84.
HessianMultiplyFcn	<p>For <code>optimset</code>, the option name is <code>TolFun</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p> <p>Hessian multiply function, specified as a function handle. For large-scale structured problems, this function computes the Hessian matrix product <math>H*Y</math> without actually forming <math>H</math>. The function has the form</p> $W = \text{hmfun}(\text{Hinfo}, Y)$ <p>where <code>Hinfo</code> (and potentially some additional parameters) contain the matrices used to compute <math>H*Y</math>.</p> <p>See “Quadratic Minimization with Dense, Structured Hessian” on page 11-18 for an example that uses this option.</p>
<i>MaxPCGIter</i>	<p>For <code>optimset</code>, the option name is <code>HessMult</code>. See “Current and Legacy Option Name Tables” on page 15-31.</p> <p>Maximum number of PCG (preconditioned conjugate gradient) iterations; a positive scalar. The default is <math>\max(1, \text{floor}(\text{numberOfVariables}/2))</math> for bound-constrained problems. For equality-constrained problems, <code>quadprog</code> ignores <code>MaxPCGIter</code> and uses <code>MaxIterations</code> to limit the number of PCG iterations. For more information, see “Preconditioned Conjugate Gradient Method” on page 11-12.</p>
<i>PrecondBandWidth</i>	Upper bandwidth of the preconditioner for PCG; a nonnegative integer. By default, <code>quadprog</code> uses diagonal preconditioning (upper bandwidth 0). For some problems, increasing the bandwidth reduces the number of PCG iterations. Setting <code>PrecondBandWidth</code> to <code>Inf</code> 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 11-9.
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 Name Tables” on page 15-31.
LinearSolver	Type of internal linear solver in the algorithm: <ul style="list-style-type: none"> <li>• 'auto' (default) — Use 'sparse' if the H matrix is sparse and 'dense' otherwise.</li> <li>• 'sparse' — Use sparse linear algebra. See “Sparse Matrices” (MATLAB).</li> <li>• 'dense' — Use dense linear algebra.</li> </ul>

**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 $Aineq*x \leq bineq$
bineq	Vector in linear inequality constraints $Aineq*x \leq bineq$

<code>Aeq</code>	Matrix in linear equality constraints $Aeq \cdot x = beq$
<code>beq</code>	Vector in linear equality constraints $Aeq \cdot x = beq$
<code>lb</code>	Vector of lower bounds
<code>ub</code>	Vector of upper bounds
<code>x0</code>	Initial point for $x$
<code>solver</code>	'quadprog'
<code>options</code>	Options created using <code>optimoptions</code> or the Optimization app

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  $1/2 \cdot x' \cdot H \cdot x + f' \cdot x$  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 4-27.

### **`fval` — Objective function value at solution**

real scalar

Objective function value at the solution, returned as a real scalar. `fval` is the value of  $1/2 \cdot x' \cdot H \cdot x + f' \cdot x$  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 <code>options.MaxIterations</code> .
-2	Problem is infeasible. Or, for <code>'interior-point-convex'</code> , the step size was smaller than <code>options.StepTolerance</code> , but constraints were not satisfied.
-3	Problem is unbounded.
<b>'interior-point-convex' Algorithm</b>	
2	Step size was smaller than <code>options.StepTolerance</code> , constraints were satisfied.
-6	Nonconvex problem detected.
-8	Unable to compute a step direction.
<b>'trust-region-reflective' Algorithm</b>	
4	Local minimum found; minimum is not unique.
3	Change in the objective function value was smaller than <code>options.FunctionTolerance</code> .
-4	Current search direction was not a direction of descent. No further progress could be made.

**output — Information about optimization process**

structure

Information about the optimization process, returned as a structure with these fields:

<code>iterations</code>	Number of iterations taken
<code>algorithm</code>	Optimization algorithm used
<code>cgiterations</code>	Total number of PCG iterations ( <code>'trust-region-reflective'</code> algorithm only)
<code>constrviolation</code>	Maximum of constraint functions
<code>firstorderopt</code>	Measure of first-order optimality
<code>linearsolver</code>	Type of internal linear solver, <code>'dense'</code> or <code>'sparse'</code> ( <code>'interior-point-convex'</code> algorithm only)
<code>message</code>	Exit message

**Lambda — Lagrange multipliers at solution**

structure

Lagrange multipliers at the solution, returned as a structure with these fields:

<code>lower</code>	Lower bounds <code>lb</code>
<code>upper</code>	Upper bounds <code>ub</code>
<code>ineqlin</code>	Linear inequalities
<code>eqlin</code>	Linear equalities

For details, see “Lagrange Multiplier Structures” on page 3-27.

## 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 11-9.

## Alternative Functionality

### App

You can use the Optimization app for quadratic programming. Enter `optimtool` at the MATLAB command line, and choose the `quadprog` - Quadratic programming solver. For more information, see “Optimization App” on page 5-2.

### Problem-Based Approach

You can solve quadratic programming problems using the “Problem-Based Optimization Setup”. For examples, see “Quadratic Programming”.

### 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.

### See Also

`linprog` | `lsqlin` | `optimoptions` | `prob2struct`

### Topics

“Solver-Based Optimization Problem Setup”

“Optimization Results”

“Quadratic Programming”

“Mixed-Integer Quadratic Programming Portfolio Optimization: Solver-Based” on page 9-86

**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**

# showbounds

**Package:** optim.problemdef

Display variable bounds

## Syntax

```
showbounds(var)
```

## Description

`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 Workflow” on page 10-2

**Introduced in R2017b**

## showconstr

**Package:** optim.problemdef

Display optimization constraint

## Syntax

showconstr(constr)

## Description

showconstr(constr) displays the optimization constraint `constr` at the MATLABCommand 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

OptimizationConstraint object

Optimization constraint, specified as an `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.

## See Also

`OptimizationConstraint` | `showproblem` | `writeconstr`

## Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

## showexpr

**Package:** optim.problemdef

Display optimization expression

## Syntax

```
showexpr(expr)
```

## Description

`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 `writexpr` to generate a text file containing the expression information.

## See Also

OptimizationExpression | writexpr

## Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

## showproblem

**Package:** optim.problemdef

Display optimization problem

### Syntax

```
showproblem(prob)
```

### Description

`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 10-45.

```
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 :
```

```
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

OptimizationProblem object

Optimization problem, specified as an OptimizationProblem object. Create a problem by using `optimproblem`.

```
Example: prob = optimproblem; prob.Objective = obj;  
prob.Constraints.consl = consl;
```

## 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.

## See Also

`OptimizationProblem` | `showbounds` | `showconstr` | `showexpr` | `writeproblem`

## Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

# showvar

**Package:** optim.problemdef

Display optimization variable

## Syntax

```
showvar(var)
```

## Description

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)`

## See Also

OptimizationVariable | `optimvar` | `writevar`

## Topics

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**



# solve

**Package:** optim.problemdef

Solve optimization problem

## Syntax

```
sol = solve(prob)
sol = solve(prob,x0)
sol = solve( ____,Name,Value)
[sol,fval] = solve( ____)
[sol,fval,exitflag,output,lambda] = solve( ____)
```

## Description

`sol = solve(prob)` solves the optimization 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 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)
```

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, convert the peaks function to an optimization expression.

```
prob = optimproblem;
x = optimvar('x');
y = optimvar('y');
fun = fcn2optimexpr(@peaks,x,y);
prob.Objective = fun;
```

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)
```

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
```

### 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);
```

```
LP:           Optimal objective value is 1554.047531.
```

```
Cut Generation: Applied 8 strong CG cuts.
                Lower bound is 1591.000000.
```

```
Branch and Bound:
```

nodes explored	total time (s)	num int solution	integer fval	relative gap (%)
10000	0.80	0	-	-
18188	1.32	1	2.906000e+03	4.509804e+01
22039	1.64	2	2.073000e+03	2.270974e+01
24105	1.79	3	1.854000e+03	9.973046e+00
24531	1.82	3	1.854000e+03	1.347709e+00
24701	1.83	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);
```

```
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 explored	total time (s)	num int solution	integer fval	relative gap (%)
3627	0.31	2	2.154000e+03	2.593968e+01
5844	0.46	3	1.854000e+03	1.180593e+01

```

6204      0.50      3  1.854000e+03  1.455526e+00
6400      0.51      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 steps.\n');
```

```
Without an initial point, solve took 24701 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 = 2x1
```

```
    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')
```

```
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 16-499 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)
```

```
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.

```
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);
```

```
LP:           Optimal objective value is -1027.472366.
```

```
Heuristics:   Found 1 solution using rounding.
              Upper bound is -1027.233133.
              Relative gap is 0.00%.
```

```
Cut Generation: Applied 1 mir cut, and 2 strong CG cuts.
                Lower 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.

## Input Arguments

### **prob** — Optimization problem

OptimizationProblem object

Optimization problem, specified as an OptimizationProblem object. Create a problem by using `optimproblem`.

Example: `prob = optimproblem; prob.Objective = obj;`  
`prob.Constraints.consl = consl;`

### **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 10-49.

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 `linprog`, `intlinprog`, `quadprog`, `lsqlin`, `lsqnonneg`, `fminunc`, or `fmincon`. For the default solver, see `'solver'`.

Ensure that `options` are 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 9-45. 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 2-75 and “Improve Results”.

Example: `options = optimoptions('intlinprog','Display','none')`

### **solver — Optimization solver**

`'intlinprog'` | `'linprog'` | `'lsqlin'` | `'lsqnonneg'` | `'quadprog'` | `'fminunc'` | `'fmincon'`

Optimization solver, specified as the comma-separated pair consisting of `'solver'` and the name of a listed solver.

<b>Problem Type</b>	<b>Default Solver</b>	<b>Other Allowed Solvers</b>
Linear objective, linear constraints	<code>linprog</code>	<code>intlinprog</code> , <code>quadprog</code>
Linear objective, linear and integer constraints	<code>intlinprog</code>	<code>linprog</code> , <code>quadprog</code> (integer constraints ignored)

Problem Type	Default Solver	Other Allowed Solvers
Quadratic objective, linear constraints	quadprog	lsqlin, lsqnonneg (if objective cannot be converted to minimize $\ C*x - d\ ^2$ then solve throws an error for these solvers)
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 $x \geq 0$ are ignored for lsqnonneg)
Minimize $\ C*x - d\ ^2$ subject to $x \geq 0$	lsqlin	quadprog, lsqnonneg
Minimize general nonlinear function $f(x)$	fminunc	fmincon
Minimize general nonlinear function $f(x)$ subject to some constraints, or minimize any function subject to nonlinear constraints	fmincon	(none)

---

**Caution** For maximization problems, do not specify the `lsqlin` and `lsqnonneg` solvers. If you do, `solve` throws an error, because these solvers cannot maximize.

---

Example: `'intlinprog'`

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.

---

**Tip** If you neglect to ask for `fval`, you can calculate it using:

```
fval = evaluate(prob.Objective,sol)
```

---

**exitflag — Reason solver stopped**

categorical variable

Reason the solver stopped, returned as a categorical variable. This table describes the exit flags for the `intlinprog` solver.

Exit Flag for <code>intlinprog</code>	Numeric Equivalent	Meaning
<code>OptimalWithPoorFeasibility</code>	3	The solution is feasible with respect to the relative <code>ConstraintTolerance</code> tolerance, but is not feasible with respect to the absolute tolerance.
<code>IntegerFeasible</code>	2	<code>intlinprog</code> stopped prematurely, and found an integer feasible point.
<code>OptimalSolution</code>	1	The solver converged to a solution $x$ .
<code>SolverLimitExceeded</code>	0	<p><code>intlinprog</code> exceeds one of the following tolerances:</p> <ul style="list-style-type: none"> <li>• <code>LPMaxIterations</code></li> <li>• <code>MaxNodes</code></li> <li>• <code>MaxTime</code></li> <li>• <code>RootLPMMaxIterations</code></li> </ul> <p>See “Tolerances and Stopping Criteria” on page 2-84. <code>solve</code> also returns this exit flag when it runs out of memory at the root node.</p>

<b>Exit Flag for intlinprog</b>	<b>Numeric Equivalent</b>	<b>Meaning</b>
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.

<b>Exit Flag for linprog</b>	<b>Numeric Equivalent</b>	<b>Meaning</b>
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.
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 <code>lsqlin</code>	Numeric Equivalent	Meaning
FunctionChangeBelowTolerance	3	Change in the residual is smaller than the specified tolerance <code>options.FunctionTolerance</code> . (trust-region-reflective algorithm)
StepSizeBelowTolerance	2	Step size smaller than <code>options.StepTolerance</code> , constraints satisfied. (interior-point algorithm)
OptimalSolution	1	The solver converged to a solution $x$ .
SolverLimitExceeded	0	The number of iterations exceeds <code>options.MaxIterations</code> .
NoFeasiblePointFound	-2	The problem is infeasible. Or, for the interior-point algorithm, step size smaller than <code>options.StepTolerance</code> , but constraints are not satisfied.
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 <code>quadprog</code>	Numeric Equivalent	Meaning
LocalMinimumFound	4	Local minimum found; minimum is not unique.

Exit Flag for quadprog	Numeric Equivalent	Meaning
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 fminunc solver.

Exit Flag for fminunc	Numeric Equivalent	Meaning
NoDecreaseAlongSearchDirection	5	Predicted decrease in the objective function is less than the options.FunctionTolerance tolerance.



Exit Flag for fminunc	Numeric Equivalent	Meaning
FunctionChangeBelowTolerance	3	Change in the objective function value is less than the <code>options.FunctionTolerance</code> tolerance.
StepSizeBelowTolerance	2	Change in $x$ is smaller than the <code>options.StepTolerance</code> tolerance.
OptimalSolution	1	Magnitude of gradient is smaller than the <code>options.OptimalityTolerance</code> tolerance.
SolverLimitExceeded	0	Number of iterations exceeds <code>options.MaxIterations</code> or number of function evaluations exceeds <code>options.MaxFunctionEvaluations</code> .
OutputFcnStop	-1	Stopped by an output function or plot function.
Unbounded	-3	Objective function at current iteration is below <code>options.ObjectiveLimit</code> .

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 $2 \times \text{options.OptimalityTolerance}$ and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .
SearchDirectionTooSmall	4	Magnitude of the search direction is less than $2 \times \text{options.StepTolerance}$ and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .

Exit Flag for fmincon	Numeric Equivalent	Meaning
FunctionChangeBelowTolerance	3	Change in the objective function value is less than <code>options.FunctionTolerance</code> and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .
StepSizeBelowTolerance	2	Change in <code>x</code> is less than <code>options.StepTolerance</code> and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .
OptimalSolution	1	First-order optimality measure is less than <code>options.OptimalityTolerance</code> , and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .
SolverLimitExceeded	0	Number of iterations exceeds <code>options.MaxIterations</code> or number of function evaluations exceeds <code>options.MaxFunctionEvaluations</code> .
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 <code>options.ObjectiveLimit</code> and maximum constraint violation is less than <code>options.ConstraintTolerance</code> .

### 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:

- 'intlinprog' output
- 'linprog' output
- 'lsqin' output

- 'lsqnonneg' output
- 'quadprog' output
- 'fminunc' output
- 'fmincon' 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. For the `intlinprog` solver, `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**

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
- `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

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 16-444.

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 9-33. 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 the algorithms that `lsqlin` uses to solve linear least-squares problems, 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 6-22.

---

**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 Least-Squares, Problem-Based” on page 12-50.

---

## 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

`OptimizationProblem` | `evaluate` | `findindex` | `fmincon` | `fminunc` | `intlinprog` | `linprog` | `lsqlin` | `lsqnonneg` | `optimoptions` | `prob2struct` | `quadprog`

### Topics

“Problem-Based Workflow” on page 10-2

“Create Initial Point for Optimization with Named Index Variables” on page 10-49

**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

OptimizationProblem object

Optimization problem, specified as an OptimizationProblem object. Create a problem by using `optimproblem`.

Example: `prob = optimproblem; prob.Objective = obj; prob.Constraints.consl = consl;`

### **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 16-516.

## See Also

`OptimizationProblem` | `prob2struct`

## Topics

“Output Function for Problem-Based Optimization” on page 7-31

“Include Derivatives in Problem-Based Workflow” on page 7-24

**Introduced in R2019a**



# writebounds

**Package:** optim.problemdef

Save description of variable bounds

## Syntax

```
writebounds(var)
writebounds(var, filename)
```

## Description

`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”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

## writeconstr

**Package:** `optim.problemdef`

Save optimization constraint description

### Syntax

```
writeconstr(constr)
writeconstr(constr, filename)
```

### Description

`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

OptimizationConstraint object

Optimization constraint, specified as an `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`.

## **See Also**

`OptimizationConstraint` | `showconstr` | `writeproblem`

## **Topics**

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

# writeexpr

**Package:** optim.problemdef

Save optimization expression description

## Syntax

```
writeexpr(expr)  
writeexpr(expr, filename)
```

## Description

`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`.

## See Also

`OptimizationExpression` | `showexpr` | `writeproblem`

## Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**



# writeproblem

**Package:** optim.problemdef

Save optimization problem description

## Syntax

```
writeproblem(prob)  
writeproblem(prob, filename)
```

## Description

`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

OptimizationProblem object

Optimization problem, specified as an OptimizationProblem object. Create a problem by using `optimproblem`.

```
Example: prob = optimproblem; prob.Objective = obj;  
prob.Constraints.cons1 = cons1;
```

### **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`.

## See Also

`OptimizationProblem` | `showproblem` | `writebounds` | `writeconstr` | `writeexpr`

## Topics

"Problem-Based Optimization Setup"

"Problem-Based Workflow" on page 10-2

**Introduced in R2017b**

## writevar

**Package:** optim.problemdef

Save optimization variable description

## Syntax

```
writevar(var)
writevar(var, filename)
```

## Description

`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`

## See Also

`OptimizationVariable` | `optimvar` | `showvar`

## Topics

“Problem-Based Optimization Setup”

“Problem-Based Workflow” on page 10-2

**Introduced in R2017b**

