COMSOL SCRIPT

OPTIMIZATION LAB

USER'S GUIDE

VERSION 1.1

COMSOL

How to contact COMSOL:

Benelux
COMSOL BV
Röntgenlaan 19
2719 DX Zoetermeer
The Netherlands
Phone: +31 (0) 79 363 4230
Fax: +31 (0) 79 361 4212
info@femlab.nl
www.femlab.nl

Denmark
COMSOL A/S
Diplomvej 376
2800 Kgs. Lyngby
Phone: +45 88 70 82 00
Fax: +45 88 70 80 90
info@comsol.dk
www.comsol.dk

Finland
COMSOL OY
Arabianranta 6
FIN-00560 Helsinki
Phone: +358 9 2510 400
Fax: +358 9 2510 4010
info@comsol.fi
www.comsol.fi

France
COMSOL France
WTC, 5 pl. Robert Schuman
F-38000 Grenoble
Phone: +33 (0)4 76 46 49 01
Fax: +33 (0)4 76 46 07 42
info@comsol.fr
www.comsol.fr

Germany
FEMLAB GmbH
Berliner Str. 4
D-37073 Göttingen
Phone: +49-551-99721-0
Fax: +49-551-99721-29
info@femlab.de
www.femlab.de

Italy
COMSOL S.r.l.
Via Vittorio Emanuele II, 22
25122 Brescia
Phone: +39-030-3793800
Fax: +39-030-3793899
info.it@comsol.com
www.it.comsol.com

Norway
COMSOL AS
Søndre gate 7
NO-7485 Trondheim
Phone: +47 73 84 24 00
Fax: +47 73 84 24 01
info@comsol.no
www.comsol.no

Sweden
COMSOL AB
Tegnérgatan 23
SE-111 40 Stockholm
Phone: +46 8 412 95 00
Fax: +46 8 412 95 10
info@comsol.se
www.comsol.se

Switzerland
FEMLAB GmbH
Technoparkstrasse 1
CH-8005 Zürich
Phone: +41 (0)44 445 2140
Fax: +41 (0)44 445 2141
info@femlab.ch
www.femlab.ch

United Kingdom
COMSOL Ltd.
UH Innovation Centre
College Lane
Hatfield
Hertfordshire AL10 9AB
Phone:+44-(0)-1707 284747
Fax: +44-(0)-1707 284746
info.uk@comsol.com
www.uk.comsol.com

United States
COMSOL, Inc.
1 New England Executive Park
Suite 350
Burlington, MA 01803
Phone: +1-781-273-3322
Fax: +1-781-273-6603

COMSOL, Inc.
10850 Wilshire Boulevard
Suite 800
Los Angeles, CA 90024
Phone: +1-310-441-4800
Fax: +1-310-441-0868

COMSOL, Inc.
744 Cowper Street
Palo Alto, CA 94301
Phone: +1-650-324-9935
Fax: +1-650-324-9936

info@comsol.com
www.comsol.com

For a complete list of international representatives, visit
www.comsol.com/contact

Company home page
www.comsol.com

COMSOL user forums
www.comsol.com/support/forums

Optimization Lab User's Guide

Version: October 2007 COMSOL 3.4

CONTENTS

Chapter 1: Introduction

Chapter 2: Using the Optimization Lab

Chapter 3: Optimization Examples

Chapter 4: Multiphysics Optimization

Chapter 5: Command Reference

Chapter 6: Solver Properties

Chapter 7: Glossary

1

Introduction

Welcome to the Optimization Lab! This *User's Guide* details features and techniques to help you use this powerful package for all kinds of optimization. Through examples and code samples you will get an understanding of the optimization problems it is possible to solve and also learn about the solvers and algorithms that the Optimization Lab contains.

This introductory chapter provides an overview of the Optimization Lab.

The Documentation Set

The documentation for the Optimization Lab consists of this book, the *Optimization Lab User's Guide*, which provides full information about the product and its applications for optimization tasks. In addition, the Optimization Lab includes the *SQOPT User's Guide* and *SNOPT User's Guide* in PDF versions. For the general use of the COMSOL Script language and for installation of the software, the following resources provide additional information:

- *COMSOL Quick Installation Guide*—basic information for installing the COMSOL software and getting started. Included in the DVD/CD package.
- *COMSOL New Release Highlights*—information about new features and models in the 3.4 release. Included in the DVD/CD package.
- *COMSOL License Agreement*—the license agreement. Included in the DVD/CD package.
- *COMSOL Installation and Operations Guide*—besides covering various installation options for COMSOL Script, this manual describes the system requirements and options for running various COMSOL software products.
- *COMSOL Script User's Guide*—explains how to use the vast range of functions in the COMSOL Script language. This guide also describes COMSOL Script's programming-language features and the powerful graphics capabilities and tools it provides for creating custom graphical user interfaces.
- *COMSOL Script Command Reference*—provided only as online documentation as a PDF and in HTML format, it reviews each function in the COMSOL Script environment with syntax descriptions and examples.

Note: Following installation, the full documentation set is available on your computer in electronic versions—as PDF files and in HTML format.

Typographical Conventions

All COMSOL manuals use a set of consistent typographical conventions that should make it easy for you to follow the discussion, realize what you can expect to see on the

screen, and know which data you must enter into various data-entry fields. In particular, you should be aware of these conventions:

- A **boldface** font of the shown size and style indicates that the given word(s) appear exactly that way on the COMSOL graphical user interface (for toolbar buttons in the corresponding tooltip). For instance, we often refer to the **Model Navigator**, which is the window that appears when you start a new modeling session in COMSOL; the corresponding window on the screen has the title **Model Navigator**. As another example, the instructions might say to click the **Multiphysics** button, and the boldface font indicates that you can expect to see a button with that exact label on the COMSOL user interface.
- The names of other items on the graphical user interface that do not have direct labels contain a leading uppercase letter. For instance, we often refer to the Draw toolbar; this vertical bar containing many icons appears on the left side of the user interface during geometry modeling. However, nowhere on the screen will you see the term "Draw" referring to this toolbar (if it were on the screen, we would print it in this manual as the **Draw** menu).
- The symbol **>** indicates a menu item or an item in a folder in the **Model Navigator**. For example, **Physics>Equation System>Subdomain Settings** is equivalent to: On the **Physics** menu, point to **Equation System** and then click **Subdomain Settings**. **COMSOL Multiphysics>Heat Transfer>Conduction** means: Open the **COMSOL Multiphysics** folder, open the **Heat Transfer** folder, and select **Conduction**.
- A `Code` (monospace) font indicates keyboard entries in the user interface. You might see an instruction such as "Type `1.25` in the **Current density** edit field." The monospace font also indicates COMSOL Script codes.
- An *italic* font indicates the introduction of important terminology. Expect to find an explanation in the same paragraph or in the Glossary. The names of books in the COMSOL documentation set also appear using an italic font.

About the Optimization Lab

The Optimization Lab extends the COMSOL Script environment—an open and extensible language for technical computing of any kind—with a suite of tools for solving optimization problems.

The optimization algorithms in this product build on the proven SNOPT package developed by Prof. Philip Gill (University of California, San Diego) along with Profs. Walter Murray and Michael Saunders (Stanford University). SNOPT is a general-purpose system for large-scale nonlinearly constrained optimization.

The Optimization Lab builds on a flexible data structure, the Opt structure, which contains the entire optimization problem within a single variable. The main optimization function, `optim`, selects the proper solver algorithm depending on properties of the objective function and the constraints. Linearity, for example, allows `optim` to take appropriate shortcuts.

For optimization involving fields of physics and geometric properties, such as shape optimization, you can connect the Optimization Lab to COMSOL Multiphysics to perform optimization on a finite element model.

This documentation set introduces you to the full range of functionality in the Optimization Lab. We are certain that you will come up with some very creative uses of the powerful optimization tools it provides. We are anxious to hear about them and invite you to get in touch with us with any feedback whatsoever. We plan on developing the Optimization Lab even further, so let us know what kinds of functionality would serve you best. Contact us at `suggest@comsol.com` with any questions or comments you might have.

Overview

What Can the Optimization Lab Do?

The Optimization Lab is a powerful collection of optimization solvers based on the proven SNOPT and SQOPT codes developed by Philip Gill, Walter Murray, and Michael Saunders. You can use the Optimization Lab to solve large-scale linear and nonlinear optimization problems. It is especially effective for nonlinear problems, where functions and gradients can be expensive to evaluate. A solver routine that is common to all problem types, `optim`, chooses the best solver type for the optimization problem that you specify, and it uses a convenient structure variable, the Opt structure.

The Optimization Lab includes solvers for the following classes of problems:

- Linear optimization
- Quadratic optimization
- Nonlinear optimization
- Linear least squares
- Nonlinear least squares
- Unconstrained nonlinear optimization (Nelder-Mead search algorithm)

The Optimization Lab is fully integrated within the COMSOL Script technical-computing environment, which provides an open and extensible scripting tool for further data analysis and visualization.

By using this software together with other members of the COMSOL Multiphysics product family, you can perform optimization of space-dependent multiphysics problems.

Which Problems Can it Solve?

The Optimization Lab provides tools for solving optimization problems within a wide range of applications:

- General-purpose linear programming, quadratic programming, and nonlinear programming
- Optimization problems in engineering, economics, and finance

- Trajectory optimization, optimal control, engineering design, nonlinear networks, trade models, and the like
- Multiphysics applications together with COMSOL Multiphysics and COMSOL Script

2

Using the Optimization Lab

This chapter describes how to use the Optimization Lab to solve different types of optimization problems. It also shows how to define an optimization problem using the Opt structure—the data structure that you use to define all types of optimization problems in the Optimization Lab. Finally it describes the solvers and their properties, concluding with a small quadratic optimization example that puts all this information together.

Formulating Optimization Problems

Basic Concepts in Optimization

Optimization deals with minimizing (or maximizing) the value of a function. The function that you want to minimize, $f(x)$, is usually called the *objective function* or *cost function*. In addition to the objective function, optimization problems often include *constraints*. The Optimization Lab solves optimization problems in the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & b_{\text{lb}} \le Ax \le b_{\text{ub}} \\ & d_{\text{lb}} \le c(x) \le d_{\text{ub}} \\ & x_{\text{lb}} \le x \le x_{\text{ub}} \end{aligned}$$

Here, $f(x)$ is the objective function, which you can specify in a number of different formats. In addition, there are three types of constraints:

- The matrix A defines the *linear constraints* together with the vectors b_{lb} and b_{ub}, which are the lower and upper bounds, respectively.
- The vector function $c(x)$ defines the *nonlinear constraints* with the lower and upper bounds d_{lb} and d_{ub} (vectors), respectively.
- The vectors x_{lb} and x_{ub} define the lower and upper bounds for the variables x, respectively.

Note: These are *inequality constraints*. You can define an *equality constraint* by setting the upper bound equal to the lower bound. For example, to define the jth linear constraint as an equality, set $b_{\text{lb}}(j) = b_{\text{ub}}(j)$.

For a specific optimization problem, some of the available constraints might not apply. For an *unconstrained minimization* there are no constraints.

If x satisfies the constraints, then x is called *feasible*. The "optimal value" p^* of the problem is defined as $+\infty$ if no x satisfies the constraints and as $-\infty$ if f is unbounded from below. Otherwise p^* is the *GLB* (greatest lower bound) of f over the feasible set. If there is a feasible x that makes $f(x) = p^*$, then x belongs to a set of optimal points. A *feasibility problem* deals with finding the x satisfying the constraints.

MAXIMIZING INSTEAD OF MINIMIZING

The functions in the Optimization Lab solve minimization problems. To instead find the maximum of the objective function, set the solver property `'maximize'` to `'on'` for the gradient-based solvers. For `optnm`, to maximize $f(x)$, provide $-f(x)$ as the objective function.

Types of Optimization Problems

The most demanding problems have a nonlinear objective function and nonlinear constraints. For important subclasses of problems, the algorithms might take shortcuts to improve speed and robustness. For these reasons, the Optimization Lab distinguishes between linear and quadratic problems, also called *linear programming (LP)* and *quadratic programming (QP)* problems.

Because the constraints are linear, the feasible set is *convex*, that is, the entire line between two feasible points is feasible. Additionally, the objective function is convex for the linear problem and also for quadratic problems with a semidefinite matrix H.

LINEAR OPTIMIZATION PROBLEMS

A linear optimization problem has a linear objective function and only linear constraints:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & b_{\text{lb}} \le Ax \le b_{\text{ub}} \\ & x_{\text{lb}} \le x \le x_{\text{ub}} \end{aligned}$$

The objective function $c^T x$ is the dot product of the vector c and the vector of variables x. In addition, there are two types of constraints:

- The matrix A defines the *linear constraints* together with the vectors b_{lb} and b_{ub}, which are the lower bounds and upper bounds, respectively. The number of rows in A corresponds to the number of constraints. The number of columns must match the number of variables. The number of elements in the bounds b_{lb} and b_{ub} corresponds to the number of constraints.
- The vectors x_{lb} and x_{ub} define the bound constraints, that is, the lower and upper bounds for the variables x. The number of elements in the bounds x_{lb} and x_{ub} must match the number of variables.

If an optimization problem appears to be more general than the linear optimization problem, it is wise to examine the quadratic and nonlinear optimization problems in the following sections.

QUADRATIC OPTIMIZATION PROBLEMS

A *quadratic* problem (or *convex quadratic programming* problem) has a quadratic objective function and only linear constraints:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^T Hx + c^T x \\ \text{subject to} \quad & b_{\text{lb}} \le Ax \le b_{\text{ub}} \\ & x_{\text{lb}} \le x \le x_{\text{ub}} \end{aligned}$$

The matrix H (the *Hessian*) defines the quadratic term, and the vector c defines the linear part. For the unconstrained problem to have a local minimum, the Hessian must be *positive semidefinite*, that is, $x^T Hx \ge 0$ for all x. If $H = 0$ the problem is linear. The Optimization Lab always treats problems with quadratic constraints as nonlinear.

The linear constraints are exactly the same as for the linear problem as outlined in the previous section.

If an optimization problem appears to be more general than the quadratic optimization problem, it is wise to examine the nonlinear optimization problem type in the following section. In particular, if H is indefinite you must treat the objective function as a general nonlinear function and accept that the solution might be just a local minimizer.

NONLINEAR OPTIMIZATION PROBLEMS

A nonlinear optimization problem has a nonlinear objective function, nonlinear constraints, or both:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & b_{\text{lb}} \le Ax \le b_{\text{ub}} \\ & d_{\text{lb}} \le c(x) \le d_{\text{ub}} \\ & x_{\text{lb}} \le x \le x_{\text{ub}} \end{aligned}$$

The objective function $f(x)$ can be a nonlinear function, and the constraint function $c(x)$ defines the nonlinear constraints $d_{\text{lb}} \le c(x) \le d_{\text{ub}}$. The number of elements in the bounds d_{lb} and d_{ub} must match the number of nonlinear constraints.

The general nonlinear optimization solver in the Optimization Lab is gradient based, that is, it uses the gradient of the objective function and the Jacobian of the constraint function. The gradient is the column vector of derivatives of the objective function with respect to the variables

$$g_i = \frac{\partial f}{\partial x_i}, \tag{2-1}$$

and the Jacobian of the constraint function is the matrix of derivatives of constraint functions with respect to the variables

$$J_{ij} = \frac{\partial c_i}{\partial x_j}. \tag{2-2}$$

In addition to the nonlinear constraints, the problem description can include linear constraints, which are in the same form as in the previous sections.

LINEAR LEAST-SQUARES PROBLEMS

A constrained linear least-squares problem is a particular type of quadratic problem,

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}\|Cx - d\|^2 \\ \text{subject to} \quad & b_{\text{lb}} \le Ax \le b_{\text{ub}} \\ & x_{\text{lb}} \le x \le x_{\text{ub}} \end{aligned}$$

which is always convex. C is an m-by-n matrix, where m is the length of the column vector d, and n is the length of the vector x. When the columns of C are linearly dependent, the Hessian is semidefinite, but it is still possible to find a local minimizer, albeit nonunique.

The constraints of the linear least-squares problem are the same as for the linear and quadratic optimization problems.

NONLINEAR LEAST-SQUARES PROBLEMS

The Optimization Lab also treats constrained nonlinear least-squares problems of the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}\|F(x)\|^2 = \frac{1}{2}\sum_i F_i(x)^2 \\ \text{subject to} \quad & b_{\text{lb}} \le Ax \le b_{\text{ub}} \\ & d_{\text{lb}} \le c(x) \le d_{\text{ub}} \\ & x_{\text{lb}} \le x \le x_{\text{ub}} \end{aligned}$$

where the vector-valued function $F(x)$ can be a nonlinear function of the variables. The constraints of the nonlinear least-squares problem are the same as for the nonlinear optimization problem.

Optimization Algorithms

Starting at a feasible point (possibly after locating one, if none are known), an optimization algorithm works iteratively by sampling the objective function and constraints in the vicinity, then it moves to decrease the objective function without violating the constraints. The *gradient* $g(x)$ is the n-vector of first derivatives, and the *Hessian* $H(x)$ is the n-by-n matrix of second derivatives of the objective,

$$g_i = \frac{\partial}{\partial x_i} f(x), \quad H_{ij} = \frac{\partial}{\partial x_j} g_i(x) = \frac{\partial}{\partial x_j}\frac{\partial}{\partial x_i} f(x), \quad i, j = 1, \ldots, n .$$

If the objective is differentiable, it is well approximated by

$$f(x + h) \approx f(x) + g(x)^T h ,$$

and a move in the gradient direction changes the objective most rapidly. For twice-differentiable functions,

$$f(x + h) \approx f(x) + g(x)^T h + \frac{1}{2} h^T H(x) h$$

furnishes a much more accurate local approximation, the basis for the effective algorithms in the Optimization Lab, based on *quadratic programming* (QP). The Optimization Lab also provides a search algorithm (`optnm`) for unconstrained minimization of nonsmooth functions, which does not rely on any derivative information at the expense of using many samples.

The search stops when the estimated improvement that is possible—as judged by, for example, the magnitude of the "reduced" gradient (see "Reduced-Gradient Methods" on page 14) or the change over the last iteration—becomes small enough.

Problems with nonlinear constraints are solved with a *sequential quadratic programming* (SQP) method in which a reduced-gradient method solves each QP subproblem.

OPTIMALITY CONDITIONS AND LAGRANGE MULTIPLIERS

Consider an inequality constraint $c(x) \le 0$. If $c(\underline{x}) = 0$ at a point $\underline{x}$, then the constraint is called active at $\underline{x}$; otherwise it is inactive. Constraints that are inactive at the optimum can be disregarded, and the active constraints can be replaced by equalities. Note that it is not possible to know beforehand which will be the active set of constraints.

Note: The next section assumes that all constraints, whether linear, nonlinear, or bound constraints, are included in $c(x)$.

A local optimum of the equality-constrained problem

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & c(x) = 0 \end{array} \tag{2-3}$$

is characterized by the *1st-order optimality conditions*

$$\begin{aligned} g + J^T y &= 0 \\ c(x) &= 0 \end{aligned}$$

which say that a perturbation to x that does not violate any of the constraints (to first order) produces no 1st-order change to the objective. The *Lagrangian*,

$$L(x, y) = f(x) + y^T c(x),$$

is the sum of the objective function and a sum of the constraint functions weighted by the *Lagrange multipliers*, y. $L(x, y)$ has a stationary point at a minimum in Equation 2-3.

For a problem involving inequality constraints

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & c(x) \le 0 \end{array}$$

the 1st-order optimality conditions are

$$g + J^T y = 0$$
$$c(x) \le 0$$
$$y \ge 0$$

Only the active constraints have a corresponding nonzero Lagrange multiplier. These conditions are known as the *Kuhn-Tucker necessary conditions.*

To see how the Lagrange multipliers enter into the problem formulation of the Optimization Lab, consider the most general optimization problem definition

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & b_{\text{lb}} \le Ax \le b_{\text{ub}} \\ & d_{\text{lb}} \le c(x) \le d_{\text{ub}} \\ & x_{\text{lb}} \le x \le x_{\text{ub}} \end{array}$$

The 1st-order optimality conditions are

$$g(x) + A^T(y_{\text{lc}_{ub}} + y_{\text{lc}_{lb}}) + J^T(y_{\text{nc}_{ub}} + y_{\text{nc}_{lb}}) + y_{\text{bc}_{ub}} + y_{\text{bc}_{lb}} = 0$$
$$y_{\text{lc}_{ub}} \ge 0, \quad y_{\text{nc}_{ub}} \ge 0, \quad y_{\text{bc}_{ub}} \ge 0$$
$$y_{\text{lc}_{lb}} \le 0, \quad y_{\text{nc}_{lb}} \le 0, \quad y_{\text{bc}_{lb}} \le 0$$
$$b_{\text{lb}} \le Ax \le b_{\text{ub}}, d_{\text{lb}} \le c(x) \le d_{\text{ub}}, x_{\text{lb}} \le x \le x_{\text{ub}}$$

Note: Positive and negative multipliers indicate active lower and upper bounds, respectively.

Reduced-Gradient Methods

Multipliers convey useful information: nonzero values are the costs in objective-function units of satisfaction of the constraint. Algorithms for constrained problems predict the active set, solve the equality-constrained problem, and check for violation of the (presumedly) inactive constraints and possible release of the active constraints. The active set strategy selects the active set for the next step. The reduced-gradient algorithms for linear constraints choose an active set and optimize within the associated subspace, perhaps adding constraints one by one to the active set if they are encountered before the objective is sufficiently optimized. They then consider releasing an active constraint and continuing as before. The Optimization Lab

algorithms are reduced-gradient methods. They use the active constraints to eliminate some of the variables, effectively reducing the dimension of the search space and eliminating the constraints. Gradients in this space are called *reduced gradients*.

For problems with nonlinear constraints, the SQP algorithm ultimately solves a QP subproblem defined at the optimal point. The reduced-gradient notion therefore still applies.

Creating the Opt Structure

The Opt Structure

The Opt structure contains a complete representation of an optimization problem and its solution. To define a problem, you define its objective function in the Opt structure's `opt.obj` field, the nonlinear constraints in the `opt.nc` field, the linear constraints in the `opt.lc` field, and finally the bound constraints in the `opt.bc` field.

The Opt structure can also provide initial values of the variables to the optimization solvers in the `opt.init` field. The solvers store the solution to the optimization problem in the `opt.sol` field.

Defining the Objective Function

You specify the objective function in the Opt structure's `opt.obj` field. In addition to the description below, the entry `optim` on page 77 in the chapter "Command Reference" contains a table with information about the ways you can specify the objective function.

LINEAR OPTIMIZATION PROBLEMS

For linear optimization problems, you need specify only the vector c in the objective function $f(x) = c^T x$. Use the `opt.obj.c` field for this vector, which must have the same number of elements as the number of variables. Strictly speaking, c should be a column vector, but you can enter a row vector as well. In addition, it is good practice to specify that the problem is linear: `opt.obj.form='lin'`.

QUADRATIC OPTIMIZATION PROBLEMS

For quadratic optimization problems you specify the Hessian H and the vector c of the objective function $f(x) = \frac{1}{2}x^T Hx + c^T x$. Use the `opt.obj.c` field for the vector and the `opt.obj.H` field for the matrix. The number of elements in c as well as the number of rows and columns in H must be the same as the number of variables.

Alternatively, you can specify the Hessian as the name of a function that computes the matrix-vector product Hx for any given vector x. This allows you to exploit structure or sparsity in H.

You must specify at least one of the fields `opt.obj.H` or `opt.obj.c`. Strictly speaking, c should be a column vector, but you can enter a row vector, as well. In addition, it is good practice to specify that the problem is quadratic: `opt.obj.form='quad'`.

NONLINEAR OPTIMIZATION PROBLEMS

Specify the objective function in the `opt.obj.f` field of the Opt structure. You can give the function either as a string, denoting the function name, or as an inline function. Similarly, you specify the gradient of the objective function in the field `opt.obj.g`. You can omit the gradient, but it is good practice to include it, when available, for performance reasons.

If you provide the same function name in the fields `opt.obj.f` and `opt.obj.g`, the function must compute both, with the first output argument being the objective function and the second its gradient. The definition of the gradient appears in Equation 2-1.

Normally, the solver tries to estimate the gradient sparsity pattern through repeated objective function evaluations. This process can be very expensive, but you can avoid it by providing the sparsity pattern explicitly as a sparse matrix in the field `opt.obj.ptrn`. In addition, it is good practice to specify that the problem is nonlinear: `opt.obj.form = 'nlin'`.

LINEAR LEAST-SQUARES OPTIMIZATION PROBLEMS

For linear least-squares problems, the objective function looks like

$$\frac{1}{2}\|Cx - d\|^2 .$$

Specify C and d in `opt.obj.C` and `opt.obj.d`, respectively. The number of columns in C must match the number of variables, and the number of rows must match the length of d. In addition, it is good practice to specify that the problem is a linear least-squares problem: `opt.obj.form = 'linlsq'`.

NONLINEAR LEAST-SQUARES OPTIMIZATION PROBLEMS

For nonlinear least-squares problems the objective function looks like

$$f(x) = \frac{1}{2}\|F(x)\|^2 = \frac{1}{2}\sum_i F_i(x)^2 .$$

Specify F by providing the name of a function that returns the vector F_i (not the sum of squares) in the field `opt.obj.F`. Optionally, provide the name of a function that returns the Jacobian of $F(x)$, that is, the matrix

$$J_{ij} = \frac{\partial F_i}{\partial x_j}$$

in the field `opt.obj.J`. The solver by default tries to evaluate a sparsity pattern for J through repeated evaluations of F. To avoid this expensive process, you can provide the pattern explicitly as a sparse matrix in `opt.obj.ptrn`. In addition, it is good practice to specify that the problem is a nonlinear least-squares problem: `opt.obj.form='nlinlsq'`.

Defining Constraints

BOUND CONSTRAINTS

Use the fields `opt.bc.lb` and `opt.bc.ub` to specify lower and upper bounds, respectively, on the variables. Both fields should be column vectors whose length matches the number of variables, but you can use row vectors, as well. Instead of using vectors you can also specify a single value that sets the default for all constraints.

If a problem does not contain any bound constraints, you can omit the `opt.bc` field.

LINEAR CONSTRAINTS

Use the field `opt.lc.A` to specify the matrix A of the linear constraints, and use the fields `opt.lc.lb` and `opt.lc.ub` for their lower and upper bounds, respectively. The number of rows in A must correspond to the number of constraints, and the number of rows must match the number of variables. Both `opt.lc.lb` and `opt.lc.ub` should be column vectors whose length matches the number of variables, but you can use row vectors, as well. Instead of vectors you can also specify a single value that sets the default for all constraints.

If a problem does not contain any linear constraints, you can omit the `opt.lc` field.

NONLINEAR CONSTRAINTS

Specify the function that computes the nonlinear constraints in the `opt.nc.c` field of the Opt structure. The function should compute the values of all nonlinear constraints and can be given either as a string denoting the function name or as an inline function. Similarly, specify the Jacobian of the constraint function in the field `opt.nc.J`. You can omit the Jacobian, but it is good practice to provide it, when available, for performance

reasons. Also for performance reasons, it is a good idea to provide the Jacobian sparsity pattern as a sparse matrix in the field `opt.nc.ptrn`. Otherwise, the solver will try to estimate the sparsity pattern through repeated evaluations of the constraint residual.

If you provide the same function name in the fields `opt.nc.c` and `opt.nc.J`, the function must compute both, with the first output argument being the constraints and the second its Jacobian.

Solving Optimization Problems

General

All Optimization Lab solvers except `optnm` share a common interface function, `optim`, which solves optimization problems of all available types. It automatically dispatches an optimization problem to the least general optimization solver that can handle the problem.

To run the `optim` solver on an optimization problem that you have set up in the Opt structure `opt`, use the call

```
opt.sol=optim(opt);
```

Upon successful completion, the solver puts the result in the `opt.sol` field.

Providing Initial Values

All solvers benefit from getting an initial guess for the variables. You can provide it in the Opt structure as a vector in the `opt.init.x` field. Make sure that its length matches the number of variables. Formally it should be a column vector, but the Optimization Lab allows you to enter a row vector, as well.

For the nonlinear solver you might want to specify initial values for the Lagrange multipliers, too. See `optnlin` and `optnlinlsq` in the "Command Reference" chapter for details.

Interpreting the Solution

The field `opt.sol` contains output information from the solver. The optimal values for the variables appears in the `opt.sol.x` field, the extremum appears in the `opt.sol.eval.f` field, and the value of the constraints, if any, appears in the `opt.sol.eval.bc`, `opt.sol.eval.lc` and `opt.sol.eval.nc` fields.

In addition, you can find information about the success of the computation in `opt.sol.exit`. A value of 1 indicates success, and a value of 0 indicates failure. Further, `opt.sol.msg` gives a more descriptive text for the result of the computation. (The corresponding SQOPT/SNOPT output code is available in the field `opt.sol.info`.) The name of the algorithm ultimately called by `optim` is available in the field `opt.sol.algorithm`.

Auxiliary information on the computation is available in the field `opt.sol.xinfo`. See the entry for `optim` on page 77 in the "Command Reference" chapter for more information.

You can also find the values of the Lagrange multipliers for the constraints in the `opt.sol.y` structure. The `opt.sol.y.bc` field contains the Lagrange multipliers for the bound constraints, the `opt.sol.y.lc` field contains the Lagrange multipliers for the linear constraints, and the `opt.sol.y.nc` field contains the Lagrange multipliers for the nonlinear constraints.

The Optimization Lab Solvers

The following list describes the various solvers in the Optimization Lab.

THE LINEAR OPTIMIZATION SOLVER

The linear optimization solver, `optlin`, is based on the SQOPT solver (Ref. 1) and uses a sparse implementation of the primal simplex method.

THE QUADRATIC OPTIMIZATION SOLVER

The solver for quadratic optimization problems, `optquad`, is also based on the SQOPT solver (Ref. 1), which uses a reduced-Hessian active-set method, also known as a reduced-gradient method.

THE NONLINEAR OPTIMIZATION SOLVER

The solver for nonlinear optimization problems, `optnlin`, is based on the SNOPT solver (Ref. 2), which uses a sparse sequential quadratic programming (SQP) method, with SQOPT as the QP subproblem solver.

THE LINEAR LEAST-SQUARES OPTIMIZATION PROBLEM SOLVER

To solve linear least-squares optimization problems, the Optimization Lab's `optlinlsq` solver uses the SQOPT package with the following objective function:

$$f(x) = \frac{1}{2}x^T C^T C x - d^T C x + \frac{1}{2}d^T d\,.$$

The implementation of the Hessian-vector products is as $C^T(Cx)$, without forming $C^T C$ explicitly.

THE NONLINEAR LEAST-SQUARES OPTIMIZATION PROBLEM SOLVER

To solve nonlinear least-squares optimization problems, the Optimization Lab's `optnlinlsq` solver uses the SNOPT package with the following objective function and gradient:

$$f(x) = \frac{1}{2}\sum_i F_i(x)^2$$

$$g(x) = J^T F$$

THE NELDER-MEAD SOLVER FOR UNCONSTRAINED PROBLEMS

The Optimization Lab contains a simple solver for nonlinear unconstrained optimization problems (those where there are no constraints on the variables). The `optnm` solver uses the Nelder-Mead simplex algorithm as defined in Ref. 3. Because this is not a gradient-based method, it is particularly useful for the minimization of nonsmooth functions.

Solver Compatibility Chart

The solvers in the Optimization Lab cover many types of optimization problems. There is also a general optimization solver, `optim`, which solves an optimization problem by looking at the fields of the `opt` structure and calling the most appropriate solver among `optlin`, `optquad`, `optnlin`, `optlinlsq`, and `optnlinlsq`. For example, a general nonlinear objective function or a quadratic objective function with nonlinear constraints results in `optnlin` being called. To control the solver selection in `optim`, you can specify the problem type in the `opt.obj.form` field (see the reference entry for `optim` on page 77 for more information).

The following table shows the compatibility among the solvers and the types of optimization problems you can solve with the Optimization Lab.

TABLE 2-1: OPTIMIZATION SOLVER/OPTIMIZATION PROBLEM COMPATIBILITY CHART

SOLVER	LINEAR PROBLEMS	QUADRATIC PROBLEMS	NONLINEAR PROBLEMS	LINEAR LEAST-SQUARES	NONLINEAR LEAST-SQUARES	LINEAR CONSTRAINT	NONLINEAR CONSTRAINT
`optlin`	√					√	
`optquad`	√	√				√	
`optnlin`	√	√	√			√	√
`optlinlsq`				√		√	

TABLE 2-1: OPTIMIZATION SOLVER/OPTIMIZATION PROBLEM COMPATIBILITY CHART

SOLVER	LINEAR PROBLEMS	QUADRATIC PROBLEMS	NONLINEAR PROBLEMS	LINEAR LEAST-SQUARES	NONLINEAR LEAST-SQUARES	LINEAR CONSTRAINT	NONLINEAR CONSTRAINT
`optnlinlsq`				√	√	√	√
`optnm`	√	√	√				

In general, it is possible to use a nonlinear or quadratic solver also for linear problems of the same type.

For more information about the syntax and properties for each solver, see the sections "Command Reference" on page 73 and "Solver Properties" on page 119.

The following section provides an introductory example of solving an optimization problem using the Optimization Lab. More illustrations are available in the sections "Optimization Examples" on page 29 and "Multiphysics Optimization" on page 49. This latter section focuses on the optimization of multiphysics models created using COMSOL Multiphysics products.

A First Optimization Example

This first example shows how to create the Opt structure with the objective function and constraints for a quadratic optimization problem and then solve it.

The Optimization Problem

For a given value of n, find the n-vector x that is closest in Euclidean norm to a given vector x_0, which in this case is equal to the vector `[1, 2, 3]'` (using COMSOL Script notation).The complication is that not only must x lie in the set

$$S = \left\{ x: \sum_{j=1}^{n} x_j = 1, x \geq 0 \right\},$$

but its components must also be nonincreasing: $x_j \leq x_{j+1}$. It is possible to write this situation in the form of a quadratic problem:

$$\begin{array}{ll} \underset{x1,\, x2}{\text{minimize}} & \frac{1}{2}\sum_{j=1} (x_j - (x_0)_j)^2 \\ \text{subject to} & x_j - x_{j+1} \leq 0,\ j = 1, 2, \ldots, n-1 \\ & \sum_{j=1}^{n} x_j = 1, x \geq 0 \end{array}$$

You can now write the objective function to be minimized as

$$\frac{1}{2}(x - x_0)^T (x - x_0) = \frac{1}{2}x_0^T x_0 - x_0^T x + \frac{1}{2}x^T x,$$

which is a quadratic problem with $H = I$ (the identity matrix) and $c = -x_0$. The additional constant term, $\frac{1}{2}x_0^T x_0$, is not part of the optimization problem that you pass to the optimization function. Instead you add it to the solution that the optimization routine returns.

To implement the constraints, write them for the nonincreasing components as $-\infty \leq x_j - x_{j+1} \leq 0$. These $n-1$ constraints, together with the restriction that the sum of all variables must be 1, define m so-called *range constraints*, $b_{\text{lb}} \leq Ax \leq b_{\text{ub}}$, in this case with m equal to n. For $n = 3$,

$$b_{\mathrm{lb}} = \begin{pmatrix} -\infty \\ -\infty \\ 1 \end{pmatrix}, A = \begin{pmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, b_{\mathrm{ub}} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

In addition, the nonnegativity constraints on the variables x form bounds constraints on x: $x_{\mathrm{lb}} \le x \le x_{\mathrm{ub}}$, where

$$x_{\mathrm{lb}} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, x_{\mathrm{ub}} = \begin{pmatrix} \infty \\ \infty \\ \infty \end{pmatrix}.$$

Creating the Opt Structure and Solving the Problem

Perform the following steps to create the Opt structure and solve the optimization problem:

1. First create a function that returns the matrix for the quadratic term of the objective function as a symmetric matrix or Hx. The following function, `objhx`, implements the later case, which is simply x, because H is the identity matrix in this optimization problem:

   ```
   function Hx = objhx(x)
   Hx = [x(1), x(2), x(3)]';
   ```

   The function that defines Hx takes the present vector of variables as the first input argument. It is possible to add extra input arguments using the `param` solver option (see `optprop` on page 99 for details). This is not necessary in this example.

2. Next define the vector for the linear part, c, which in this case is $-x_0$:

   ```
   c = -[1 2 3]';
   ```

3. Start defining the Opt structure:

   ```
   clear opt;
   opt.obj.H = 'objhx';
   opt.obj.c = c;
   ```

4. Next define the constraint matrix A plus the lower and upper bounds (b_{lb} and b_{ub}) for the linear constraints:

   ```
   opt.lc.A = [-1 1 0; 0 -1 1; 1 1 1];
   opt.lc.lb = [-Inf -Inf 1]';
   ```

```
opt.lc.ub = [0 0 1];
```

Instead of defining A directly, writing a small script function that creates A for an arbitrary column dimension would provide an extensible code.

Notice that the lower and upper bounds are equal for the third row, which sums the variables, making it an equality constraint.

5 Define the range for the variables through the lower and upper bounds for x (x_{lb} and x_{ub}, respectively):

```
opt.bc.lb = 0;
opt.bc.ub = Inf;
```

6 Finally define an initial guess and solve the optimization problem using the general optimization routine `optim`.

```
opt.init.x = ones(3,1);
opt.sol = optim(opt);
```

You could also use the dedicated function for solving quadratic optimization problems, `optquad` (see `optquad` on page 107 in the "Command Reference" chapter for more information).

Bibliography

1. P.E. Gill, W. Murray, and M.A. Saunders, *User's Guide for SQOPT Version 7: Software for Large-Scale Nonlinear Programming*, Systems Optimization Laboratory, Dept. Management Science and Engineering, Stanford Univ., 2006.

2. P.E. Gill, W. Murray, and M.A. Saunders, *User's Guide for SNOPT Version 7: Software for Large-Scale Linear and Quadratic Programming*, Systems Optimization Laboratory, Dept. Management Science and Engineering, Stanford Univ., 2006.

3. J.C. Lagarias, J.A. Reeds, M.H. Wright, and P.E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions," *SIAM J. Optimization*, vol. 9, pp. 112–147, 1998.

4. J. Nocedal and S.J. Wright, *Numerical Optimization*, Springer-Verlag, 1999.

3

Optimization Examples

This chapter reviews a series of solved optimization problems that make use of the suite of solvers in the Optimization Lab. A first introductory set of examples deals with circumscribing points on a plane or inscribing a circle on a polytope and uses several approaches to solving the problems.

Overview

Table 3-1 lists the examples in this and the next chapter in their order of appearance and also categorizes them by the solver each uses. The introductory "circle" models have nice geometric interpretations and take you through the modeling procedure with the least effort. They also illustrate the gains possible by refining the problem formulation when problems are large or when minimizing computing time is important, or inversely, the gains to be made by selecting the simplest problem formulation and leaving the work to the computer. The Rosenbrock problem is a classic example of an unconstrained optimization.

Examples using COMSOL Multiphysics appear in the next chapter, "Multiphysics Optimization" on page 49.

TABLE 3-1: OPTIMIZATION EXAMPLES CHART

EXAMPLE MODEL	LIN	QUAD	NLIN	NM	LINLSQ	NLINLSQ	GRADIENT	PAGE
`circlenm.m`				√			none	31
`circlenlin.m`			√				supplied	35
`circlequad.m`		√					supplied	39
`circlelin.m`	√						supplied	43
`rosenbrock.m`				√			none	47
`diode.m`[a]						√	numeric	51
`spinning_gear.m`[b]			√	√			numeric	63

a. Requires COMSOL Multiphysics.

b. Requires COMSOL Multiphysics and the Structural Mechanics Module.

Circumscribing Points on a Plane

This example shows how to set up the same optimization problem for different solvers in the Optimization Lab, and also touches upon the importance of the problem formulation.

Introduction

The goal is to find the smallest circle enclosing all the points $\{x^{(i)}\}, i = 1, 2, \ldots n$ on a plane. If $x \in (x_1, x_2)^T$ is the circle's center, the distance from the center to point $x^{(i)}$ is given by the norm $\|x^{(i)} - x\|_2$. Then you can express the optimization problem as "minimize, by choice of x, the maximum distance from x to any of the points $x^{(i)}$:"

$$\underset{x}{\text{minimize}} \qquad \max_{i\,=\,1,\,\ldots,\,n} \left\|x^{(i)} - x\right\|_2 .$$

There are no constraints on x (except, of course, that $x \in R^2$). The objective function is a strictly convex function of x. It follows that there is a single, global minimum. The only potential problem is that although the objective function is continuous, it is not differentiable. The adaptive search method `optnm` (see page 95 in the "Command Reference" chapter) does not require the objective to be differentiable, so try it first. Next you can reduce the number of iterations by reformulating the problem with differentiable but nonlinear constraints and use the nonlinear solver `optnlin`. Recognizing that the problem can be stated with a quadratic objective function with linear constraints, you can further reduce the number of iterations with the quadratic solver `optquad`.

Unconstrained Optimization

The distance from x to $x^{(i)}$ is given by the norm $\|x^{(i)} - x\|_2$. Then the distance to the furthest point is the radius of the smallest circle enclosing all the points for that x.

MODEL DEFINITION

Minimize, by choice of x, the square of the maximum distance from x to any of the points $x^{(i)}$

$$\min_{x} \qquad \max_{i\,=\,1,\,\ldots,\,n} (x^{(i)} - x)^T (x^{(i)} - x) .$$

RESULTS

For the set of points

$$X = \left[x^{(1)} \ \ldots \ x^{(5)}\right] = \begin{bmatrix} 0 & 9 & 13 & 7 & 2 \\ 0 & -2 & 4 & 8 & 7 \end{bmatrix}$$

the smallest circle containing all the points has its center at $x = [6.5000 \ 2.0000]^T$ and has a radius $r = 6.8007$. The results appear in Figure 3-1.

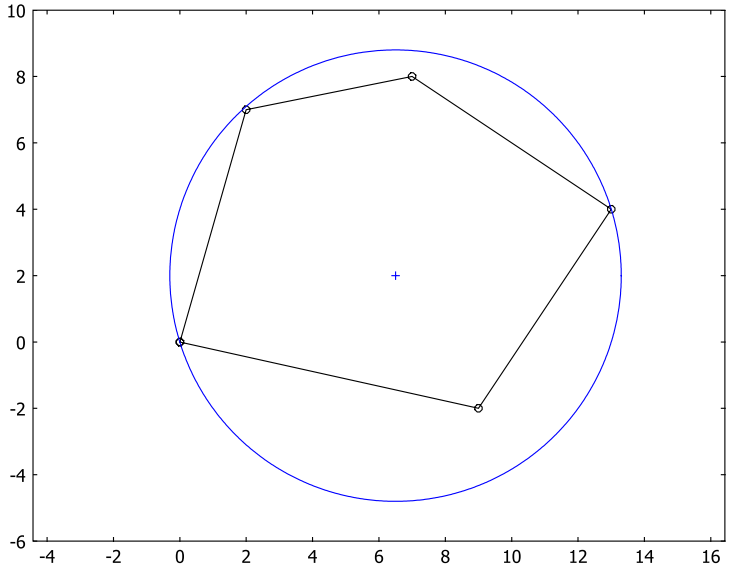


Figure 3-1: The smallest circle circumscribing all the points. Notice that the top left point lies inside, rather than on, the circle.

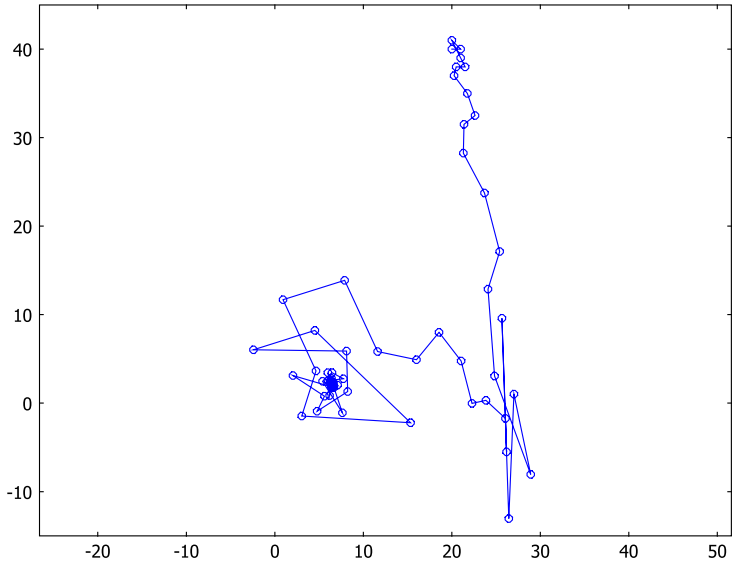


Figure 3-2: With an initial guess of $x = [20\ 40]^T$, the software reaches the solution in 93 iterations using 179 function evaluations. This can be improved by refining the problem formulation.

STEP-BY-STEP INSTRUCTIONS

1 Create a function file `circlenm_obj.m` that returns the objective function, and save it in a directory that is on the COMSOL Script path.

```
function f = circlenm_obj(x)
global X IND xLIST
[m,n]=size(X);

% Help variable
y = X - x*ones(1,n);

% Objective function
f = max(y(1,:).^2 + y(2,:).^2);

% Solution history
xLIST(:,IND) = x;
IND = IND+1;
```

2 Either from the command line or in another file, for example, `circlenm.m`, define the point matrix X and declare it global.

```
clear opt;
global X IND xLIST

% Points
X = [0 9 13 7 2;
     0 -2 4 8 7];

% Solution history
IND = 1;
xLIST = zeros(2,100);

% Objective
opt.obj.f = 'circlenm_obj';

% Initial guess
opt.init.x = [20;40];

% Solve
opt.sol = optnm(opt,'report','on','dtol',1e-6);

% Postprocessing
if(opt.sol.exit==1)
  fprintf(['Optimality conditions satisfied.\n'])
end
x = opt.sol.x
r = sqrt(max(sum((X-x).^2)))

% Points
figure
plot([X(1,:) X(1,1)],[X(2,:) X(2,1)],'k')
hold on
plot([X(1,:) X(1,1)],[X(2,:) X(2,1)],'ko')

% Circle
theta = linspace(0,2*pi,1000);
plot(x(1),x(2),'+')
plot(x(1)+r*cos(theta),x(2)+r*sin(theta))
axis equal

% Solution history
figure
plot(xLIST(1,1:IND-1),xLIST(2,1:IND-1),xLIST(1,1:IND-1),xLIST(2,1
:IND-1),'ob')
axis equal
```

3 Here you have just set up the problem using the `opt` structure. When working with the Nelder-Mead Simplex solver, `optnm`, you can also use the shortcut command

```
[x,f,exit] =
optnm('circlenm_obj',[20;40],'report','on','dtol',1e-6).
```

By reformulating the optimization problem as a constrained optimization problem, you can solve it with considerably less computational effort. By minimizing over both the circle center x and the radius r, it is possible to write the problem as

$$\begin{array}{ll} \underset{x,\,r}{\text{minimize}} & r \\ \text{subject to} & r \geq \left\| x - x^{(i)} \right\|_2 \qquad i = 1, \ldots, n \end{array}$$

MODEL DEFINITION

By introducing the auxiliary 1D variable s representing the square of the radius, you can state the problem with nonlinear constraints:

$$\begin{array}{ll} \underset{x,\,s}{\text{minimize}} & s \\ \text{subject to} & s + 2x^{(i)T}x - x^T x \geq x^{(i)T}x^{(i)} \qquad i = 1, \ldots, n \end{array}$$

By comparing it to `optnlin` on page 86 of the "Command Reference" it is possible to identify the components of the Opt structure.

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & b_{\text{lb}} \leq Ax \leq b_{\text{ub}} \\ & d_{\text{lb}} \leq c(x) \leq d_{\text{ub}} \\ & x_{\text{lb}} \leq x \leq x_{\text{ub}} \end{array}$$

By introducing $y = \begin{bmatrix} s & x_1 & x_2 \end{bmatrix}^T$ in the objective and constraint functions you can enter the problem using

$$f = s = y(1),\ c(y) = \begin{bmatrix} s \\ \vdots \\ s \end{bmatrix} + \begin{bmatrix} 2x^{(1)T}x \\ \vdots \\ 2x^{(5)T}x \end{bmatrix} - \begin{bmatrix} x^T x \\ \vdots \\ x^T x \end{bmatrix}, \text{ and } d_{\text{lb}} = \begin{bmatrix} x^{(1)T}x^{(1)} \\ \vdots \\ x^{(5)T}x^{(5)} \end{bmatrix}$$

There are no linear constraints, and the remaining bounds are $-\infty$ and $+\infty$, which are set by default.

You should also supply the gradient of the objective function and the Jacobian of the constraints function whenever possible. Shortly this discussion examines how performance decreases as the amount of information decreases. In this case the gradient is simply $g = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$ and the Jacobian is this 5-by-3 matrix:

$$J = \begin{bmatrix} 1 & 2(x_1^{(1)} - x_1) & 2(x_2^{(1)} - x_2) \\ \vdots & \vdots & \vdots \\ 1 & 2(x_1^{(5)} - x_1) & 2(x_2^{(5)} - x_2) \end{bmatrix}$$

RESULTS

The result is, of course, the same as before, but the way and speed the software reaches it is rather different (see Figure 3-3).

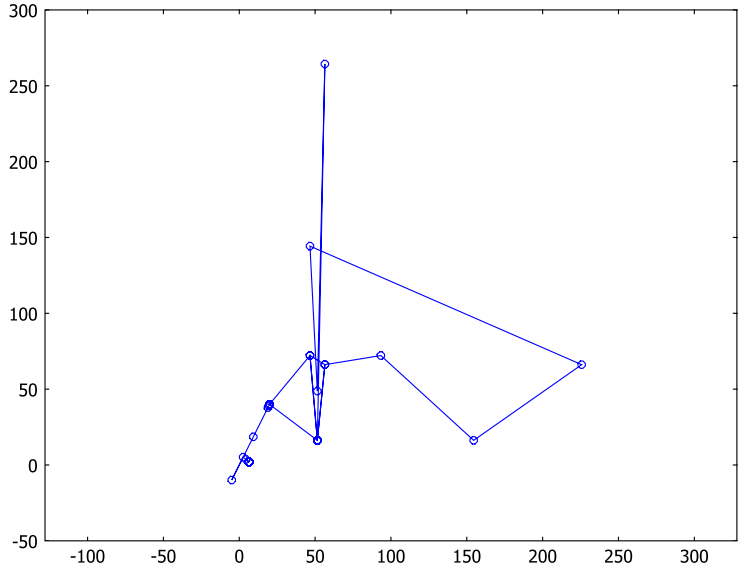


Figure 3-3: With an initial guess s = 1 and x = (20, 40), the solver reaches a solution after 31 function evaluations. This can also be improved by further refining the problem formulation.

You could, as noted earlier, solve the optimization problem without supplying the Jacobian of the constraints function or the gradient of the objective. The Optimization Lab then approximates them numerically; this can be useful for cases when these are

difficult or impossible to provide analytically. The tradeoff, of course, is slower convergence as shown in Table 3-2.

TABLE 3-2: COMPUTATIONAL COST OF OMITTING THE GRADIENT AND JACOBIAN

INFORMATION SUPPLIED	OBJECTIVE FUNCTION EVALUATIONS NEEDED
Both gradient g and Jacobian J	31
No gradient	57
No Jacobian	80
Neither gradient nor Jacobian	120

STEP-BY-STEP INSTRUCTIONS

1 Create a function file `circlenlin_obj.m` that returns the objective function and the gradient of the objective function with respect to the auxiliary variable, s, and the circle center, x:

```
function [f,g]= circlenlin_obj(y)
global X IND xLIST

% Variables
s = y(1);
x = [y(2);y(3)];

% Objective function
f = s;

% Gradient
g = [1,0,0];

% Solution history
xLIST(:,IND) = [s; x];
IND = IND+1;
```

2 Create another function file `circlenlin_con.m` that returns the nonlinear constraints function and the gradient of the constraints function.

```
function [c,J] = circlenlin_con(y)
global X IND xLIST

% Variables
s = y(1);
x = [y(2);y(3)];
[m,n]=size(X);

% Constraints function
c = s*ones(n,1)+2*X'*x-x'*x*ones(n,1);

% Jacobian of constraints function
```

```
J = [ones(n,1),2*(X'-ones(n,1)*x')];
```

3 Finally, either create a file `circlenlin.m` or run the optimization from the command line:

```
clear opt;
global X IND xLIST

% Points
X = [0 9 13 7 2;
     0 -2 4 8 7];
[m,n] = size(X);

% Solution history
IND = 1;
xLIST = zeros(3,1);

% Objective
opt.obj.f = 'circlenlin_obj';
opt.obj.g = 'circlenlin_obj';

% Nonlinear constraints
opt.nc.c = 'circlenlin_con';
opt.nc.J = 'circlenlin_con';

% Bounds for nonlinear constraints
opt.nc.lb = X(1,:).^2+X(2,:).^2;
opt.nc.ub = Inf*ones(1,n);

% Initial guess
opt.init.x = [1;20;40];

% Solve
opt.sol = optnlin(opt);

% Postprocessing
fprintf([opt.sol.msg '\n'])
s = opt.sol.x(1)
x = opt.sol.x(2:3)
r = sqrt(s)

% Points
figure
plot([X(1,:) X(1,1)],[X(2,:) X(2,1)],'k')
hold on
plot([X(1,:) X(1,1)],[X(2,:) X(2,1)],'ko')

% Circle
theta = linspace(0,2*pi,1000);
plot(x(1),x(2),'+')
```

```
plot(x(1)+r*cos(theta),x(2)+r*sin(theta))
axis equal

% Solution history
figure
plot(xLIST(2,1:IND-1),xLIST(3,1:IND-1),xLIST(2,1:IND-1),xLIST(3,1
:IND-1),'ob')
axis equal
```

Note: The solution vector in the `opt` structure is still named `opt.sol.x` no matter what you choose to call the variables in the objective and constraints functions. You can see this in the previous postprocessing section.

Quadratic Optimization

A close look at the problem formulation in "Unconstrained Optimization" on page 31 shows that you can rewrite the problem as

$$\underset{x}{\text{minimize}} \qquad x^T x + \max_{i\,=\,1,\,\ldots,\,n} (x^{(i)^T} x^{(i)} - 2x^{(i)^T} x)$$

MODEL DEFINITION

By introducing the auxiliary 1D variable t you can state the problem with linear constraints:

$$\begin{aligned} &\underset{x}{\text{minimize}} && x^T x + t \\ &\text{subject to} && x^{(i)^T} x^{(i)} \le 2x^{(i)^T} x + t \quad i = 1, \ldots, n \end{aligned}$$

This is a convex quadratic problem, that is, a convex problem with quadratic objective and linear constraints. By comparing the problem to `optquad` on page 107 of the "Command Reference" it is once again possible to identify the components of the `opt` structure:

$$\begin{aligned} &\underset{x}{\text{minimize}} && \frac{1}{2} x^T H x + c^T x \\ &\text{subject to} && b_{\text{lb}} \le Ax \le b_{\text{ub}} \\ & && x_{\text{lb}} \le x \le x_{\text{ub}} \end{aligned}$$

You can see that

$$x = \begin{bmatrix} t \\ x_1 \\ x_2 \end{bmatrix}, H = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}, c = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, A = \begin{bmatrix} 1 & 2x_1^{(1)} & 2x_2^{(1)} \\ \vdots & \vdots & \vdots \\ 1 & 2x_1^{(n)} & 2x_2^{(n)} \end{bmatrix}, \text{ and}$$

$$b_{\text{lb}} = \begin{bmatrix} x^{(1)^T} x^{(1)} \\ \vdots \\ x^{(n)^T} x^{(n)} \end{bmatrix}$$

The remaining bounds are $-\infty$ and $+\infty$, which are set by default.

RESULTS

Yet again you have improved the method of solving the problem. Now it takes just 10 function evaluations compared to the 179 evaluations for the unconstrained problem.

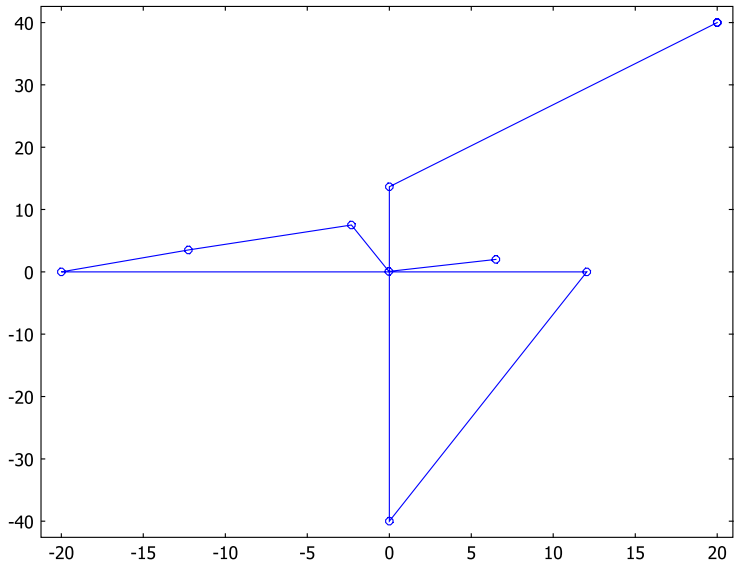


Figure 3-4: With an initial guess of t = 0 and x = (20, 40), the software reaches a solution after10 objective function evaluations.

Reworking this example by preparing and reformulating the problem shows that you can gain considerable efficiency by using more specialized solvers. The cost is the preparation time. For a small problem, just setting it up the first way you think of

probably minimizes the time to solution, whereas for large problems more care is needed.

STEP-BY-STEP INSTRUCTIONS

1 Create a function file `circlequad_Hx.m` that returns the Hx part of the objective function. In this example it would be easier to instead supply only the matrix H, but then you could not use the trick with global variables in the objective function to see how the solution was reached.

```
function Hx = circlequad_Hx(x)
global IND xLIST

% Variables
Hx = 2*x;
Hx(1) = 0;

% Solution history
xLIST(:,IND) = x;
IND = IND+1;
```

2 Create a file `circlequad.m` or run this code from the command line:

```
clear opt;
global IND xLIST

% Points
X = [0 9 13 7 2;...
     0 -2 4 8 7];
[m,n] = size(X);

% Solution history
IND = 1;
xLIST = zeros(3,1);

% Objective
opt.obj.H = 'circlequad_Hx';
opt.obj.c = [1 0 0]';

% Linear constraints:
opt.lc.A  = [ones(n,1), 2*X(1,:)', 2*X(2,:)'];

% Bounds for linear constraints
opt.lc.ub = inf*ones(n,1);
opt.lc.lb = (X(1,:).^2+X(2,:).^2)';

% Initial guess
opt.init.x = [0;20;40];

% Solve
```

```
opt.sol = optquad(opt);

% Postprocessing
fprintf([opt.sol.msg '\n'])
t = opt.sol.x(1)
x = opt.sol.x(2:3)
r = sqrt(t+x'*x)

% Points
figure
plot([X(1,:) X(1,1)],[X(2,:) X(2,1)],'k')
hold on
plot([X(1,:) X(1,1)],[X(2,:) X(2,1)],'ko')

% Circle
theta=linspace(0,2*pi,1000);
plot(x(1),x(2),'+')
plot(x(1)+r*cos(theta),x(2)+r*sin(theta))
axis equal

% Solution history
figure
plot(xLIST(2,1:IND-1),xLIST(3,1:IND-1),xLIST(2,1:IND-1),xLIST(3,1
:IND-1),'ob')
```

Note: The function call to solve the quadratic optimization problem is made to `optim` rather than `optquad`. The Optimization Lab automatically detects the problem's form and applies the most specialized solver.

Inscribing a Circle in a Polytope

As an extension of the previous problem, consider the irregular pentagon that was plotted in the solution in Figure 3-1 on page 32. This model exemplifies linear optimization by inscribing a circle in that pentagon.

Linear Optimization

The pentagon is a convex polytope formed by five points. It is an important restriction that the polytope be convex; if it were not, you would have to keep track of where the facets end, which certainly complicates matters.

Let the set of points $\{x^{(i)}\}$ i=1, …, n be ordered clockwise in the plane, and append the first point $x^{(1)}$ at the end so that the set consist of n+1 points.

Then you can compute the normalized normal vector to the ith facet as

$$v^{(i)} = \frac{1}{\left\|x^{(i+1)} - x^{(i)}\right\|} \begin{bmatrix} x_2^{(i+1)} - x_2^{(i)} \\ -(x_1^{(i+1)} - x_1^{(i)}) \end{bmatrix}$$

The distance from the circle center to the ith facet is $(x^{(i)} - x)^T v^{(i)}$, which means that the constraint on the radius of a circle inscribed in the polytope is

$$r \le (x^{(i)} - x)^T v^{(i)} \qquad , \mathrm{i}=1, \ldots, n \, .$$

MODEL DEFINITION

Following these definitions, you can formulate the problem as the linear optimization problem

$$\begin{array}{ll} \underset{x,\, r}{\text{minimize}} & -r \\ \text{subject to} & v^{(i)T} x + r \le v^{(i)T} x^{(i)} \quad i = 1, \ldots, n \end{array}$$

By comparing the problem to `optlin` on page 82 (and notice the similarity to the quadratic problem)

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & b_{\text{lb}} \le Ax \le b_{\text{ub}} \\ & x_{\text{lb}} \le x \le x_{\text{ub}} \end{aligned}$$

you can identify how to set up the problem. Call the parameter vector x instead of y because there are no constraints or objective functions this time, only matrices that you enter directly into the Opt structure:

$$x = \begin{bmatrix} r \\ x_1 \\ x_2 \end{bmatrix}, \; c = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & v_1^{(1)} & v_2^{(1)} \\ \vdots & \vdots & \vdots \\ 1 & v_1^{(5)} & v_2^{(5)} \end{bmatrix}, \text{ and } b_{\text{ub}} = \begin{bmatrix} v^{(1)T} x^{(1)} \\ \vdots \\ v^{(5)T} x^{(5)} \end{bmatrix}$$

RESULTS

The largest radius is 4.4801 and the circle center is (6.0589, 3.2429).

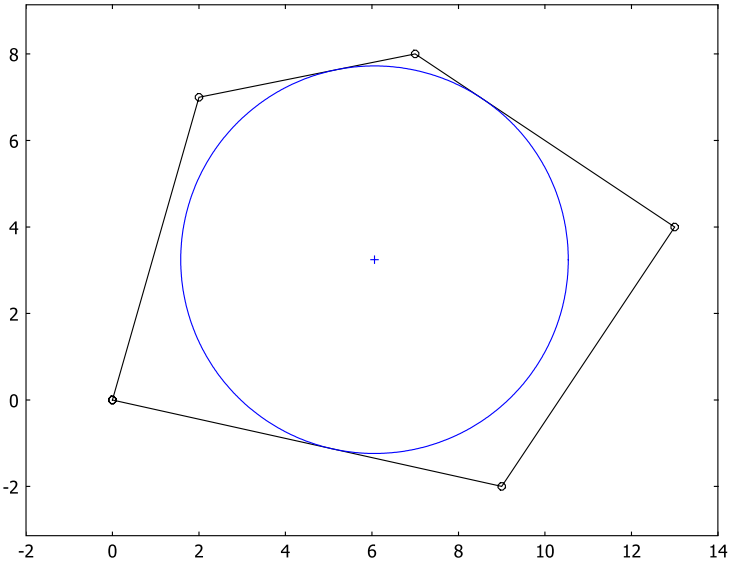


Figure 3-5: The largest circle inscribed in the pentagon.

Typing `help optprop` on the command line shows the solver properties and their default values, and you can see that both `feastol` and `opttol` are 1e-6. Looking at

`opt.sol.xinfo.state.lc` note that constraints 1, 3, and 4 are active. Also, from the tolerances and from the fact that optimality was reached, you know that the error is smaller than 1e-6. You can also easily check the actual error by typing in

```
opt.lc.ub'-opt.lc.A*opt.sol.x
```

Where the constraints are active, the distance is approximately 1e-14.

STEP-BY-STEP INSTRUCTIONS

1 Create a file `circlelin.m` or run this code from the command line:

```
clear opt;

% Points
X = [0 9 13 7 2 0;
     0 -2 4 8 7 0];
[m,n] = size(X);

% Normal vectors
V = 1./sqrt(sum(diff(X,1,2).^2,1)).*([0 1;-1 0]*diff(X,1,2));

% Objective
opt.obj.c = [-1,0,0];

% Linear constraints
opt.lc.A = [ones(1,n-1);V]';

% Bounds for linear constraints
opt.lc.lb = -Inf*ones(1,n-1);
opt.lc.ub = sum(V.*X(:,1:n-1));

% Initial guess
opt.init.x = [1;20;40];

% Solve
opt.sol = optlin(opt);

% Postprocessing
fprintf([opt.sol.msg '\n'])
r = opt.sol.x(1)
x = opt.sol.x(2:3)

% Points
figure
plot([X(1,:) X(1,1)],[X(2,:) X(2,1)],'k')
hold on
plot([X(1,:) X(1,1)],[X(2,:) X(2,1)],'ko')

% Circle
```

```
theta = linspace(0,2*pi,1000);
plot(x(1),x(2),'+')
plot(x(1)+r*cos(theta),x(2)+r*sin(theta))
axis equal
```

The Rosenbrock Function

The Rosenbrock function is a common test bed for solving unconstrained optimization problems:

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

This function is sometimes called a "banana function" due to the shape of the contours (see Figure 3-6 below). This flat shape with a narrow valley makes a steepest-descent approach converge slowly toward the minimum value of 0 at (1, 1), which is easy to see from the function itself. To compute the minimum using `optnm`:

1. Create a COMSOL Script function `rosenbrock.m` that computes the value of the Rosenbrock function:

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

2. Compute the minimum of the function and the values of x_1 and x_2 at the minimum, using 0 as the starting guess for both variables:

```
[x,f]=optnm('rosenbrock',[0 0]);
```

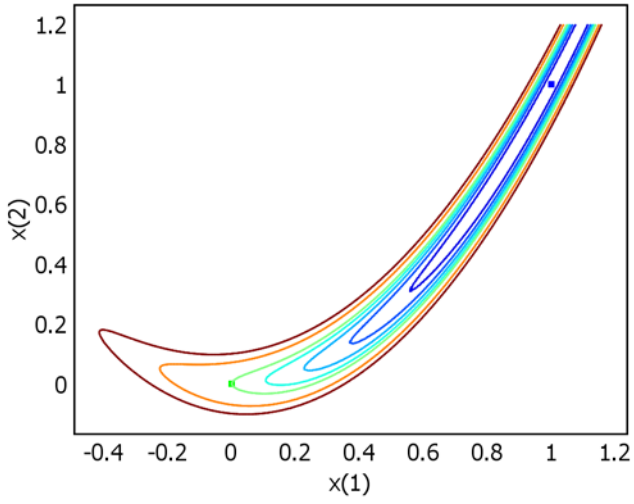


Figure 3-6: Contour plot of the Rosenbrock function, where the lower (green) dot indicates the starting point and the upper (blue) dot indicates the optimal solution.

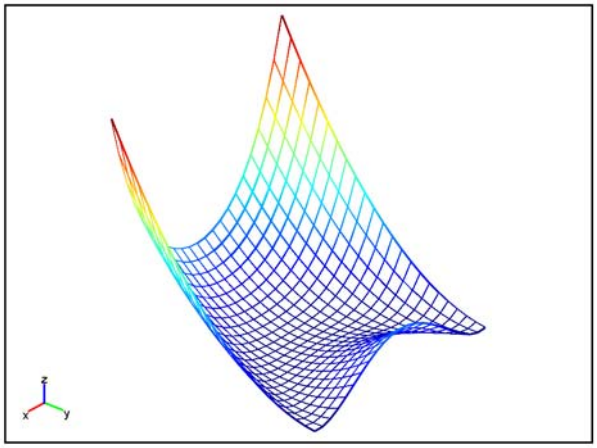


Figure 3-7: The Rosenbrock function where $z = f(x, y) = (1-x)^2+100(y-x^2)^2$.

4

Multiphysics Optimization

This chapter shows how to combine the Optimization Lab with COMSOL Multiphysics to optimize physics models. Thus the following examples require both software packages.

Overview

By combining the Optimization Lab with COMSOL Multiphysics it is possible to perform various optimizations using parameters from a physics model. To illustrate this concept, the two examples in this chapter include:

- a diode model, which shows how the Optimization Lab can extract SPICE parameters for the equivalent circuit of a diode modeled in COMSOL Multiphysics
- a spinning-gear model, which uses the Optimization lab to locate the separation frequency of a spinning gear.

In general, optimization of models from COMSOL Multiphysics includes the following steps:

1. Set up the model in COMSOL Multiphysics.
2. Whenever possible, make sure to define the parameters you intend to optimize in the **Options>Constants** dialog box. This makes them readily available to pass back and forth between the Optimization Lab routines and the COMSOL Multiphysics solvers (the `fem.const` field).
3. Export the FEM structure to the Script prompt.
4. Write a script or function that defines the problem in the Opt structure.
5. Write the callback routines, including the FEM structure as an additional input argument, and make sure to update the `fem.const` field at each iteration.
6. Call `optim` to solve the problem using the `'param'` option to pass along the FEM structure and any additional arguments to the callback routines. Use the `persistent` (or `global`) modifier to retain variables across calls.

SPICE Parameter Extraction for a Semiconductor Diode

This model shows how the Optimization Lab can extract SPICE parameters from the model of a semiconductor diode for use in an equivalent circuit.

Introduction

In the design of a semiconductor device, it is often desirable to develop a compact model to use when analyzing its behavior in larger systems. SPICE models are compact descriptions of electronic circuits, where a set of SPICE parameters determines the device's behavior in static, transient, and time-harmonic analysis. In developing such a compact model, the extraction of the SPICE parameters usually requires several different characteristics that show how the device behaves for a range of operating conditions. The characteristics can either be the result of measurements or come from simulations of a more-detailed reference model.

This example reviews the development of a compact model for a semiconductor diode. The reference finite-element model is "Semiconductor Diode" on page 442 of the *COMSOL Multiphysics Model Library*. From this model it is possible to extract the forward characteristics where the diode is biased from 0 V up to 1.5 V. Six SPICE parameters control the forward characteristics of the compact model. This sets a lower limit on the number of reference data required to extract the compact model.

Model Definition

SEMICONDUCTOR MODEL

The previously mentioned example in the Model Library solves for a forward bias of only up to 1 V. This level is not high enough to extract all the needed parameters, especially the resistor parameter, so the characteristics must go a bit further. As a result, the first part of this example solves the extra steps for the semiconductor diode with a bias reaching 1.5 V.

EQUIVALENT DIODE CIRCUIT

A diode is an electrical rectifier that conducts current for positive voltages and insulates for negative voltages. The current-voltage characteristics (or IV characteristics) for an ideal semiconductor diode follow the relationship

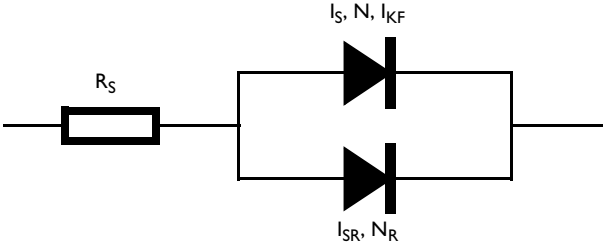
$$I = I_S\left(e^{\frac{V}{V_T}} - 1\right)$$

where V_T is the thermal voltage. The compact model of a real diode actually consists of two ideal diodes in parallel and a resistor in series (see nearby figure). Furthermore, a parameter called the ideality factor, N, is introduced into the ideal diode equation so that

$$I = I_S\left(e^{\frac{V}{NV_T}} - 1\right).$$

One of the ideal diodes also has a parameter for high-level injection, which is a change in its characteristics for high currents.

The following image shows the complete equivalent circuit for a real diode with the SPICE parameters next to each device.



You cannot express the compact model as an explicit relationship between its current and voltage, so to obtain its characteristics, it is necessary to solve an implicit nonlinear problem. An easy way to set up this equivalent circuit is to use the `spiceimport` script available in the AC/DC Module. With a simple circuit file, the script automatically generates ODE expressions that model the equivalent diode circuit. In this case the ODE representation does not involve any time derivatives, so from the mathematical viewpoint you have the special case of static, algebraic equations. The solve command `femstatic` can then solve the ODEs and give the current for a predefined list of

voltages. The SPICE parameters are given as constants in the equivalent circuit model according to the following table:

PARAMETER	CONSTANT	DESCRIPTION
I_S	`IS_ID_D1_cir`	Saturation current
N	`N_ID_D1_cir`	Ideality factor
I_{KF}	`IKF_ID_D1_cir`	High-injection knee current
I_{SR}	`ISR_IR_D1_cir`	Recombination current
N_R	`NR_IR_D1_cir`	Ideality factor for recombination current
R_S	`R_RS_D1_cir`	Series resistance

THE OPTIMIZATION

The Optimization Lab searches for values for the six SPICE parameters shown in the equivalent circuit such that the IV characteristics from the ODE circuit simulations match the IV characteristics extracted from the reference finite-element diode model. This is an optimization problem of six unknowns that enter the objective function in a nonlinear fashion. The initial guess for the parameters are crucial to reduce the search time.

It is also necessary to define a couple of constraints in the search because otherwise the two diodes in parallel become hard to distinguish. These diodes have a physical interpretation, where the upper one controls the main diode characteristics. The ideality factor of the main diode should lie close to one. The lower diode is responsible for the recombination effects in the semiconductor, and that effect shows an ideality factor close to 2. Therefore, a proper constraint is to force the ideality factor N to be less than the ideality factor N_R. A high ideality factor also results in a high saturation current, I_{SR}, so the same constraint can be used for the saturation currents. That is,

$$N < N_R$$
$$I_S < I_{SR}$$

In addition, it is also necessary to set lower and upper bounds to all parameters and to use the logarithmic values for the parameters, I_S, I_{SR}, R_S, and I_{KF} because they usually span several orders of magnitude.

To summarize, the optimization problem is

$$
\begin{array}{lll}
\underset{x}{\text{minimize}} & \frac{1}{2}\sum_i F_i(V_i, x)^2 & \text{Objective function} \\
\text{subject to} & I_S \le I_{SR} & \text{Linear constraints} \\
& N \le N_R & \\
& \begin{bmatrix} -20 \\ 1 \\ -20 \\ 1 \\ 2 \\ -7 \end{bmatrix} \le x \le \begin{bmatrix} -10 \\ 2 \\ -10 \\ 2 \\ 6 \\ -4 \end{bmatrix} & \text{Parameter bounds} \\
& I_{\text{cir}} = I_{\text{cir}}[V, x] & \text{Algebraic equations}
\end{array}
$$

where

$$x = (\log(I_S), N, \log(I_{SR}), N_R, \log(R_S), \log(I_{KF}))^T .$$

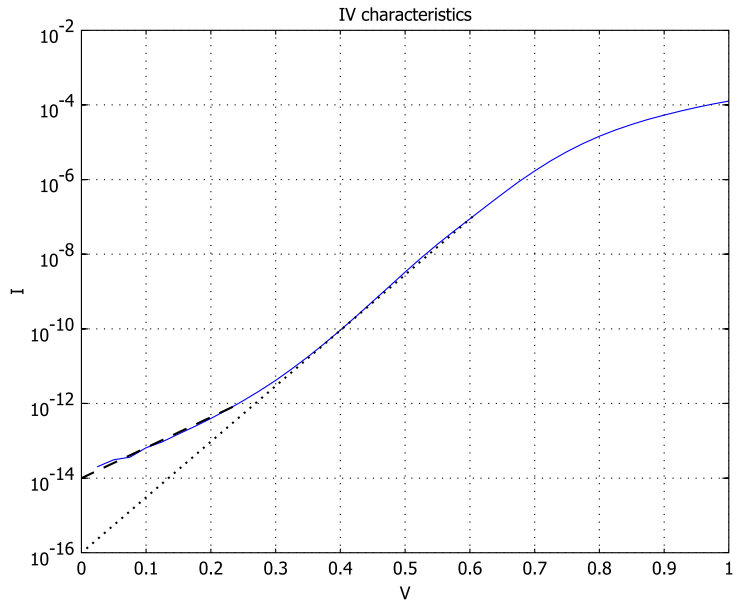
The objective is to determine the SPICE parameters (x) so that the equivalent circuit matches the original diode with respect to the IV characteristics in a least-squares sense. Again, using a logarithmic scale:

$$F_i(x) = \log_{10}(I(V_i)) - \log_{10}(I_{\text{cir}}[V_i, x]) .$$

Because I_{cir} depends implicitly on the SPICE parameters, the system of algebraic equations describing the IV relationship of the equivalent circuit must be solved by `femstatic` for each voltage in the range of the characteristic you wish to match. This must be done every time the optimization solver evaluates the objective function with updated SPICE parameters.

Initial Guess

The initial guess for the parameter values is very important in order to speed up the search, and in some cases also to get convergence. You can easily extract proper parameter values directly from the IV characteristics of the semiconductor diode. In the following figure you can see some straight lines that represents the diode equations presented earlier, one for each diode in the equivalent circuit.



The intersections between these lines and the y-axis serve as good initial guesses for the parameters I_S and I_{SR}. Next, the ideality factors usually have values close to one and two, so use those values as initial guess. The parameter I_{KF} determines where the curve has its knee for large currents, so the current value where this appears is suitable as an initial guess. The final parameter, R_S, also controls the flat region of the curve, so approximately the maximum voltage divided by the maximum current is a good guess. The initial parameters from this simple analysis are summarized in this table:

PARAMETER	VALUE	INITIAL VALUE IN X	DESCRIPTION
I_S	10^{-16} A/m	-16	Saturation current
N	1	1	Ideality factor
I_{SR}	10^{-14} A/m	-14	Recombination current
N_R	2	2	Ideality factor for recombination current
R_S	10^{4} Ωm	4	Series resistance
I_{KF}	10^{-6} A/m	-6	High injection knee current

The unit for current is A/m because the circuit is compared to a 2D simulation in SI units.

Results and Discussion

You export the finite element model from COMSOL Multiphysics to the COMSOL Script command line, where the optimization takes place using the Optimization Lab. Fitting the parameters takes a few minutes on a modern PC, and the resulting compact model characteristics appear as crosses in the next figure, which plots the IV characteristics from the finite element reference model as a solid line. The agreement is quite good over the entire voltage range up to 1.5 V.

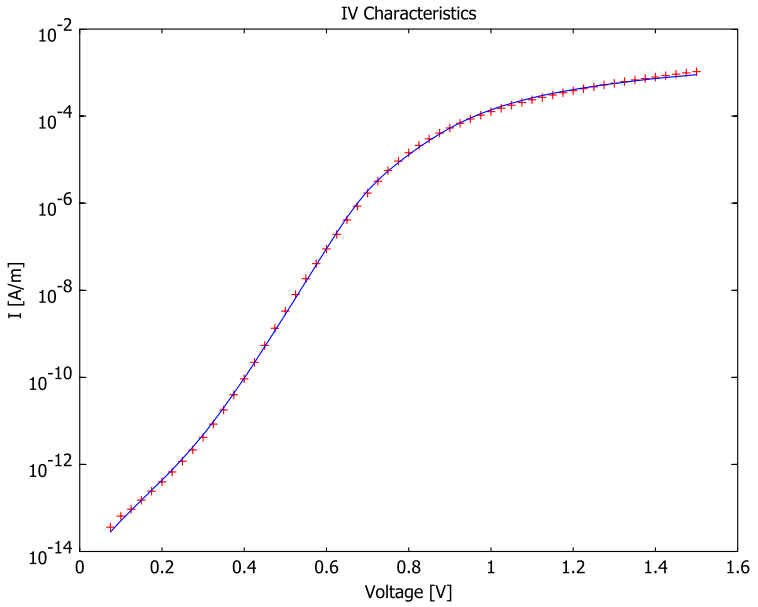


Figure 4-1: The crosses show the IV characteristics of the equivalent circuit, the blue (solid) line is that from the COMSOL Multiphysics Model Library model semiconductor_diode.

PARAMETER	OPTIMIZED VALUE	DESCRIPTION
I_S	$4.9 \cdot 10^{-17}$ A/m	Saturation current
N	1.1	Ideality factor
I_{SR}	$8.6 \cdot 10^{-15}$ A/m	Recombination current
N_R	2.0	Ideality factor for recombination current
R_S	$5.21 \cdot 10^{2}$ Ωm	Series resistance
I_{KF}	$1.7 \cdot 10^{-6}$ A/m	High injection knee current

Modeling in the Graphical User Interface

First start COMSOL Multiphysics and open the semiconductor diode example and export it to COMSOL Script.

1. In the Model Navigator, click the **Model Library** tab.
2. Browse to the model **COMSOL Multiphysics>Semiconductor Devices>semiconductor diode**.
3. Select it and click **OK**.
4. Export the model to COMSOL Script by choosing **File>Export>Export FEM structure as 'fem'**. COMSOL Script opens automatically.

Modeling in COMSOL Script

If you followed the previous instructions, the semiconductor diode model should now be present in the workspace as the variable `fem`. The optimization procedure involves solving the equivalent circuit model for the list of voltages used in the reference model, and then comparing the values of the current from the circuit model with those from the reference model.

1. Start by creating a function called `diode_obj.m` by entering the following lines of code:

```
function F = diode_obj(x,fem,V,I,logind)

global ITER

% Update iterations counter
if isempty(ITER)==0
  ITER = ITER+1;
end

% Default parameter values
values = [-16 1.0 -13 2 4 -5];

% SPICE parameter map
params = {'IS_ID_D1_cir','N_ID_D1_cir','ISR_IR_D1_cir', ...
          'NR_IR_D1_cir','R_RS_D1_cir','IKF_ID_D1_cir'};
params_name = {'IS','N','ISR','NR','RS','IKF'};

% Replace values
values(1:length(x)) = x;

% Replace logarithmic values
values(logind) = 10.^(values(logind));
```

```
% Update constant list in fem
for ind=1:length(values)
    ic = 2*strmatch(params{ind},fem.const(1:2:end),'exact');
    fem.const{ic} = values(ind);
end

% Output values on screen
if (mod(ITER,10)==0)
  fprintf('It.  IS     N     ISR      NR        RS        IKF \n')
end
fprintf('%3d  %2.6e  %2.6f  %2.6e  %2.6f  %e  %2.6e \n', ...
       ITER,values(1),values(2),values(3),values(4),values(5),...
       values(6))

em.xmesh = meshextend(fem);

% Solve circuit model
fem.sol = femstatic(fem,'pname','value_VIN_cir','plist',V);

% Evaluate the current of equivalent circuit
data = postglobaleval(fem,{'-I_VIN_cir'});
I_cir = data.y;

% Compare with real data in a logarithmic scale
F = log10(I)-log10(I_cir);
```

The arguments to this function are explained in the following table.

ARGUMENT	DESCRIPTION
`x`	Array with the current parameter values in the order specified in the `params` cell array
`fem`	FEM structure containing the equivalent circuit model
`V`	List of voltages to solve for
`I`	Values of current from the semiconductor diode model
`logind`	Indices to parameters in `x` that are the logarithm of the actual value in the equivalent circuit model

The function `diode_obj.m` first defines some default values that it uses if you do not search for all parameters. Next the code replaces the parameters in the FEM structure with the supplied parameters in the argument x. Before passing the parameter to the FEM structure, the function converts the variables that contain the logarithm of the true parameter into the true parameter value. After the `for` loop, it prints the parameter values to make it easy to follow the search. Then it updates the FEM structure with the new values and calculates the solution. Finally, it extracts

the IV characteristics of the equivalent circuit and compares them to the supplied IV characteristics from the reference model. Note that the code uses the logarithm of the current in the comparison because the highest and lowest values differ by several orders of magnitude.

2 Next either run the following code from the command line or from an m-file `diode.m`. A detailed explanation of each step follows.

Note: The following example requires the SPICE import feature of the *AC/DC Module*. Replace the line `fem = spiceimport(diode_cir_path)` below with `flload diode_spicefem` if you do not have this module. This will load the pre-saved FEM structure into the workspace.

```
global ITER

% Initialize iteration counter
ITER = 0;

% Solve up to 1.5 V to see more resistive effects
fprintf('Solving diode model from 1.0 to 1.5 V to see more resistive
effects...\n')
sol = femstatic(fem,'init',fem.sol.u(:,end),'pname','Va',...
                    'plist',1.025:0.025:1.5);

% Concatenate solutions
fprintf('Concatenating solutions 0 to 1.5 V.\n')
sol = femsol(cat(2,fem.sol.u,sol.u),'plist',...
             cat(2,fem.sol.plist,sol.plist));

% Extract the IV-characteristics (avoid low V)
IV = postglobaleval(fem,{'Ic'},'u',sol,...
                    'solnum',4:length(sol.plist));

% Create equivalent circuit
fprintf('Setting up equivalent circuit equations from SPICE model
diode.cir...\n')
clear fem;
diode_cir_path = which('diode.cir');
fem = spiceimport(diode_cir_path);

% Optimization
fprintf('Minimizing 1/2*|log(I)-log(I_cir)|^2 by fitting the SPICE
parameters IS, N, ISR, NR, RS, IKF...\n')
% Start parameter extraction
clear opt;
```

```
% Objective function
opt.obj.F = 'diode_obj';

% Parameters: log(IS), N, log(ISR), NR, log(RS), log(IKF)
% Parameter bounds
opt.bc.lb = [-20 ; 1 ; -20 ; 1 ; 2 ; -7];
opt.bc.ub = [-10 ; 2 ; -10 ; 2 ; 6 ; -4];

% Indices to logarithmic parameters: IS, ISR, RS, IKF
logind = [1 3 5 6];

% Linear constraints: IS<ISR, N<NR
opt.lc.A = [-1 0 1 0 0 0 ; 0 -1 0 1 0 0];
opt.lc.lb = [0 ; 0];

% Initial guess
opt.init.x = [-16 1 -14 2 4 -6];

% Find parameters
opt.sol = optnlinlsq(opt,'opttol',1e-2,...
                     'param',{fem,IV.x,IV.y,logind});

% Plot parameters
I_log = -diode_obj(opt.sol.x,fem,IV.x,ones(size(IV.x)),logind);
semilogy(IV.x,IV.y,'r+',IV.x,10.^(I_log),'b');
```

Now take a closer look at the steps in the optimization. Start by solving the extra steps up to 1.5 V with the following command:

```
sol = femstatic(fem,'init',fem.sol.u(:,end),'pname','Va',...
                'plist',1.025:0.025:1.5);
```

Then merge the solutions so you have one solution where the parameter `Va` ranges from 0 to 1.5:

```
sol = femsol(cat(2,fem.sol.u,sol.u),'plist',...
             cat(2,fem.sol.plist,sol.plist));
```

From this solution you extract the IV characteristics:

```
IV = postglobaleval(fem,{'Ic'},'u',sol,'solnum',...
                    4:length(sol.plist));
```

The voltage and current values are now stored in the fields `IV.x` and `IV.y`, respectively. Note that the lowest voltages are not included because they contain singularities and noisy results. You no longer need the semiconductor diode model, so you can use the same variable for the equivalent circuit. The following commands produce an FEM structure containing ODEs and expressions that models the circuit:

```
clear fem;
diode_cir_path = which('diode.cir');
fem = spiceimport(diode_cir_path);
```

The file `diode.cir` is the SPICE netlist for a diode and contains the following text:

```
Vin 1 0 1
D1 1 0 diode
.MODEL diode D IS=1e-13 ISR=1e-10 N=1.1 NR=2.0 RS=1e1 IKF=1e-5
```

The first line defines a voltage of 1 V between nodes 1 and 0. The second line specifies that you place a diode between these two nodes, and the final line specifies some diode parameters for the SPICE model. These parameters are the ones that you optimize later on, and they are available as constants in the created FEM structure.

The following code sets up an optimization structure for the objective function:

```
clear opt;
opt.obj.F = 'diode_obj';
```

Then you set up the bounds for all the parameters. Note that some parameters are in log scale:

```
opt.bc.lb = [-20 ; 1 ; -20 ; 1 ; 2 ; -7];
opt.bc.ub = [-10 ; 2 ; -10 ; 2 ; 6 ; -4];
```

The variable in log scale are specified by the `logind` variable that contains indices to the parameter in the `x` argument to the objective function:

```
logind = [1 3 5 6];
```

You also need some linear constraints to separate the parameter range for the ideality factor and saturation currents:

```
opt.lc.A = [-1 0 1 0 0 0 ; 0 -1 0 1 0 0];
opt.lc.lb = [0 ; 0];
```

Before you start the parameter search, it is necessary to specify the initial guess that you can estimate from the IV characteristics of the semiconductor diode:

```
opt.init.x = [-16 1 -14 2 4 -6];
```

The setup is now complete, and the next step is to search for the optimum parameters in a least-square sense. The following command starts the search:

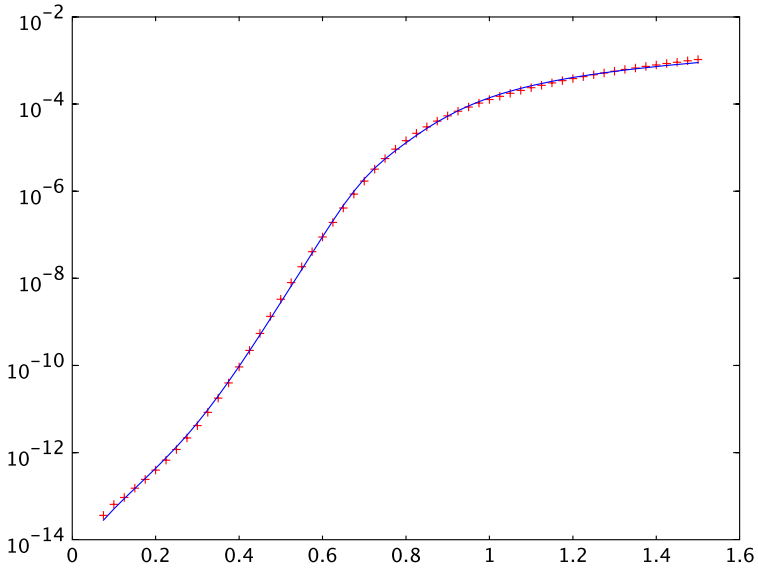
```
opt.sol =  optnlinlsq(opt,'opttol',1e-2,...
                      'param',{fem,IV.x,IV.y,logind});
```

The `optol` option specifies the tolerance, and the `param` option lists the extra argument that you pass to the objective function. The routine prints out a long list of parameter values during the search process. When the search is finished, you can

plot the IV characteristics from both the semiconductor diode and the equivalent circuit in the same figure with the following commands:

```
I_log = -diode_obj(opt.sol.x,fem,IV.x,ones(size(IV.x)),logind);
semilogy(IV.x,IV.y,'r+',IV.x,10.^(I_log),'b');
```

You should now see the following plot.



Spinning Gear

Introduction

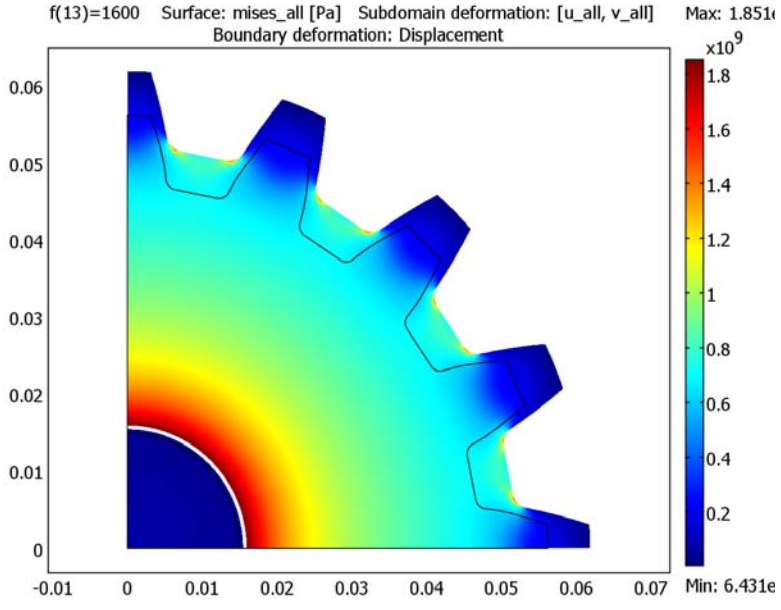


Figure 4-2: Stresses and deformation at 1600 Hz.

One way to fasten a gear to a shaft is by thermal interference. In the preparation of the assembly, the shaft diameter is oversized and the gear is thermally expanded in a heat-treating oven. At an appropriate state of expansion, the gear is removed from the oven, slid onto the shaft, and allowed to cool. As the gear's temperature drops, that component shrinks and comes into contact with the shaft before it can reach its original shape. From this point on, additional gear shrinkage results in hoop stresses in the gear as well as normal compression of the shaft. At thermal equilibrium, an intimate bond between the two components is reached.

Such an assembly can operate safely in many situations. However, there are operating conditions under which the fastening stresses become insufficient—for instance, when spinning the assembly at high rpm.

The goal of this analysis is to determine the critical spinning frequency at which the gear and shaft separate.

Model Definition

The model computations consist of two initial steps in COMSOL Multiphysics: one for the thermal interference fit, and one for spinning the shaft-gear assembly (for details see Chapter 3, "Spinning Gear," on page 80 of the *Structural Mechanics Module Model Library*.)

The resulting physics model describes the spinning gear at any frequency. The optimization problem is the inverse: find the frequency at which the gear and shaft separate. The optimization equation is

$$\underset{x}{\text{minimize}} \qquad \texttt{intdisp}(x)$$

where **intdisp** is the displacement integral (a COMSOL Multiphysics integration coupling variable).

The optimization routine does not need to know the details of the physics model, it simply queries for the distance between shaft and gear at any given frequency.

Reference

1. http://claymore.engineer.gvsu.edu/~schmitte/assign5.html

Model Library path:
`Structural_Mechanics_Module/Automotive_Applications/spinning_gear`

Note: The following section requires COMSOL Multiphysics and the Structural Mechanics Module.

Modeling using the Graphical User Interface

MODEL NAVIGATOR

First open the spinning gear model and export it to COMSOL Script.

1 In the Model Navigator click the **Model Library** tab.

2 Browse to the model **COMSOL Multiphysics>Structural Mechanics Module>Automotive Applications>spinning gear**.

3 Select it and click **OK**.

4 Export the model to COMSOL Script by choosing **File>Export>Export FEM structure as 'fem'**. COMSOL Script opens automatically.

Modeling using COMSOL Script

Now you can determine the separation frequency. The spinning gear model should now be present in the workspace as the variable `fem`.

Note: All the functions in the following discussion are available in COMSOL Script.

---

1 Start by creating a function called `spinning_gear_obj.m`. It defines the value of the objective at a given frequency.

```
function f = spinning_gear_obj(x,fem,arclength)

persistent ITER;
global SPG_ITER

% The solver is calling this function for the first time
if isequal(optgetstatus,1)
  ITER = 1;
else
  ITER = ITER+1;
end

% Solve the problem at frequency x
fem.sol=femlin(fem, ...
               'solcomp',{'u2','v3','v2','u3'}, ...
               'outcomp',{'u2','v3','u','v2','u3','v'}, ...
               'pname','f', ...
               'plist',x, ...
               'nonlin','off');

% Pick up displacement integral
pd = posteval(fem,'int_disp','edim',0,'dl',1);

% Normalize
f = pd.d/arclength;

% Print progress
if (mod(ITER-1,10)==0)
  fprintf('Iteration   Frequency     Displacement \n')
end
```

```
fprintf('%3d         %8.3f       %8.4e\n',ITER,x,f)
SPG_ITER(ITER)=x;
```

2 Create a function called `spinning_gear.m`. This is the main function that defines the optimization problem and calls the solver.

```
function [opt,fem] = spinning_gear(fem, arclength)

global SPG_ITER;

clear opt;
opt.obj.f = 'spinning_gear_obj';

% Start from first existing solution
% (e.g. opt.init.x = 1000;)
opt.init.x = fem.sol.plist(1);

% Solution history
SPG_ITER = 0;

% Solve
opt.sol = optim(opt,'param',{fem,arclength},'opttol',1e-8);

freq = opt.sol.x;
fprintf('Separation Frequency = %8.3f [Hz] \n',freq);

% Plots
figure, plot(1:numel(SPG_ITER),SPG_ITER,'-o')
ylim([800,3200]);
title('Convergence')
xlabel('Iterations')
ylabel('frequency f [Hz]')
grid

% Plot solution
figure;
postplot(fem, ...
         'tridata',{'mises_all','cont','internal','unit',...
         'N/m^2'}, ...
         'trimap','jet(1024)', ...
         'deformsub',{'u_all','v_all'}, ...
         'solnum',1, ...
         'title',['f =' sprintf('%8.3f',freq),...
         '   Surface: mises_all [N/m^2]   Subdomain',...
         ' deformation: [u_all, v_all] [m]   Boundary',...
         'deformation: Displacement [m]']);

figure;
postplot(fem, ...
         'tridata',{'mises_all','cont','internal'}, ...
         'triz','mises_all', ...
         'trimap','jet(1024)', ...
         'solnum',1, ...
```

```
'title',['f =' sprintf('%8.3f',freq),...
' Surface: mises_all, Height: mises_all'], ...
'refine',2, ...
'grid','on', ...
'camlight','on');
```

For the initial solution, this script uses the first solution among those in the physics model although it is not the best one. This is for demonstration purposes only; it is always advisable to start with the best available solution. (In this case, the initial sweep range happens to have hit very close to the optimum at a frequency of 1550 Hz; this would normally be the solution from which to begin optimization.)

3 To run the optimization call the `spinning_gear` function:

```
[opt,fem]=spinning_gear(fem,(2*pi*0.015)/4)
```

This prints the frequency and distance during the iterations and returns an optimal solution of 1550.185 Hz. (The second argument, `(2*pi*0.015)/4`, is the length of the circle segment and is used in `spinning_gear_obj` to normalize the displacement integral.)

Note that the first few iterations indicate that the solver is preparing to estimate the gradient through finite differences. (During this step, the solver determines the sparsity pattern for the Jacobian and identifies the constant elements automatically):

```
Iteration   Frequency     Displacement
  1         1000.000       7.2929e-005
  2          242.516       1.2185e-004
  3         1666.588       1.9463e-005
  4         1803.304       4.4120e-005
  5          485.031       1.1268e-004
  6         4999.764       1.1744e-003
  7         7213.217       2.5795e-003
```

In problems where the derivatives are known, this step is not included.

Because this problem is unconstrained, you could also solve it using the `optnm` function. In that case, replace the line

```
opt.sol = optim(opt,'param',fem,'opttol',1e-6);
```

with

```
opt.sol = optnm(opt,'param',fem);
```

You also need to remove `optgetstatus` call from `spinning_gear_obj`, either by checking if `ITER` is empty or by making `ITER` a global variable and resetting it before each call to the solver.

Note that `spinning_gear` outputs three plots: the separation frequency during optimization (Figure 4-3), and the stress distribution at separation as a 2D (Figure 4-4) and 3D plot (Figure 4-5).

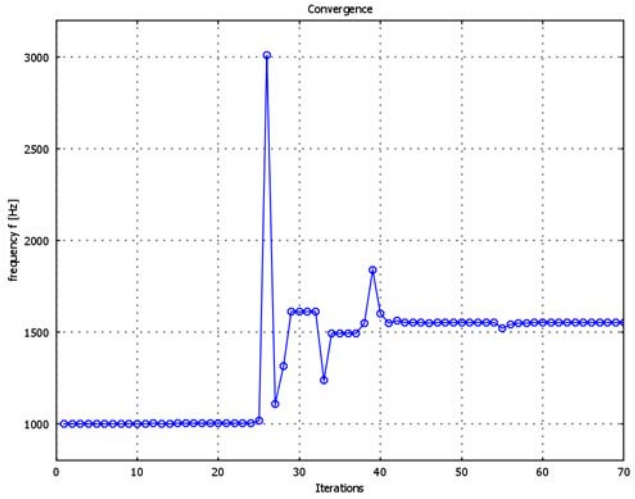


Figure 4-3: Convergence of the separation frequency during optimization.

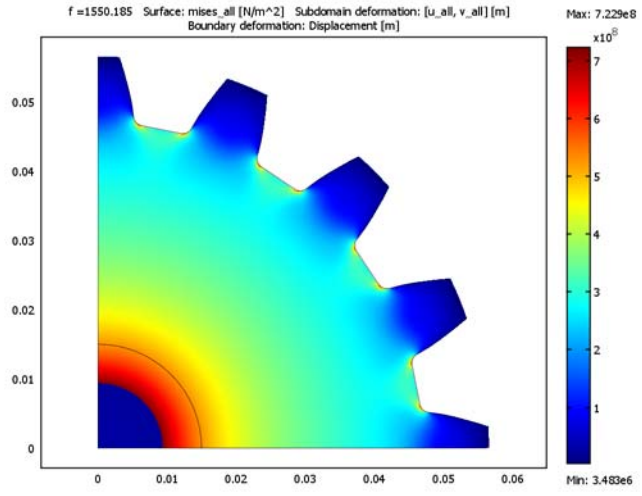


Figure 4-4: Stress distribution at separation.

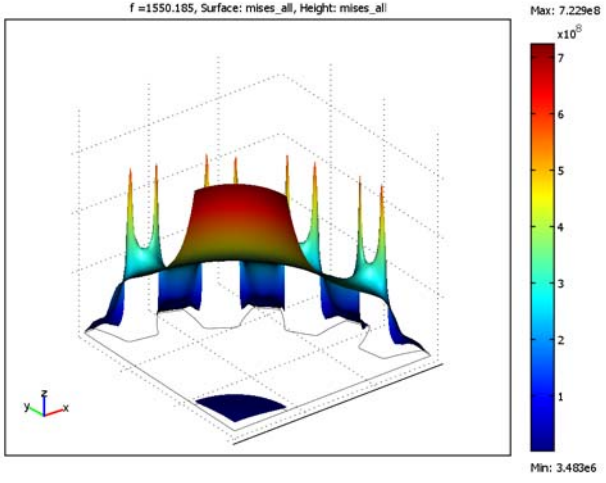


Figure 4-5: Stress distribution at separation.

EXPLOITING MODEL SYMMETRY

To reduce the required computation time, you can exploit the remaining model symmetries instead of solving the complete model as above. Figure 4-6 illustrates the stresses superimposed on the minimal model at the separation frequency.

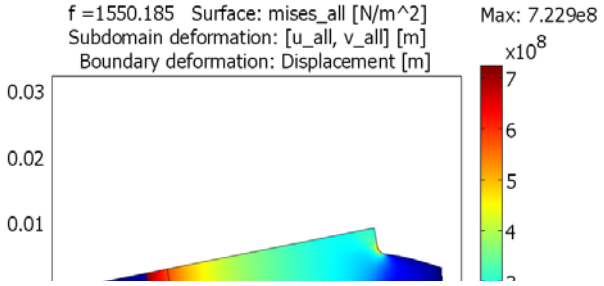


Figure 4-6: Von Mises stresses superimposed on the minimal model.

To run the example on the minimal model, load the modified FEM structure, then proceed as before.

1. At the Script prompt, type:

   ```
   fem = cl_gear_minimal(1000);
   ```

   This solves the reduced physics problem at an initial frequency of 1000 Hz.

2. Solve the optimization problem as before, but normalizing with the reduced segment length (1/32 of the full circle). Type:

   ```
   [opt,fem]=spinning_gear(fem,(2*pi*0.015)/32);
   ```

Figure 4-7 shows the resulting stress distribution plot.

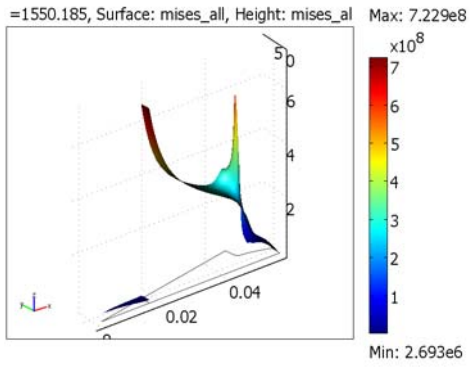


Figure 4-7: Stress distribution at separation.

5

Command Reference

Summary of Commands

Commands Grouped by Function

General Optimization Functions

FUNCTION	PURPOSE
`optim`	Solve a general optimization problem

Minimization Functions

FUNCTION	PURPOSE
`optlin`	Solve a linear optimization problem
`optnlin`	Solve a nonlinear optimization problem
`optnm`	Solve an unconstrained nonlinear optimization problem using the Nelder-Mead simplex algorithm
`optquad`	Solve a quadratic optimization problem

Least-Squares Functions

FUNCTION	PURPOSE
`optlinlsq`	Solve a linear least-squares problem
`optnlinlsq`	Solve a nonlinear least-squares problem

Solver Properties

FUNCTION	PURPOSE
`optprop`	Common solver property/value pairs for the linear and quadratic solvers in the Optimization Lab
`optpropnlin`	Common solver property/value pairs for the nonlinear solvers in the Optimization Lab

Miscellaneous Functions

FUNCTION	PURPOSE
`optgetstatus`	Check solver status during callback
`optsetstatus`	Set solver status during callback

Purpose

Get solver status during callback.

Syntax

```
s = optgetstatus
```

Description

`s = optgetstatus` is intended for use in the callback functions in the Optimization Lab and returns specific information about the call.

If `s = 0`, there is nothing special about the current call.

If `s = 1`, SNOPT/SQOPT is calling your function(s) for the first time.

If `s ≥ 2`, SNOPT/SQOPT is calling your function(s) for the last time.

For more information see the documentation for the SQOPT and SNOPT packages.

See Also

`optsetstatus`

Purpose

Solve a general optimization problem.

Syntax

```
opt.sol = optim(opt,...)
opt.sol = optim(opt,options)
```

Description

`optim` solves an optimization problem by examining the fields in the Opt structure and then calling the most appropriate solver among `optlin`, `optquad`, `optnlin`, `optlinlsq`, and `optnlinlsq`. For example, a general nonlinear objective function or a quadratic objective function with nonlinear constraints results in `optnlin` being called.

`optim` uses the problem formulation

$$\begin{aligned} &\min \quad f(x) \\ &b_{\mathrm{lb}} \le Ax \le b_{\mathrm{ub}} \\ &d_{\mathrm{lb}} \le c(x) \le d_{\mathrm{ub}} \\ &x_{\mathrm{lb}} \le x \le x_{\mathrm{ub}} \end{aligned}$$

where $f(x)$ can be one the following forms:

TABLE 5-1: THE OPT.OBJ SUBSTRUCTURE

OBJ.FORM	OBJECTIVE FUNCTION min $f(x)$	FIELDS	INTERPRETATION
`'lin'`	$f(x) = c^T x$	`obj.c`	Coefficient vector for the linear term of the objective function (numeric vector)
`'quad'`	$f(x) = \frac{1}{2} x^T Hx + c^T x$	obj.H	Coefficient matrix for the quadratic terms of the objective function (symmetric matrix or function returning Hx), default zero
		obj.c	Coefficient vector for the linear term of the objective function (numeric vector), default vector of zeros
`'nlin'`	$f(x)$	`obj.f`	Objective function (function)
		`obj.g`	(optional) Gradient of objective function (function)
		`obj.ptrn`	(optional) Gradient sparsity pattern (matrix)

TABLE 5-1: THE OPT.OBJ SUBSTRUCTURE

OBJ.FORM	OBJECTIVE FUNCTION min $f(x)$	FIELDS	INTERPRETATION
`'linlsq'`	$f(x) = \frac{1}{2}\|Cx - d\|^2$	`obj.C`	Objective function matrix (matrix)
		`obj.d`	Objective function vector (numeric vector), default vector of zeros
`'nlinlsq'`	$f(x) = \frac{1}{2}\|F(x)\|^2$ $= \frac{1}{2}\sum_i F_i(x)^2$	`obj.F`	Objective function (function). Note that the function should return the vector of F_i to be summed, not the sum of squares
		`obj.J`	(optional) Jacobian of F (matrix)
		`obj.ptrn`	(optional) Jacobian sparsity pattern (matrix)

Aside from the objective function, the Opt structure can contain the following (optional) fields:

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.lc.A`	A	Linear constraints (matrix)
`opt.lc.lb`	b_{lb}	Lower bounds for linear constraints (vector or scalar), default `-Inf`
`opt.lc.ub`	b_{ub}	Upper bounds for linear constraints (vector or scalar), default `Inf`
`opt.nc.c`	c	Nonlinear constraints (function)
`opt.nc.J`		Jacobian of nonlinear constraints (function)
`opt.nc.ptrn`		Constraint Jacobian sparsity pattern (matrix)
`opt.nc.lb`	d_{lb}	Lower bounds for nonlinear constraints (vector or scalar), default `-Inf`
`opt.nc.ub`	d_{ub}	Upper bounds for nonlinear constraints (vector or scalar), default `Inf`
`opt.bc.lb`	x_{lb}	Lower bounds for variables (vector or scalar), default `-Inf`
`opt.bc.ub`	x_{ub}	Upper bounds for variables (vector or scalar), default `Inf`

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.init.x`		Initial guess (vector), default vector of zeros
`opt.init.y`		Initial guess for Lagrange multipliers of the linear and nonlinear constraints (structure), default zero for all multipliers
`opt.sol.x`		Solution (vector)
`opt.sol.eval`		Substructure containing value of objective function and constraints, if any
`opt.sol.y`		Lagrange multipliers in solution (structure)
`opt.sol.xinfo`		Information about the solution (structure)
`opt.sol.exit`		Exit condition (scalar). `exit` is 1 for successful completion, 0 otherwise.
`opt.sol.algorithm`		Indicates which function was ultimately called. Can be either `'optlin'`, `'optquad'`, `'optnlin'`, `'optlinlsq'`, or `'optnlinlsq'`

Functions can be given either as strings, denoting the function name, or as inline functions. By specifying the same function name for the objective function and its gradient, it is possible to use one function to compute both. The gradient must then be returned as a second output argument. The same strategy applies to constraints. If the gradient or Jacobian field for the objective or constraints is not assigned, the solvers estimate the missing derivatives by finite differences.

The sparsity patterns for the objective function gradient and the constraint Jacobian may be given as sparse matrices. If they are not provided, the solver will estimate a sparsity pattern of a nonlinear problem through repeated objective function and constraint residual evaluations.

User-defined functions take the present vector `x` as their input, but it is possible to supply extra arguments by using the `param` solver option. The solver then passes these additional arguments at each callback.

Note: All fields in the Opt structure are optional except `opt.obj`, but if a specific constraint type is supplied, it is necessary to include the main field of that type (for example, `opt.lc.A` for the linear constraints and `opt.nc.c` for the nonlinear constraints) and at least one of the upper or lower bounds. In some cases, when the solver cannot determine the number of variables from the Opt structure, it is necessary to supply an initial guess (the default is zero for all variables). An example of this would be an unconstrained nonlinear problem.

`sol.eval` is a structure with the following fields:

TABLE 5-2: THE EVAL STRUCTURE

FIELD NAME	INTERPRETATION
`f`	Final value of objective function (scalar)
`lc`	Final value of linear constraints, if any (vector)
`nc`	Final value of nonlinear constraints, if any (vector)

`y` is a structure with the following fields:

TABLE 5-3: THE Y STRUCTURE

FIELD NAME	INTERPRETATION
`bc`	Lagrange multipliers for the bounds on the variables
`lc`	Lagrange multipliers for the linear constraints
`nc`	Lagrange multipliers for the nonlinear constraints

Note: The initial guess for the Lagrange multipliers cannot be supplied for some solvers and, when available, it is for the linear and nonlinear constraints only. See documentation about the specific solver for details.

Positive and negative multipliers indicate active lower and upper bounds, respectively.

`xinfo` is a structure with the following fields:

TABLE 5-4: THE XINFO STRUCTURE

FIELD NAME	INTERPRETATION
`xinfo.state.bc`	Vector containing the final state of the variables. state(j) = 0 means that $x(j)$ is nonbasic and usually equals bc.xlb(j). state(j) = 1 means that $x(j)$ is nonbasic and usually equals bc.xub(j). state(j) = 2 means that $x(j)$ is superbasic and usually between bc.xlb(j) and bc.xub(j). state(j) = 3 means that $x(j)$ is basic and usually between bc.xlb(j) and bc.xub(j). Basic and superbasic variables might be outside their bounds by as much as the feasibility tolerance (or minor feasibility tolerance in the case of a nonlinear problem). Note that if scaling is specified, the feasibility tolerance applies to the variables of the scaled problem. In this case, the variables of the original problem might be as much as 0.1 outside their bounds, but this is unlikely unless the problem is very badly scaled. Very occasionally, some nonbasic variables might be outside their bounds by as much as the feasibility tolerance, and there might be some nonbasics for which $x(j)$ lies strictly between its bounds. If `nInf` > 0, some basic and superbasic variables might be outside their bounds by an arbitrary amount (bounded by `sInf` if scaling was not used)
`xinfo.state.lc`	Analogous to `xinfo.state.bc` but for the linear constraints
`xinfo.state.nc`	Analogous to `xinfo.state.bc` but for the nonlinear constraints.
`xinfo.ns`	The final number of superbasic variables
`xinfo.ninf`	The number of infeasibilities
`xinfo.sinf`	The sum of infeasibilities

For further details regarding the `xinfo` fields, see the SNOPT documentation.

`optim` accepts the solver properties for the respective functions as given by `optprop` and `optpropnlin`. These can be given either in the `options` structure or as property/value pairs.

See Also `optlin`, `optnlin`, `optquad`, `optlinlsq`, `optnlinlsq`

Purpose

Solve a linear optimization problem.

Syntax

```
opt.sol = optlin(opt,...)
opt.sol = optlin(opt,options)
```

Description

`optlin` solves the linear optimization problem

$$\begin{aligned} &\min && c^T x \\ &b_{\mathrm{lb}} \le Ax \le b_{\mathrm{ub}} \\ &x_{\mathrm{lb}} \le x \le x_{\mathrm{ub}} \end{aligned}$$

which corresponds to the following fields in the Opt structure (for information about which fields are optional, see `optim`):

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.obj.c`	c	Coefficient vector for the linear term of the objective function (numeric vector)
`opt.obj.form`		Specifies the form of the objective function. If it is not included, `optlin` assumes the general linear formulation, `'lin'` (the one given in this table). It is also possible to call `optlin` with an objective of the `'quad'` form, in which case the quadratic terms are ignored. (For more information on objective function forms, see `optim`.)
`opt.lc.A`	A	Linear constraints (matrix)
`opt.lc.lb`	b_{lb}	Lower bounds for linear constraints (vector or scalar), default `-Inf`
`opt.lc.ub`	b_{ub}	Upper bounds for linear constraints (vector or scalar), default `Inf`
`opt.bc.lb`	x_{lb}	Lower bounds for variables (vector or scalar), default `-Inf`
`opt.bc.ub`	x_{ub}	Upper bounds for variables (vector or scalar), default `Inf`
`opt.init.x`		Initial guess (vector), default vector of zeros
`opt.sol.x`		Solution (vector)
`opt.sol.eval`		Substructure containing value of objective function and constraints, if any

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.sol.y`		Lagrange multipliers in solution (structure)
`opt.sol.xinfo`		Information about the solution (structure)
`opt.sol.exit`		Exit condition (scalar). `exit` is 1 for successful completion, 0 otherwise.
`opt.sol.info`		Info flag as specified by SQOPT. (See the SQOPT User's Guide for further details.)
`opt.sol.msg`		Message corresponding to the info flag as specified by SQOPT. (See the SQOPT User's Guide for further details.)
`opt.sol.algorithm`		`'optlin'`, to indicate solver used

`optlin` ignores any nonlinear constraints present in the Opt structure.

`y` is a structure with the following field:

TABLE 5-5: THE Y STRUCTURE

FIELD NAME	INTERPRETATION
`lc`	Lagrange multipliers for the linear constraints

Positive and negative multipliers indicate active lower and upper bounds, respectively.

For information about `eval` and `xinfo`, see `optim`.

Parameters to the solvers can be given either as property/value pairs or in the `options` structure. For a list of solver properties see `optprop`.

Algorithm

`optlin` uses the SQOPT package. For further details, see *User's Guide for SQOPT: A Fortran Package for Large-Scale Linear and Quadratic Programming* (Philip E. Gill, Walter Murray, and Michael A. Saunders).

See Also

`optquad`, `optnlin`

Purpose

Solve a linear least-squares problem.

Syntax

```
opt.sol = optlinlsq(opt,...)
opt.sol = optlinlsq(opt,options)
```

Description

`oplinlsq` solves the linear least-squares problem

$$\min \quad \frac{1}{2}\|Cx - d\|^2$$

$$b_{\text{lb}} \le Ax \le b_{\text{ub}}$$

$$x_{\text{lb}} \le x \le x_{\text{ub}}$$

which corresponds to the following fields in the Opt structure (for information about which fields are optional, see `optim`):

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.obj.C`	C	Objective function matrix (matrix)
`opt.obj.d`	d	Objective function vector (numeric vector), default vector of zeros
`opt.obj.form`		Specifies the form of the objective function. If it is not included, `optlinlsq` assumes the linear least squares formulation, `'linlsq'` (the one given in this table). This is the only form allowed for the `optlinlsq` solver. (For more information on objective function forms, see `optim`.)
`opt.lc.A`	A	Linear constraints (matrix)
`opt.lc.lb`	b_{lb}	Lower bounds for linear constraints (vector or scalar), default `-Inf`
`opt.lc.ub`	b_{ub}	Upper bounds for linear constraints (vector or scalar), default `Inf`
`opt.bc.lb`	x_{lb}	Lower bounds for variables (vector or scalar), default `-Inf`
`opt.bc.ub`	x_{ub}	Upper bounds for variables (vector or scalar), default `Inf`
`opt.init.x`		Initial guess (vector), default vector of zeros
`opt.sol.x`		Solution (vector)

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.sol.eval`		Substructure containing value of objective function and constraints, if any
`opt.sol.y`		Lagrange multipliers in solution (structure)
`opt.sol.xinfo`		Information about the solution (structure)
`opt.sol.exit`		Exit condition (scalar). `exit` is 1 for successful completion, 0 otherwise.
`opt.sol.info`		Info flag as specified by SQOPT. (See the SQOPT User's Guide for further details.)
`opt.sol.msg`		Message corresponding to the info flag as specified by SQOPT. (See the SQOPT User's Guide for further details.)
`opt.sol.algorithm`		`'optlinlsq'`, to indicate solver used

`optlinlsq` ignores any nonlinear constraints present in the Opt structure.

`y` is a structure with the following field:

TABLE 5-6: THE Y STRUCTURE

FIELD NAME	INTERPRETATION
`lc`	Lagrange multipliers for the linear constraints

Positive and negative multipliers indicate active lower and upper bounds, respectively.

For information about `eval` and `xinfo`, see `optim`.

Parameters to the solvers can be given either as property/value pairs or in the `options` structure. For a list of solver properties see `optprop`.

Algorithm

`optlinlsq` uses the SQOPT package with the following objective function:

$$f(x) = \frac{1}{2}x^T C^T Cx - 2d^T Cx + \frac{1}{2}d^T d$$

For further details about SQOPT, see *User's Guide for SQOPT: A Fortran Package for Large-Scale Linear and Quadratic Programming* (Philip E. Gill, Walter Murray, and Michael A. Saunders).

See Also

`optnlinlsq`

Purpose

Solve a nonlinear optimization problem.

Syntax

```
opt.sol = optnlin(opt,...)
opt.sol = optnlin(opt,options)
```

Description

`optnlin` solves the nonlinear optimization problem

$$\begin{aligned} &\min && f(x) \\ &b_{\mathrm{lb}} \le Ax \le b_{\mathrm{ub}} \\ &d_{\mathrm{lb}} \le c(x) \le d_{\mathrm{ub}} \\ &x_{\mathrm{lb}} \le x \le x_{\mathrm{ub}} \end{aligned}$$

which corresponds to the following fields in the Opt structure (for information about which fields are optional, see `optim`):

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.obj.f`	f	Objective function (function)
`opt.obj.g`		Gradient of objective function (function)
`opt.obj.ptrn`		Gradient sparsity pattern (matrix)
`opt.obj.form`		Specifies the form of the objective function. If it is not included, `optnlin` assumes the general nonlinear formulation, `'nlin'` (the one given in this table). It is also possible to call `optnlin` with an objective of the `'lin'` or `'quad'` forms. (For more information on objective function forms, see `optim`.)
`opt.lc.A`	A	Linear constraints (matrix)
`opt.lc.lb`	b_{lb}	Lower bounds for linear constraints (vector or scalar), default `-Inf`
`opt.lc.ub`	b_{ub}	Upper bounds for linear constraints (vector or scalar), default `Inf`
`opt.nc.c`	c	Nonlinear constraints (function)
`opt.nc.J`		Jacobian of nonlinear constraints (function)
`opt.nc.lb`	d_{lb}	Lower bounds for nonlinear constraints (vector or scalar), default `-Inf`

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.nc.ub`	d_{ub}	Upper bounds for nonlinear constraints (vector or scalar), default `Inf`
`opt.nc.ptrn`		Constraint Jacobian sparsity pattern (matrix)
`opt.bc.lb`	x_{lb}	Lower bounds for variables (vector or scalar), default `-Inf`
`opt.bc.ub`	x_{ub}	Upper bounds for variables (vector or scalar), default `Inf`
`opt.init.x`		Initial guess (vector), default vector of zeros
`opt.init.y`		Initial guess for Lagrange multipliers (structure), default zero for all multipliers
`opt.sol.x`		Solution (vector)
`opt.sol.eval`		Substructure containing value of objective function and constraints, if any
`opt.sol.y`		Lagrange multipliers in solution (structure)
`opt.sol.xinfo`		Information about the solution (structure)
`opt.sol.exit`		Exit condition (scalar). `exit` is 1 for successful completion, 0 otherwise.
`opt.sol.info`		Info flag as specified by SNOPT. (See the SNOPT user's manual for further details.)
`opt.sol.msg`		Message corresponding to the info flag as specified by SNOPT. (See the SNOPT user's manual for further details.)
`opt.sol.algorithm`		`'opntlin'`, to indicate solver used

Functions can be given either as strings, denoting the function name, or as inline functions. By specifying the same function name for the objective function and its gradient, it is possible to use one function to compute both. The gradient must then be returned as a second output argument. The same strategy applies to constraints.

If the gradient or Jacobian field for the objective or constraints is not assigned, `optnlin` estimates the missing derivatives by finite differences.

The sparsity patterns for the objective function gradient and the constraint Jacobian may be given as sparse matrices. If they are not provided, the solver will estimate a sparsity pattern through repeated objective function and constraint residual evaluations.

User-defined functions take the present vector `x` as their input, but it is possible to supply extra arguments by using the `param` solver option. The solver then passes along these additional arguments at each callback.

`y` is a structure with the following fields:

TABLE 5-7: THE Y STRUCTURE

FIELD NAME	INTERPRETATION
`bc`	Lagrange multipliers for the bounds on the variables (Output only. Cannot be supplied as initial guess.)
`lc`	Lagrange multipliers for the linear constraints
`nc`	Lagrange multipliers for the nonlinear constraints

Positive and negative multipliers indicate active lower and upper bounds, respectively.

For information about `eval` and `xinfo`, see `optim`.

Parameters to the solvers can be given either as property/value pairs or in the `options` structure. For a list of solver properties, see `optpropnlin`.

Algorithm

`optnlin` uses the SNOPT package. For further details, see *User's Guide for SNOPT, A Fortran Package for Large-Scale Nonlinear Programming* (Philip E. Gill, Walter Murray, and Michael A. Saunders).

Example

The code in this example defines and solves the following nonlinear optimization problem:

$$\begin{aligned} \min \quad & 3x_1 + (x_1 + x_2 + x_3)^2 + 5x_4 \\ & 4x_2 + 2x_3 \geq 0 \\ & x_1 + x_2^2 + x_3^2 = 2 \\ & x_2^4 + x_3^4 + x_4 = 4 \\ & x_1 \geq 0, x_4 \geq 0 \end{aligned}$$

```
function [f,g] = exnlin_obj(x)
f = 3*x(1) + (x(1) + x(2) + x(3))^2 + 5*x(4);

g = [3 + 2*(x(1) + x(2) + x(3)),
2*(x(1) + x(2) + x(3)),
2*(x(1) + x(2) + x(3)),
5]';

function [f,g] = exnlin_con(x)
f = [x(1) + x(2)^2 + x(3)^2;
     x(2)^4 + x(3)^4 + x(4)];

g = [1 2*x(2) 2*x(3) 0;
     0 4*x(2) 4*x(3) 1];

clear opt;
opt.obj.f = 'exnlin_obj';
opt.obj.g = 'exnlin_obj';

opt.nc.c = 'exnlin_con';
opt.nc.J = 'exnlin_con';
opt.nc.lb = [2 4];
opt.nc.ub = opt.nc.lb;

opt.lc.A = [0 4 2 0];
opt.lc.lb = 0;
opt.lc.ub = Inf;

opt.bc.lb = [0,-Inf,-Inf, 0]';
opt.bc.ub = Inf;

opt.init.x = ones(4,1);
opt.sol = optnlin(opt);
```

See Also optlin, optquad

Purpose

Solve a nonlinear least-squares problem.

Syntax

```
opt.sol = optnlinlsq(opt,...)
opt.sol = optnlinlsq(opt,options)
```

Description

`optnlinlsq` solves the nonlinear least-squares problem

$$\min \quad \frac{1}{2}\|F(x)\|^2 = \frac{1}{2}\sum_i F_i(x)^2$$
$$b_{\mathrm{lb}} \le Ax \le b_{\mathrm{ub}}$$
$$d_{\mathrm{lb}} \le c(x) \le d_{\mathrm{ub}}$$
$$x_{\mathrm{lb}} \le x \le x_{\mathrm{ub}}$$

which corresponds to the following fields in the Opt structure (for information about which fields are optional, see `optim`):

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.obj.F`	F	Objective function (function). Note that the function must return the vector of F_i to be summed, not the sum of squares
`opt.obj.J`	J	Jacobian of F (matrix).
`opt.obj.ptrn`		Objective Jacobian sparsity pattern (matrix)
`opt.obj.form`		Specifies the form of the objective function. If it is not included, `optnlinlsq` assumes the nonlinear least-squares formulation, `'nlinlsq'` (the one given in this table). It is also possible to call `optnlinlsq` with an objective function of the `'linlsq'` form. (For more information on objective function forms, see `optim`.)
`opt.lc.A`	A	Linear constraints (matrix)
`opt.lc.lb`	b_{lb}	Lower bounds for linear constraints (vector or scalar), default `-Inf`
`opt.lc.ub`	b_{ub}	Upper bounds for linear constraints (vector or scalar), default `Inf`
`opt.nc.c`	c	Nonlinear constraints (function)

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.nc.J`		Jacobian of nonlinear constraints (function)
`opt.nc.lb`	d_{lb}	Lower bounds for nonlinear constraints (vector or scalar), default `-Inf`
`opt.nc.ub`	d_{ub}	Upper bounds for nonlinear constraints (vector or scalar), default `Inf`
`opt.nc.ptrn`		Constraint Jacobian sparsity pattern (matrix)
`opt.bc.lb`	x_{lb}	Lower bounds for variables (vector or scalar), default `-Inf`
`opt.bc.ub`	x_{ub}	Upper bounds for variables (vector or scalar), default `Inf`
`opt.init.x`		Initial guess (vector), default vector of zeros
`opt.init.y`		Initial guess for Lagrange multipliers (structure), default zero for all multipliers
`opt.sol.x`		Solution (vector)
`opt.sol.eval`		Substructure containing value of objective function and constraints, if any
`opt.sol.y`		Lagrange multipliers in solution (structure)
`opt.sol.xinfo`		Information about the solution (structure)
`opt.sol.exit`		Exit condition (scalar). `exit` is 1 for successful completion, 0 otherwise.
`opt.sol.info`		Info flag as specified by SNOPT. (See the SNOPT user's manual for further details.)
`opt.sol.msg`		Message corresponding to the info flag as specified by SNOPT. (See the SNOPT user's manual for further details.)
`opt.sol.algorithm`		`'opntlinlsq'`, to indicate solver used

Functions can be given either as strings, denoting the function name, or as inline functions. By specifying the same function name for the objective function and its

Jacobian, it is possible to use one function to compute both. The Jacobian must then be returned as a second output argument. The same strategy applies to constraints. If the Jacobian field is not assigned, `optnlinlsq` estimates the missing derivatives by finite differences.

The sparsity patterns for the objective function and constraint Jacobian may be given as sparse matrices. If they are not provided, the solver will estimate a sparsity pattern through repeated objective function and constraint residual evaluations.

User-defined functions take the present vector `x` as their input, but it is possible to supply extra arguments by using the `param` solver option. The solver then passes along these additional arguments at each callback.

`y` is a structure with the following fields:

TABLE 5-8: THE Y STRUCTURE

FIELD NAME	INTERPRETATION
`bc`	Lagrange multipliers for the bounds on the variables (Output only. Cannot be supplied as initial guess.)
`lc`	Lagrange multipliers for the linear constraints
`nc`	Lagrange multipliers for the nonlinear constraints

Positive and negative multipliers indicate active lower and upper bounds, respectively.

For information about `eval` and `xinfo` see `optim`.

Parameters to the solvers can be given either as property/value pairs or in the `options` structure. For a list of solver properties, see `optpropnlin`.

Example

The code in this example defines and solves the following nonlinear optimization problem:

$$\min \quad \frac{1}{2}(x_1^2 + x_2^2)$$

$$x_1 + x_2 \geq 1$$
$$x_1^2 + x_2^2 \geq 1$$
$$9x_1^2 + x_2^2 \geq 9$$
$$x_1^2 - x_2 \geq 0$$
$$x_2^2 - x_1 \geq 0$$
$$-50 \leq x_1 \leq 50$$
$$-50 \leq x_2 \leq 50$$

(Reference: W. Hock and K. Schittkowski, "Test Examples for Nonlinear Programming Codes," *Lecture Notes in Economics and Mathematical Systems 187*, Springer-Verlag, 1981.)

```
function [F,J] = exnlinlsq_con(x)
F = [  x(1)^2 + x(2)^2;
     9*x(1)^2 + x(2)^2;
       x(1)^2 - x(2);
       x(2)^2 - x(1)];

J = [ 2*x(1), 2*x(2);
     18*x(1),2*x(2);
      2*x(1),-1;
     -1,2*x(2)];

clear opt;
opt.obj.C = eye(2);
opt.obj.form = 'linlsq';

opt.lc.A = [1 1];
opt.lc.lb = [1];

opt.nc.c = 'exnlinlsq_con';
opt.nc.J = 'exnlinlsq_con';
opt.nc.lb = [1 9 0 0];

opt.bc.lb = -50;
opt.bc.ub = 50;

% Infeasible starting point
opt.init.x = [3 0.6];
opt.sol = optnlinlsq(opt);
```

Algorithm

`optnlinlsq` uses the SNOPT package with the following objective function and gradient:

$$f(x) = \frac{1}{2}\sum_i F_i(x)^2$$

$$g(x) = J^T F$$

For further details about SNOPT see *User's Guide for SNOPT, A Fortran Package for Large-Scale Nonlinear Programming* (Philip E. Gill, Walter Murray, and Michael A. Saunders).

See Also

`optlinlsq`

Purpose

Solve an unconstrained nonlinear optimization problem using the Nelder-Mead simplex algorithm.

Syntax

```
opt.sol = optnm(opt,...)
opt.sol = optnm(opt,options)
x = optnm(fun,x0,...)
[x,f] = optnm(fun,x0,...)
[x,f,exit] = optnm(fun,x0,...)
optnm(fun,x0,options)
```

Description

`optnm` solves the unconstrained nonlinear optimization problem min $f(x)$, which corresponds to the following fields of the Opt structure (for information about which fields are optional, see `optim`):.

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.obj.f`	f	Objective function (function)
`opt.obj.form`		Specifies the form of the objective function. `optnm` expects the form to equal `'nlin'` (assumed if `form` is not given) but also accepts `'lin'` or `'quad'` form. (For more information on objective function forms, see `optim`.)
`opt.init.x`		Initial guess (vector)
`opt.sol.x`		Solution (vector)
`opt.sol.eval.f`		Value of objective function (scalar)
`opt.sol.exit`		Exit condition (scalar). `exit` is 1 for successful completion, 0 otherwise.
`opt.sol.algorithm`		`'optnm'`, to indicate solver used

Functions can be either strings, denoting the function name, or inline functions.

The user-defined function takes the present vector of variables `x` as its input, but it is possible to supply extra arguments by using the `param` solver option. The solver then passes along these additional arguments at each callback.

`optnm` accepts the following properties, which can be given either as property/value pairs or in the `options` structure (`n` is the number of variables):

TABLE 5-9: VALID PROPERTIES FOR THE OPTNM FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`dtol`	numeric	`1e-4`	Absolute termination tolerance on the largest diameter of the simplex, using infinity norm
`functol`	numeric	`1e-4`	Absolute termination tolerance on the function precision
`funlim`	integer	`200*n`	Limit on number of function evaluations
`itlim`	integer	`200*n`	Iterations limit
`lscale`	numeric	1	Length scale used when creating the initial simplex, which is defined by the starting guess `x0` and `n` more points `x0 + lscale*eye(n)`.
`out`	`'sol'` | `'opt'` | `'x'` | `'f'` | `'exit'` | `'funs'` | `'iter'`	`'sol'`	Output variables. `'sol'` returns the `opt.sol` structure; `'opt'` returns the complete Opt structure; `'x'`, `'f'`, and `'exit'` return the respective fields of the `opt.sol` structure; `'funs'` returns the actual number of function evaluations; and `'iter'` returns the number of iterations

TABLE 5-9: VALID PROPERTIES FOR THE OPTNM FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`param`	any	`empty`	Allows additional arguments to be passed along to the callback function. Use a cell array to pass along more than one argument. Note that the cell array is unpacked in the function call, so setting `param` to `{a1,a2}` results in the function being called with `userfun(x,a1,a2)`.
`report`	`'on'` | `'off'`	`'off'`	`'on'` displays the result at each iteration and whether `optnm` performs a reflection, expansion, inner or outer contraction, or a shrinking step.

`optnm(fun,x0,...)` is a shortcut version of the structure form.

`[x,f] = optnm(fun,x0)`, for example, equals the following command sequence:

```
opt.obj.f = fun;
opt.init.x = x0;
options.out = {'x','f'};
[x,f] = optnm(opt,options);
```

Examples

```
function f = exnm_obj(x)
f = x(1)^4 + x(2)^4 - x(1)*x(2) + 1;

clear opt;
opt.obj.f = 'exnm_obj';
opt.init.x = [1 1];
opt.sol = optnm(opt);

% Vector notation:
[x,f] = optnm('exnm_obj',[1 1]);

% With inline function
[x,f] = optnm(inline('x(1)^4 + x(2)^4 - x(1)*x(2) + 1'),[1 1]);
```

Algorithm

`optnm` uses the Nelder-Mead simplex algorithm as defined in "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions" (J.C. Lagarias, J.A. Reeds, M.H. Wright, and P.E. Wright, *SIAM J. Optimization*, vol. 9, pp. 112–147, 1998).

See Also `optnlin`, `optlin`, `optquad`

Purpose

Optimization Lab solver properties.

Syntax

```
options = optprop
```

Description

The following table lists all common property/value pairs for the linear and quadratic solvers in the Optimization Lab. In the table, `m` is the number of constraints, while `n1` is the number of leading nonzero columns of the Hessian in the case of a quadratic problem and `0` in the linear case.

`options = optprop` returns a struct `options` with all available properties, each set to its default value. Properties whose default depends on the input problem are not set (the corresponding field contains an empty matrix).

For more details on the solver properties see "Solver Properties" in Chapter 6.

TABLE 5-10: COMSOL OPTIMIZATION LAB PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`checkfreq`	integer	60	Check frequency
`elastic`	0 | 1 | 2	1	Elastic mode. 0 indicates that Elastic mode is never invoked, which causes the solver to terminate as soon as infeasibilities are detected. 1 indicates that Elastic mode is entered when infeasibilities are detected. 2 indicates that the solver starts and remains in Elastic mode

TABLE 5-10: COMSOL OPTIMIZATION LAB PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`elasticbc`	`scalar or vector of 0 | 1 | 2 | 3`	0	Indicates which bound constraints can be elastic. 0 indicates that corresponding constraints cannot be violated. 1 indicates that the lower bound can be violated. 2 indicates that the upper bound can be violated. 3 indicates that either bound can be violated.
`elasticlc`	`scalar or vector of 0 | 1 | 2 | 3`	3	Indicates which linear constraints can be elastic. 0 indicates that corresponding constraints cannot be violated. 1 indicates that the lower bound can be violated. 2 indicates that the upper bound can be violated. 3 indicates that either bound can be violated.
`elasticobj`	`0 | 1 | 2`	2	Elastic objective
`elasticw`	numeric	`1.0`	Elastic weight
`expfreq`	integer	10000	Expand frequency
`facfreq`	integer	100 (linear) 50 (quadratic)	Factorization frequency
`feastol`	numeric	`1.0e-6`	Feasibility tolerance

TABLE 5-10: COMSOL OPTIMIZATION LAB PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
hessdim	integer	min(1000,n1+1)	Hessian dimension
infbound	positive numeric	1.0e20	Infinite bound size
itlim	integer	3*m	Iterations limit
maximize	'on' | 'off'	'off'	'on' if objective should be maximized
print	filename	no printing	Print information about the solver progress and solution to file
opttol	numeric	1.0e-6	Optimality tolerance
out	'sol' | 'opt' | 'x' | 'f' | 'y' | 'xinfo' | 'state' | 'ns' | 'ninf' | 'sinf' | 'exit' | 'info' | 'msg' | 'algorithm'	'sol'	Output variables
param	any	empty	Allows additional arguments to be passed along to the user callback functions. Use a cell array to pass along more than one argument. Note that the cell array is unpacked in the function call, so setting param to {a1,a2} results in functions being called with userfun(x,a1,a2).
parprice	integer	10 (linear) 1 (quadratic)	Partial price
pivtol	numeric	3.7e-11	Pivot tolerance

TABLE 5-10: COMSOL OPTIMIZATION LAB PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
print	filename	empty (no printing)	Print information about the solver progress and solution to file
qpsolver	'cholesky' | 'cg' | 'qn'	'cholesky'	Specifies the active-set algorithm used in the optimality phase. 'cholesky' indicates the Cholesky solver and 'qn' indicates the quasi-Newton method. 'cg' uses an active-set method similar to 'qn' but uses the conjugate-gradient method to solve all systems involving the reduced Hessian.
scaleopt	0 | 1 | 2	2 (linear) 1 (quadratic)	Scale option
scaletol	numeric	0.9	Scale tolerance
stop	'on' | 'off'	'on'	When 'on', deliver partial solution when failing
suplim	integer	n1+1	Superbasics limit

See Also optpropnlin, optnlin, optlinlsq, optquad

Purpose

Nonlinear Optimization Lab solver properties.

Syntax

```
options = optpropnlin
```

Description

The following table lists all common property/value pairs used in the Optimization Lab solvers `optnlin` and `optnlinlsq`. In the table, `m` is the number of constraints and `n1` is the number of nonlinear variables, which is computed internally by the solvers.

`options = optpropnlin` returns a struct with all available properties, each set to its default value. Properties whose default depends on the input problem are not set (the corresponding field contains an empty matrix).

For more details on the solver properties, see "Solver Properties" in Chapter 6.

TABLE 5-11: COMSOL OPTIMIZATION LAB PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`cendiff`	numeric	`6.0e-6`	Central difference interval
`checkfreq`	integer	`60`	Check frequency
`diffint`	`numeric`	`1.5e-8`	Difference interval
`elasticw`	numeric	`1.0e4`	Elastic weight
`expfreq`	integer	`10000`	Expand frequency
`facfreq`	integer	`50`	Factorization frequency
`feastol`	numeric	`1.0e-6`	Minor feasibility tolerance
`funcprec`	numeric	`3.8e-11`	Function precision
`hessdim`	integer	`min(1000,n1+1)`	Hessian dimension
`hessfreq`	integer	`9999999`	Hessian frequency
`hessmem`	`'full'` | `'limited'`	`'limited'` if `n1 > 75` or qpsolver `'cg'`	Hessian memory
`hessupd`	integer	`10 if hessmem 'limited'`	Hessian updates
`infbound`	positive numeric	`1.0e20`	Infinite bound size
`itlim`	integer	`500`	Minor iteration limit
`linesearch`	`'derivative'` | `'nonderivative'`	`derivative`	Linesearch method

TABLE 5-11: COMSOL OPTIMIZATION LAB PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`linestol`	numeric	`0.9`	Linesearch tolerance
`majfeastol`	numeric	`1.0e-6`	Major feasibility tolerance
`majitlim`	integer	`max(1000,m)`	Major iterations limit
`majsteplim`	numeric	`2.0`	Major step limit
`maximize`	`'on'` | `'off'`	`'off'`	'on' if objective should be maximized
`newsuplim`	integer	`99`	New superbasics limit
`opttol`	numeric	`1.0e-6`	Optimality tolerance
`out`	`'sol'` | `'opt'` | `'x'` | `'f'` | `'y'` | `'xinfo'` | `'state'` | `'ns'` | `'ninf'` | `'sinf'` | `'exit'` | `'info'` | `'msg'` | `'algorithm'`	`'sol'`	Output variables
`param`	any	`empty`	Allows additional arguments to be passed along to the user callback functions. Use a cell array to pass along more than one argument. Note that the cell array is unpacked in the function call, so setting `param` to `{a1,a2}` results in functions being called with `userfun(x,a1,a2)`.
`pivtol`	numeric	`3.7e-11`	Pivot tolerance
`print`	`filename`	`empty (no printing)`	Print information about the solver progress and solution to file

TABLE 5-11: COMSOL OPTIMIZATION LAB PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
proxmeth	1 | 2	1	Proximal point method
qpsolver	'cholesky' | 'cg' | 'qn'	'cholesky'	Specifies the active-set algorithm used to solve the QP subproblem. 'cholesky' indicates the Cholesky solver and 'qn' indicates the quasi-Newton method. 'cg' uses an active-set method similar to 'qn' but uses the conjugate-gradient method to solve all systems involving the reduced Hessian.
scaleopt	0 | 1 | 2	1	Scale option
scaletol	numeric	0.9	Scale tolerance
stop	'on' | 'off'	'on'	When 'on', deliver partial solution when failing
suplim	integer	n1+1	Superbasics limit
totitlim	integer	max(10000,20*m)	Iterations limit (absolute limit on the total number of minor iterations

TABLE 5-11: COMSOL OPTIMIZATION LAB PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`verify`	`-1 | 0 | 1 | 2 | 3`	0	Verification level of derivatives through finite-differences. Derivatives are checked at the first point that satisfies all bounds and linear constraints. `-1` indicates that derivative checking is disabled. 0 indicates that only a cheap test is performed, requiring two calls to user functions. 1 indicates that individual gradients are checked with a more reliable test. 2 indicates that individual columns of the problem Jacobian are checked. 3 indicates that both options 2 and 1 occur (in that order).
`viollim`	numeric	10	Violation limit

See Also `optprop`, `optnlin`, `optnlinlsq`

Purpose

Solve a quadratic optimization problem.

Syntax

```
opt.sol = optquad(opt,...)
opt.sol = optquad(opt,options)
```

Description

`optquad` solves the quadratic optimization problem

$$\min \quad \frac{1}{2}x^T Hx + c^T x$$

$$b_{\text{lb}} \le Ax \le b_{\text{ub}}$$

$$x_{\text{lb}} \le x \le x_{\text{ub}}$$

which corresponds to the following fields in the Opt structure (for information about which fields are optional, see `optim`):

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.obj.H`	H	Coefficient matrix for the quadratic terms of the objective function (symmetric matrix or function returning Hx), default zero
`opt.obj.c`	c	Coefficient vector for the linear term of the objective function (numeric vector), default vector of zeros
`opt.obj.form`		Specifies the form of the objective function. If it is not included, `optquad` assumes the general quadratic formulation, `'quad'` (the one given in this table). It is also possible to call `optquad` with an objective of the `'lin'` form. (For more information on objective function forms see `optim`.)
`opt.lc.A`	A	Linear constraints (matrix)
`opt.lc.lb`	b_{lb}	Lower bounds for linear constraints (vector or scalar), default `-Inf`
`opt.lc.ub`	b_{ub}	Upper bounds for linear constraints (vector or scalar), default `Inf`
`opt.bc.lb`	x_{lb}	Lower bounds for variables (vector or scalar), default `-Inf`
`opt.bc.ub`	x_{ub}	Upper bounds for variables (vector or scalar), default `Inf`

FIELD NAME	MATHEMATICAL NOTATION	INTERPRETATION
`opt.init.x`		Initial guess (vector), default vector of zeros
`opt.sol.x`		Solution (vector)
`opt.sol.eval`		Substructure containing value of objective function and constraints, if any
`opt.sol.y`		Lagrange multipliers in solution (structure)
`opt.sol.xinfo`		Information about the solution (structure)
`opt.sol.exit`		Exit condition (scalar). `exit` is 1 for successful completion, 0 otherwise.
`opt.sol.info`		Info flag as specified by SQOPT. (See the SQOPT User's Guide for further details.)
`opt.sol.msg`		Message corresponding to the info flag as specified by SQOPT. (See the SQOPT User's Guide for further details.)
`opt.sol.algorithm`		`'optquad'`, to indicate solver used

When *Hx* is supplied as a function, it can be either a string, denoting the function name, or an inline function. It should take the present vector `x` as its input, but it is possible to supply extra arguments by using the `param` solver option. The solver then passes along these additional arguments at each callback.

`y` is a structure with the following field:

TABLE 5-12: THE Y STRUCTURE

FIELD NAME	INTERPRETATION
`lc`	Lagrange multipliers for the linear constraints

Positive and negative multipliers indicate active lower and upper bounds, respectively.

For information about `eval` and `xinfo` see `optim`.

Parameters to the solvers can be given either as property/value pairs or in the `options` structure. For a list of properties see `optprop`.

Algorithm

`optquad` uses the SQOPT package. For further details, see *User's Guide for SQOPT: A Fortran Package for Large-Scale Linear and Quadratic Programming* (Philip E. Gill, Walter Murray, and Michael A. Saunders).

See Also optlin, optnlin

Purpose

Set solver status during callback.

Syntax

```
optsetstatus(s)
```

Description

`optsetstatus(s)` is intended for use in the Optimization Lab's nonlinear callback routines. Call `optsetstatus` with `s = -1` to reduce the step length during a line search (for example, if the current point is undefined). Call `optsetstatus` with `s ≤ -2` to request a solver abort.

For more information see the SNOPT documentation.

See Also

`optgetstatus`

Diagnostics

Error Messages and Troubleshooting

The Optimization Lab solvers all return three pieces of information: an information flag (`sol.info`), an integer exit code or flag indicating the exit condition, and a message describing this particular state (`sol.msg`). The various states are grouped into more general categories such that for all problems of a specific type, the most significant digit of the exit code is the same. The list of general groups are:

TABLE 5-13: GROUPS OF POSSIBLE EXIT CONDITIONS

STATE GROUP	DESCRIPTION
0	Finished successfully
10	The problem appears to be infeasible
20	The problem appears to be unbounded
30	Resource limit error
40	Terminated after numerical difficulties
50	Error in the user-supplied functions
60	Undefined user-supplied functions
70	User requested termination
80	Insufficient storage allocated
90	Input arguments out of range
140	System error

The exit codes 0–20 appear when a solution exists (though it might not be optimal).

Following is a description of each message and possible courses of action. Not all return states are possible for all types of problems; some typically occur only for the nonlinear solvers.

SUCCESSFUL COMPLETION

TABLE 5-14: OUTPUT AT SUCCESSFUL COMPLETION

INFO FLAG	DESCRIPTION
0	Finished successfully
1	Optimality conditions satisfied
2	Feasible point found
3	Requested accuracy could not be achieved
4	Weak QP minimizer

These messages call for guarded optimism! They are certainly preferable to every other message, and you naturally want to believe what they say. However, in every case a distinct level of caution is in order. For example, if the objective value is much better than expected, you might have obtained an optimal solution to the wrong problem. Almost any item of data could have that effect if it has the wrong value. Verifying that you have defined the problem correctly is one of the more difficult tasks for a model builder.

If nonlinearities exist, you must always ask the question: could there be more than one local optimum? When the constraints are linear and the objective is known to be convex (for example, a sum of squares), then all is well if you are minimizing the objective—a local minimum is a global minimum in the sense that no other point has a lower function value. (However, many points could have the same objective value, particularly if the objective is largely linear.)

Conversely, if you are maximizing a convex function, you cannot expect a local maximum to be global unless there are sufficient constraints to confine the feasible region. Similar statements could be made about nonlinear constraints defining convex or concave regions. However, the functions of a problem are more likely to be neither convex nor concave. It is good practice to specify a starting point that is as good an estimate as possible and then to also include reasonable upper and lower bounds on all variables in order to confine the solution to the specific region of interest.

For Flag 4, the final point is a weak minimizer. (The objective value is a global optimum, but it might be achieved by an infinite set of points x.) This exit code applies to `optquad` only (QP problems) and arises when (i) the problem is feasible, (ii) the reduced gradient is negligible, (iii) the Lagrange multipliers are optimal, and (iv) the reduced Hessian is singular or there are some very small multipliers. This exit code cannot appear if H is positive definite.

One other caution about the return value 1, "Optimality conditions satisfied." Some of the variables or constraints might lie outside their bounds more than desired, especially if scaling was requested.

INFEASIBLE PROBLEMS

TABLE 5-15: OUTPUT FOR INFEASIBLE PROBLEMS

INFO FLAG	DESCRIPTION
0	The problem appears to be infeasible
11	Infeasible linear constraints
12	Infeasible linear equalities
13	Nonlinear infeasibilities minimized
14	Infeasibilities minimized

This exit code appears if the solver is unable to find a point that satisfies the constraints. When the constraints are linear, the output messages are based on a relatively reliable indicator of infeasibility. Feasibility is measured with respect to the upper and lower bounds. Violations as small as the (minor) feasibility tolerance (`feastol`) are ignored, but at least one component violates a bound by more than the tolerance.

When nonlinear constraints are present, infeasibility is much harder to recognize correctly. Even if a feasible solution exists, the current linearization of the constraints might not contain a feasible point. In an attempt to deal with this situation, when solving each quadratic subproblem, the nonlinear solvers are prepared to relax the nonlinear bounds.

If a quadratic subproblem proves to be infeasible or unbounded (or if the Lagrange multiplier estimates for the nonlinear constraints become large), the solvers enter a so-called Elastic mode (provided the `elastic` property is nonzero). The subproblem includes the original quadratic objective and the sum of the infeasibilities—suitably weighted using the elastic weight parameter. In Elastic mode, the bounds on the nonlinear constraints become "elastic"—that is, the constraints may violate their bounds. If the original problem has a feasible solution and the elastic weight is sufficiently large, a feasible point is eventually obtained for the perturbed constraints, and optimization can continue on the subproblem. If the problem has no feasible solution, the solvers tend to determine a "good" infeasible point if the elastic weight is sufficiently large.

Unfortunately, even though the solvers locally minimize the nonlinear constraint violations, there might still exist other regions in which the nonlinear constraints are

satisfied. Wherever possible, try to define nonlinear constraints in such a way that feasible points are known to exist when the constraints are linearized.

UNBOUNDED PROBLEMS

TABLE 5-16: OUTPUT FOR UNBOUNDED PROBLEMS

INFO FLAG	DESCRIPTION
20	The problem appears to be unbounded
21	Unbounded objective
22	Constraint violation limit reached

For linear problems, unboundedness is detected by the simplex method when a nonbasic variable can be increased or decreased by an arbitrary amount without causing a basic variable to violate a bound. Consider adding an upper or lower bound to the variable that is unbounded. Also examine the constraints that have nonzeros in the associated column to see if they have been formulated as intended. Very rarely, the scaling of the problem could be so poor that a numerical error gives an erroneous indication of unboundedness. Consider using the scale option `scaleopt`.

For nonlinear problems, the solvers monitor both the size of the current objective function and the size of the change in the variables at each step. If either of these is very large, the problem is terminated and declared unbounded. To avoid large function values, it might be necessary to impose bounds on some of the variables in order to keep them away from singularities in the nonlinear functions.

Flag 22 indicates an abnormal termination while enforcing the limit on the constraint violations. This exit code implies that the objective is not bounded below in the feasible region defined by expanding the bounds by the value of the violation limit `viollim`.

RESOURCE LIMIT ERRORS

TABLE 5-17: OUTPUT FOR RESOURCE LIMIT ERRORS

INFO FLAG	DESCRIPTION
30	Resource limit error
31	Iteration limit reached
32	(Major) iteration limit reached
33	The superbasics limit is too small

Either the iterations limit (`itlim`, see page 129) or the total iterations limit (`totitlim`, see page 138) was exceeded before the required solution could be found.

Check if the solver was making progress, for example by printing out the solution in the callback functions. If so, restart the run using the last solution as an initial one.

NUMERICAL DIFFICULTIES

TABLE 5-18: OUTPUT AFTER NUMERICAL DIFFICULTIES

INFO FLAG	DESCRIPTION
40	Terminated after numerical difficulties
41	Current point cannot be improved
42	Singular basis
43	Cannot satisfy the general constraints
44	Ill-conditioned null-space basis

Several circumstances can lead to the solvers not being able to improve on a non-optimal point:

- The user-supplied functions could be returning accurate function values but inaccurate gradients (or vice versa). This is the most likely cause. Study the comments for Flags 51 and 52 and do your best to ensure that the coding is correct.
- The function and gradient values could be consistent, but their precision could be too low. You might need to raise the default optimality tolerance (`opttol`) if it is not possible to increase the precision of the functions themselves.
- If function values are obtained from an expensive iterative process, they might be accurate to rather few significant figures, and gradients are probably not available. You should specify a function precision t and an optimality tolerance $\sqrt{t}$, but even then, if t is as large as 10^{-5} or 10^{-6} (only 5 or 6 significant figures), the same exit condition could occur. At present the only remedy is to increase the accuracy of the function calculation.

Termination because of a singular basis is highly unlikely to occur. The first factorization attempt finds the basis to be structurally or numerically singular. The modified basis is refactorized, but a singularity persists. This likely means that the problem is badly scaled.

If the general constraints cannot be satisfied, an LU factorization of the basis has just been obtained and used to recompute the basic variables. A step of "iterative refinement" has also been applied to increase the accuracy. However, a row check has revealed that the resulting solution does not satisfy the current constraints sufficiently well. This probably means that the current basis is very ill-conditioned. If there are

some linear constraints and variables, try setting `scaleopt` to 1 if scaling has not yet been used.

ERRORS IN USER-SUPPLIED FUNCTIONS

TABLE 5-19: OUTPUT FOR ERRORS IN USER-SUPPLIED FUNCTIONS

INFO FLAG	DESCRIPTION
50	Error in the user-supplied functions
51	Incorrect objective derivatives
52	Incorrect constraint derivatives
54	The QP Hessian is indefinite

This exit code implies that there might be errors in the functions that define the problem objective and constraints. If the objective derivatives appear to be incorrect, the solver has made a check on some individual elements of the objective gradient array at the first point that satisfies the linear constraints. At least one component in the user routines has been set to a value that disagrees markedly with its associated forward-difference estimate. (The relative difference between the computed and estimated values is 1.0 or more.) This exit code is a safeguard because the solvers usually fail to make progress when the computed gradients are seriously inaccurate. In the process the solver might expend considerable effort before terminating with Flag 41 ("Current point cannot be improved"). Check the function and gradient computation very carefully. A simple omission, such as forgetting to divide by two, could explain everything. If some component is very large, then give serious thought to scaling the function or the nonlinear variables.

If you feel certain that the computed objective gradient is correct (and that the forward-difference estimate is therefore wrong), you can specify a verification level of 0 (using the property `verify`; see page 138) to prevent individual elements from being checked. However, the optimization procedure might have difficulties.

If some constraint derivatives appear to be incorrect, then at least one of the computed constraint derivatives is significantly different from an estimate obtained by forward differences. Follow the advice given earlier for the objective function, trying to ensure that the constraints and their derivatives are set correctly.

Return Flag 54 refers to the quadratic solver and indicates that an indefinite matrix was detected during the computation of the reduced Hessian factor R such that $R^T R = Z^T H Z$. This might be caused by the matrix H being indefinite, that is, there might exist a vector y such that $y^T H y < 0$. In this case the QP problem is not convex and cannot be solved using `optquad`. You should check that H is correct and that all

relevant components of Hx are assigned their correct values, in the case when Hx is computed in a function.

If H is symmetric positive semidefinite, then the problem might be the ill-conditioning of the reduced Hessian caused by ill-conditioning in either H or Z.

PROBLEMS WITH UNDEFINED FUNCTIONS

TABLE 5-20: OUTPUT FOR UNDEFINED FUNCTION ERRORS

INFO FLAG	DESCRIPTION
60	Undefined user-supplied functions
61	Undefined function at the first feasible point
62	Undefined function at the initial point
63	Unable to proceed into undefined region

If the user calls `optsetstatus(-1)` in any unction then the problem is considered to be "undefined" at that point. The solvers attempt to evaluate the problem functions closer to a point at which the objective and constraints have already been computed. Flags 61 and 62 indicate that the solver is unable to proceed because the functions are undefined at the initial point or first feasible point.

Flag 63 implies that repeated attempts to move into a region where the functions are not defined resulted in the change in variables being unacceptably small.

USER REQUESTED TERMINATION

TABLE 5-21: OUTPUT FOR USER REQUESTED TERMINATION

INFO FLAG	DESCRIPTION
70	User requested termination
71	Terminated during function evaluation
72	Terminated during constraint evaluation
73	Terminated during objective evaluation
74	Terminated from monitor routine

These exit codes appear when some user-defined function has called `optsetstatus` for values < -1. The solvers assume that you want the problem to be abandoned immediately.

If the following exit codes appear during the first basis factorization, the primal and dual variables have their original input values.

INSUFFICIENT STORAGE

TABLE 5-22: OUTPUT FOR INSUFFICIENT STORAGE

INFO FLAG	DESCRIPTION
80-84	Insufficient storage allocated

These exit codes indicate that the solver ran out of memory. Be sure that the Hessian dimension is not unreasonably large.

INPUT ARGUMENTS OUT OF RANGE

TABLE 5-23: OUTPUT FOR INPUT ARGUMENT DIFFICULTIES

INFO FLAG	DESCRIPTION
90	Input arguments out of range
91	Invalid input argument

These exit codes appear if some data associated with the problem is out of range or invalid.

SYSTEM ERROR

TABLE 5-24: OUTPUT FOR SYSTEM ERRORS

INFO FLAG	DESCRIPTION
140	System error
141	Wrong number of basic variables
142	Error in basis package

These exit codes appear if some fatal system error has occurred. The solver abandons the problem.

6

Solver Properties

This chapter contains details about the available properties for the gradient-based solvers. These properties can be useful to tune the solvers' performance.

Gradient-Based Solver Properties

A list of the available solver properties appears in the "Command Reference" chapter in the entries for `optprop` on page 99 (properties for linear and quadratic solvers) and `optpropnlin` on page 103 (nonlinear solvers). The following sections provide more detailed explanations of these properties.

Note: In the following sections, ε represents the machine precision and is approximately equal to $2.2 \cdot 10^{-16}$. The machine precision is available as `eps` in COMSOL Script.

Cendiff

Central difference interval
Type: numeric
Default: $\varepsilon^{1/3} \approx 6.0 \cdot 10^{-6}$
Applicable for: Nonlinear solvers

When some problem derivatives are unknown, the solver uses the central difference interval near an optimal solution to obtain more accurate (but more expensive) estimates of gradients. Twice as many function evaluations are required compared to forward differencing. If r is the central difference interval, the interval used for the jth variable is $h_j = r(1 + |x_j|)$. The resulting derivative estimates should be accurate to $O(r^2)$, unless the functions are badly scaled.

Checkfreq

Check frequency
Type: integer
Default = 60
Applicable for: All solvers

Every ith iteration after the most recent basis factorization, the solver makes a numerical test to see if the current solution x satisfies the general linear constraints (including linearized nonlinear constraints, if any). The constraints are of the form $Ax - s = b$ where s is the set of slack variables. To perform the numerical test, the solver

computes the residual vector $r = b - Ax + s$. If the largest component of r is judged to be too large, the current basis is refactorized and the basic variables are recomputed to satisfy the general constraints more accurately.

`checkfreq = 1` is useful for debugging purposes, but otherwise this option should not be needed.

Diffint

Difference interval
Type: numeric
Default: $\varepsilon^{1/2} \approx 1.5 \cdot 10^{-8}$
Applicable for: Nonlinear solvers

This property alters the interval that the solver uses to estimate gradients by forward differences in the following circumstances:

- In the initial ("cheap") phase of verifying the problem derivatives
- For verifying the problem derivatives
- For estimating missing derivatives

In all cases, the solver estimates a derivative with respect to x_j by perturbing that component of x to the value $x_j + h_1(1 + |x_j|)$, where h_1 is the difference interval, and then evaluating the goal function at the perturbed point.

The resulting gradient estimates should be accurate to $O(h_1)$ unless the functions are badly scaled. Judicious alteration of the difference interval can sometimes lead to greater accuracy.

Elastic

Elastic mode
Type: integers 0, 1, *or* 2
Default: 1
Applicable for: Linear and quadratic solvers

This parameter determines if (and when) Elastic mode is started. Three variations are available:

TABLE 6-1: ELASTIC MODES

MODE	DESCRIPTION
0	Elastic mode is never invoked. The solver terminates as soon as it detects infeasibility. There might be other points with significantly smaller sums of infeasibilities.
1	Elastic mode is invoked only if the constraints are found to be infeasible. If so, the solver continues in the Elastic mode with the composite objective determined by the values of properties `elasticobj` (elastic objective) and `elasticw` (elastic weight).
2	The iterations start and remain in Elastic mode. This option allows you to minimize the composite objective function directly without first performing phase–1 iterations (also sometimes called the feasibility phase, which minimizes the sum of infeasibilities to find a feasible point). The success of this option depends critically on the choice of elastic weight. If that value is sufficiently large and the constraints are feasible, the minimizer of the composite objective and the solution of the original problem are identical. However, if the elastic weight is not sufficiently large, the minimizer of the composite function might be infeasible even though a feasible point for the constraints could exist.

Elasticobj

Elastic objective
Type: integers 0, 1, *or* 2
Default: 2
Applicable for: Linear and quadratic solvers

This option determines the form of the composite objective. Three types of composite objective are available:

TABLE 6-2: ELASTIC OBJECTIVE MODES

MODE	DESCRIPTION
0	Include only the true objective in the composite objective. This option sets the elastic weight to 0 in the composite objective and allows the solver to ignore the elastic bounds and find a solution that minimizes the objective function subject to the nonelastic constraints. This option is useful if there are some "soft" constraints that you would like to ignore if the constraints are infeasible.
1	Use a composite objective defined with elastic weight determined by the value of the `elasticw` property. This value is intended to be used in conjunction with Elastic mode set to 2.
2	Include only the elastic variables in the composite objective. The elastics are weighted by 1. This choice minimizes the violations of the elastic variable at the expense of possibly increasing the true objective. This option can be used to find a point that minimizes the sum of the violations of a subset of constraints determined by the parameters `elasticbc` and `elasticlc`.

Elasticbc

Elastic bound constraints
Type: scalar or vector of 0, 1, 2, *or* 3
Default: 0
Applicable for: Linear and quadratic solvers

Indicates which bound constraints can be elastic:

- 0—corresponding constraints cannot be violated.
- 1—the lower bound can be violated.
- 2—the upper bound can be violated.
- 3—either bound can be violated.

Elasticlc

Elastic linear constraints
Type: scalar or vector of 0, 1, 2, *or* 3

Default: 3
Applicable for: Linear and quadratic solvers

Indicates which linear constraints may be elastic.

- `0`—corresponding constraints cannot be violated.
- `1`—the lower bound can be violated.
- `2`—the upper bound can be violated.
- `3`—either bound can be violated.

Elasticw

Elastic weight
Type: numeric
Default: 1.0 *for linear and quadratic problems,* $1.0 \cdot 10^4$ *for nonlinear problems*
Applicable for: Linear and quadratic solvers

This property determines the initial weight associated with the elastic QP problem. For more details, see the SNOPT and SQOPT manuals. In general, in Elastic mode, if the original problem has a feasible solution and the elastic weight is sufficiently large, a feasible point is eventually obtained for the perturbed constraints and optimization can continue.

Expfreq

Expand frequency
Type: integer
Default: 10,000
Applicable for: All solvers

This option is part of the internal procedure designed to make progress even on highly degenerate problems.

For linear models, the strategy is to force a positive step at every (minor) iteration at the expense of violating the bounds on the variables by a small amount. Suppose that the expand frequency is i and the feasibility tolerance (property `feastol`) is δ. Over a period of i iterations, the tolerance the solvers actually use increases from 0.5δ to δ (in steps of $0.5\delta/i$).

For nonlinear models, the same procedure is used for iterations in which there is only one superbasic variable. (Cycling can occur only when the current solution is at a

vertex of the feasible region.) Thus, zero steps are allowed if there is more than one superbasic variable, but otherwise positive steps are enforced.

Increasing *i* helps reduce the number of slightly infeasible nonbasic variables (most of which are eliminated during a resetting procedure). However, it also diminishes the freedom to choose a large pivot element (see `pivtol` on page 134).

Facfreq

Factorization frequency
Type: integer
Default: 100 *for linear problems,* 50 *for quadratic or nonlinear problems*
Applicable for: All solvers

At most k basis changes occur between factorizations of the basis matrix, where k is the factorization frequency.

With linear programs, the basis factors are usually updated every iteration. The default k is reasonable for typical problems. Higher values to $k = 100$ might be more efficient on problems that are extremely sparse and well scaled.

When the objective function is nonlinear or quadratic, fewer basis updates occur as an optimum is approached. The number of iterations between basis factorizations therefore increases. During these iterations a test is made regularly (according to the check frequency, `Checkfreq`) to ensure that the general constraints are satisfied. If necessary the basis is refactorized before the limit of k updates is reached.

Feastol

Feasibility tolerance
Type: numeric
Default: $1.0 \cdot 10^{-6}$
Applicable for: All solvers

The solvers try to ensure that all bound and linear constraints are eventually satisfied to within the feasibility tolerance t. (Feasibility with respect to nonlinear constraints is instead judged by the major feasibility tolerance, `majfeastol`.)

If the bounds and linear constraints cannot be satisfied to within t, the problem is declared infeasible. Let **sInf** be the corresponding sum of infeasibilities. If **sInf** is quite small, it might be appropriate to raise t by a factor of 10 or 100. Otherwise you should suspect some error in the data.

Nonlinear functions are evaluated only at points that satisfy the bound and linear constraints. If there are regions where a function is undefined, every attempt should be made to eliminate these regions from the problem. For example, if $f(x) = \sqrt{x_1} + \log x_2$, it is essential to place lower bounds on both variables. If $t = 10^{-6}$, the bounds $x_1 \geq 10^{-5}$ and $x_2 \geq 10^{-4}$ might be appropriate. (The log singularity is more serious. In general, keep x as far away from singularities as possible.)

If `scaleopt` (see page 136) is `1`, feasibility is defined in terms of the scaled problem (because it is then more likely to be meaningful).

In practice, the nonlinear solvers use t as a feasibility tolerance for satisfying the bound and linear constraints in each QP subproblem. If the sum of infeasibilities cannot be reduced to zero, the QP subproblem is declared infeasible. The solver is then in the Elastic mode thereafter (with only the linearized nonlinear constraints defined to be elastic).

For the quadratic and linear solvers, note that if **sInf** is not small and you have not asked the solver to minimize the violations of the elastic variables (that is, you have not specified `elastobj = 2`), other points might have a significantly smaller sum of infeasibilities. The solvers do not attempt to find the solution that minimizes the sum unless `elastobj = 2`. (See `Elasticobj` on page 122 for further details.)

Funcprec

Function precision
Type: numeric
Default: $\varepsilon^{0.8} \approx 3.8 \cdot 10^{-11}$
Applicable for: Nonlinear solvers

The relative function precision is intended to be a measure of the relative accuracy with which the nonlinear functions can be computed. For example, if $f(x)$ is computed as 1000.56789 for some relevant x and if the first 6 significant digits are known to be correct, the appropriate value for the function precision would be `1e-6`. (Ideally the functions should have a magnitude of order 1. If all functions are substantially less than 1 in magnitude, the function precision should be the absolute precision. For example, if $f(x) = 1.23456789 \cdot 10^{-4}$ at some point and if the first 6 significant digits are known to be correct, the appropriate precision would be `1e-10`.)

The default value is appropriate for simple analytic functions.

In some cases the function values are the result of extensive computations, possibly involving an iterative procedure that can provide rather few digits of precision at

reasonable cost. Specifying an appropriate function precision might lead to savings by allowing the line search procedure to terminate when the difference between function values along the search direction becomes as small as the absolute error in the values.

Hessdim

Hessian dimension
Type: numeric
Default: $\min\{1000, n_1 + 1\}$*, where* n_1 *in the nonlinear case is the number of nonlinear variables, in the quadratic case is the number of leading nonzero columns of the Hessian, and in the linear case is 0.*
Applicable for: All solvers

Let r be the value given by the `hessdim` property. This specifies that an r-by-r triangular matrix R is to be available for use by the Cholesky QP solver (to define the reduced Hessian according to $RTR = Z^T HZ$). See the SNOPT and SQOPT user's manuals for further details.

Hessfreq

Hessian frequency
Type: numeric
Default: 999,999
Applicable for: Nonlinear solvers

If the `hessmem` property is set to `'full'` and `hessfreq` BFGS updates have already been carried out, the Hessian approximation is reset to the identity matrix. (For certain problems occasional resets might improve convergence, but in general they should not be necessary.) `hessmem` set to `'full'` and `hessfreq` set to `20` have a similar effect to `hessmem` set to `'limited'` and `hessupd` set to `20` (except that the latter retains the current diagonal during resets).

Hessmem

Hessian memory
Type: string 'full' or 'limited'
Default: 'full' if the number of nonlinear variables is ≤ 75*. When QP problem solver is set to conjugate-gradient, the default is always 'limited'.*
Applicable for: Nonlinear solvers

This option selects the method for storing and updating the approximate Hessian. (The nonlinear solvers use a quasi-Newton approximation to the Hessian of the Lagrangian. A BFGS update is applied after each major iteration.)

If Hessian `full` memory is specified, the approximate Hessian is treated as a dense matrix and the BFGS updates are applied explicitly. This option is most efficient when the number of nonlinear variables is not too large (say, less than 75). In this case, the storage requirement is fixed and you can expect 1-step Q-superlinear convergence to the solution.

Hessian `limited` memory should be used on problems where the number of nonlinear variables is very large. In this case a limited-memory procedure is used to update a diagonal Hessian approximation a limited number of times.

Hessupd

Hessian updates
Type: integer
Default: 10
Applicable for: Nonlinear solvers

If `hessmem` is set to `limited` memory and `hessupd` BFGS updates have already been carried out, all but the diagonal elements of the accumulated updates are discarded and the updating process starts again. Broadly speaking, the more updates stored, the better the quality of the approximate Hessian. However, the more vectors stored, the greater the cost of each QP iteration. The default value is likely to give a robust algorithm without significant expense, but faster convergence can sometimes be obtained with significantly fewer updates (e.g., `hessupd = 5`).

Infbound

Infinite bound size
Type: positive numeric
Default: $1.0 \cdot 10^{20}$
Applicable for: All solvers

Defines the "infinite" bound in the definition of the problem constraints. Any upper bound greater than or equal to this bound is regarded as plus infinity (and similarly for a lower bound less than or equal to `-infbound`).

Itlim

Iterations limit
Type: nonnegative integer
Default: 500 *for nonlinear solvers, otherwise* 3*m*, *where* *m* *is the number of general constraints*
Applicable for: All solvers

For the linear and quadratic solvers, this is the maximum number of allowed iterations of the simplex method or the QP reduced-gradient algorithm. It is allowable to set `itlim` to 0 whereby the solver checks both feasibility and optimality.

For the nonlinear solvers, this is the number of minor iterations for the optimality phase of the QP subproblem. If `itlim` is exceeded, then all nonbasic QP variables that have not yet moved are frozen at their current values and the reduced QP is solved to optimality.

Note that more than `itlim` minor iterations might be necessary to solve the reduced QP to optimality. These extra iterations are necessary to ensure that the terminated point gives a suitable direction for the line search.

Note that `totitlim` (total iterations limit; see page 138) defines an independent absolute limit on the total number of minor iterations (summed over all QP subproblems).

Linesearch

Linesearch method
Type: string 'derivative' or 'nonderivative'
Default: 'derivative'
Applicable for: Nonlinear solvers

At each major iteration a line search is used to improve the merit function. A `derivative` linesearch uses safeguarded cubic interpolation and requires both function and gradient values to compute estimates of the step. If some analytic derivatives are not provided or a nonderivative linesearch is specified, the solver employs a line search based upon safeguarded quadratic interpolation, which does not require gradient evaluations.

A `nonderivative` line search can be slightly less robust on difficult problems, and we recommend use of the default if the functions and derivatives can be computed at approximately the same cost. If the gradients are very expensive relative to the

functions, a nonderivative line search might give a significant decrease in computation time.

Linestol

Linesearch tolerance
Type: numeric
Default: 0.9
Applicable for: Nonlinear solvers

This parameter controls the accuracy with which a step length is located along the direction of search during each iteration. At the start of each line search, the solver identifies a target directional derivative for the merit function. This parameter determines the accuracy to which this target value is approximated.

`linestol` must be a real value in the range 0 to 1. The default value of 0.9 requests just moderate accuracy in the line search. If the nonlinear functions are cheap to evaluate, a more accurate search might be appropriate; try a `linestol` value of 0.1, 0.01 or 0.001. The number of major iterations might decrease.

If the nonlinear functions are expensive to evaluate, a less accurate search might be appropriate. If all gradients are known, try `linestol` = 0.99. (The number of major iterations might increase, but the total number of function evaluations could decrease enough to compensate.)

If not all gradients are known, a moderately accurate search remains appropriate. Each search requires only one to five function values (typically), but many function calls are then needed to estimate missing gradients for the next iteration.

Majfeastol

Major feasibility tolerance
Type: numeric
Default: $1.0 \cdot 10^{-6}$
Applicable for: Nonlinear solvers

This parameter specifies how accurately the nonlinear constraints should be satisfied. The default value of $1.0 \cdot 10^{-6}$ is appropriate when the linear and nonlinear constraints contain data to roughly that accuracy.

Let **rowerr** be the maximum nonlinear constraint violation, normalized by the size of the solution. It is required to satisfy

$$\texttt{rowerr} = \max_i \texttt{viol}_i/(\|x\| + 1) \leq \texttt{majfeastol}$$

where $\texttt{viol}_i$ is the violation of the ith nonlinear constraint. If some of the problem functions are known to be of low accuracy, a larger major feasibility tolerance might be appropriate.

Majitlim

Major iterations limit
Type: numeric
Default: max(1000, m), *where* m *is the number of general constraints*
Applicable for: Nonlinear solvers

This is the maximum number of major iterations allowed. It is intended to guard against an excessive number of linearizations of the constraints.

Majsteplim

Major step limit
Type: numeric
Default: 2.0
Applicable for: Nonlinear solvers

This parameter limits the change in x during a line search. It applies to all nonlinear problems once the solver has found a "feasible solution" or "feasible subproblem."

- A line search determines a step α over the range $0 < \alpha \leq \beta$, where β is 1 if there are nonlinear constraints, or the step to the nearest upper or lower bound on x if all the constraints are linear. Normally, the first step length tried is $\alpha_1 = \min(1, \beta)$.
- In some cases, such as $f(x) = ae^{bx}$ or $f(x) = ax^b$, even a moderate change in the components of x can lead to floating-point overflow. The parameter `majsteplim` is therefore used to define a limit $\bar{\beta} = r(1 + \|x\|)/\|p\|$ (where p is the search direction and r the value of `majsteplim`), and the first evaluation of $f(x)$ is at the potentially smaller step length $\alpha_1 = \min(1, \bar{\beta}, \beta)$.
- Wherever possible, use upper and lower bounds on x s to prevent evaluation of nonlinear functions at meaningless points. The major step limit provides an additional safeguard. The default value `majsteplim` = 2.0 should not affect progress on well-behaved problems, but setting it to 0.1 or 0.01 might be helpful when rapidly varying functions are present. A "good" starting point might be

required. An important application is to the class of nonlinear least-squares problems.

- In cases where several local optima exist, specifying a small value for `majsteplim` could help locate an optimum near the starting point.

Maximize

Maximize objective
Type: string 'on' or 'off'
Default: 'off'
Applicable for: All solvers

Setting this property to `on` causes the objective to be maximized instead of minimized.

Opttol

Optimality tolerance
Type: numeric
Default: $1.0 \cdot 10^{-6}$
Applicable for: All solvers

The linear and quadratic solvers use this parameter to judge the size of the reduced gradients $d_j = g_j - \pi^T a_j$, where g_j is the jth component of the gradient, a_j is the associated column of the constraint matrix $(A - I)$, and π is the set of dual variables.

By construction, the reduced gradients for basic variables are always 0. Let t be the optimality tolerance. The problem is declared optimal if the reduced gradients for nonbasic variables at their lower or upper bounds satisfy

$$d_j / \|\pi\| \geq -t \text{ or } d_j / \|\pi\| \leq t$$

respectively, and if $d_j / \|\pi\| \leq t$ for superbasic variables.

In these tests, $\|\pi\|$ is a measure of the size of the dual variables. It is included to make the tests independent of a scale factor on the objective function. The quantity $\|\pi\|$ actually used is defined by

$$\|\pi\| = \max \{\sigma / \sqrt{m}, 1\}, \text{ where } \sigma = \sum_{i=1}^{m} |\pi_i|$$

so that only large scale factors are allowed.

If the objective is scaled down to be very small, the optimality test reduces to comparing d_j against $0.01t$.

For the nonlinear solvers, `opttol` is the major optimality tolerance and specifies the final accuracy of the dual variables. On successful termination, the solvers compute a solution (x, s, π) such that

$$\text{maxComp} = \max_j \ \text{Comp}_j / \|\pi\| \le \text{opttol}$$

where Comp_j is an estimate of the complementarity slackness for variable j. The values Comp_j are computed from the final QP solution using the reduced gradients $d_j = g_j - \pi^T a_j$, as above. Hence you have

$$\text{Comp}_j = \begin{cases} d_j \min\ \{x_j - l_j, 1\} & \text{if } d_j \ge 0 \\ -d_j \min\ \{u_j - x_j, 1\} & \text{if } d_j < 0 \end{cases}$$

See the SNOPT user's manual for further details.

Newsuplim

New superbasics limit
Type: integer
Default: 99
Applicable for: Nonlinear solvers

This option causes early termination of the QP subproblems if the number of free variables has increased significantly since the first feasible point. If the number of new superbasics is greater than `newsuplim`, the nonbasic variables that have not yet moved are frozen and the resulting smaller QP is solved to optimality.

Parprice

Partial price
Type: integer
Default: 10 *for linear problems,* 1 *for quadratic*
Applicable for: Linear and quadratic solvers

This parameter is recommended for large problems that have significantly more variables than constraints. It reduces the work required for each "pricing" operation (when a nonbasic variable is selected to become superbasic).

When the partial price is 1, all columns of the constraint matrix $(A - I)$ are searched. Otherwise, A and I are partitioned to give the partial price i roughly equal segments A_j, I_j ($j = 1$ to i). If the previous pricing search was successful on A_j, I_j, the next search begins on the segments A_{j+1}, I_{j+1}. (All subscripts here are modulo i.)

If a reduced gradient is found that is larger than some dynamic tolerance, the variable with the largest such reduced gradient (of appropriate sign) is selected to become superbasic. If nothing is found, the search continues on the next segments A_{j+2}, I_{j+2}, and so on.

Partial price t (or $t/2$ or $t/3$) might be appropriate for time-stage models having t time periods.

Pivtol

Pivot tolerance
Type: numeric
Default: $3.7 \cdot 10^{-11}$
Applicable for: All solvers

During solution of QP subproblems, the solver uses the pivot tolerance to prevent columns entering the basis if they would cause the basis to become almost singular.

When x changes to $x + \alpha p$ for some search direction p, a "ratio test" is used to determine which component of x first reaches an upper or lower bound. The corresponding element of p is called the pivot element.

Elements of p are ignored (and therefore cannot be pivot elements) if they are smaller than the pivot tolerance.

It is common for two or more variables to reach a bound at essentially the same time. In such cases, the (minor) feasibility tolerance (say t) provides some freedom to maximize the pivot element and thereby improve numerical stability. Excessively small values of t should therefore not be specified.

To a lesser extent, the expand frequency (property `expfreq`) also provides some freedom to maximize the pivot element. Excessively large values of `expfreq` should therefore not be specified.

Print

Print information about the solver progress and solution to file
Type: string
Default: empty
Applicable for: All solvers

When the print option is activated, the following information is output to the file during the solution process. All printed lines are less than 131 characters.

- An estimate of the working storage needed and the amount available.
- Some statistics about the problem being solved.
- The storage available for the LU factors of the basis matrix.
- A summary of the scaling procedure, if `scaleopt` > 0.
- Notes about the initial basis.
- The iteration log.
- Basis factorization statistics.
- The exit condition and some statistics about the solution obtained.
- The printed solution, if requested.

For a more detailed overview of the various sections of the print files, see the SQOPT and SNOPT user's manuals.

Proxmeth

Proximal point method
Type: 1 *or* 2
Default: 1
Applicable for: Nonlinear solvers

`proxmeth` set to `1` or `2` specifies minimization of $\|x - x_0\|_1$ or $\frac{1}{2}\|x - x_0\|_2^2$, respectively, when the starting point x_0 is changed to satisfy the linear constraints (where x_0 refers to nonlinear variables).

Qpsolver

QP problem solver
Type: string 'cholesky', 'cg', or 'qn'

Default: 'cholesky'
Applicable for: All solvers

Specifies the active-set algorithm used to solve the QP problem, or in the nonlinear case, the QP subproblem.

`'cholesky'` holds the full Cholesky factor R of the reduced Hessian Z^THZ. As the QP iterations proceed, the dimension of R changes with the number of superbasic variables. If the number of superbasic variables increases beyond the value of reduced Hessian dimension (property `Hessdim`), the reduced Hessian cannot be stored and the solver switches to `qpsolver = 'cg'`.

The Cholesky solver is reactivated if the number of superbasics stabilizes at a value less than the reduced Hessian dimension.

`'qn'` solves the QP subproblem using a quasi-Newton method. In this case, R is the factor of a quasi-Newton approximate Hessian.

`'cg'` uses an active-set method similar to `'qn'` but uses the conjugate-gradient method to solve all systems involving the reduced Hessian.

The Cholesky QP solver is the most robust but might require a significant amount of computation if the number of superbasics is large.

The quasi-Newton QP solver does not require the computation of the exact R at the start of each QP and might be appropriate when the number of superbasics is large but each QP subproblem requires relatively few minor iterations.

The conjugate-gradient QP solver is appropriate for problems with large numbers of degrees of freedom (many superbasic variables). The Hessian memory option `'hessmem'` is defaulted to `'limited'` when this solver is used.

Scaleopt

Scale option
Type: integers 0, 1, *or* 2
Default: 1 *for quadratic and nonlinear problems,* 2 *for linear problems*
Applicable for: All solvers

Three scale options are available:

TABLE 6-3: SCALE OPTIONS

SCALEOPT	DESCRIPTION
0	No scaling. This is recommended if it is known that x and the constraint matrix (and Jacobian) never have very large elements (say, larger than 1000).
1	Linear constraints and variables are scaled by an iterative procedure that attempts to make the matrix coefficients as close as possible to 1.0. This sometimes improves the performance of the solution procedures.
2	All constraints and variables are scaled by the iterative procedure. Also, an additional scaling is performed that takes into account columns of $(A - I)$ that are fixed or have positive lower bounds or negative upper bounds. If nonlinear constraints are present, the scales depend on the Jacobian at the first point that satisfies the linear constraints. Scale option 2 should therefore be used only if a good starting point is provided and the problem is not highly nonlinear.

Scaletol

Scale tolerance
Type: numeric
Default: 0.9
Applicable for: All solvers

Scale tolerance affects how many passes might be needed through the constraint matrix. On each pass, the scaling procedure computes the ratio of the largest and smallest nonzero coefficients in each column:

$$\rho_j = (\max_i |a_{ij}| / \min_i |a_{ij}|) \qquad (a_{ij} \neq 0)$$

If $\max_j \rho_j$ is less than `scaletol` times its previous value, another scaling pass is performed to adjust the row and column scales. Raising the scale tolerance from 0.9 to 0.99 (for instance) usually increases the number of scaling passes through A. At most 10 passes are made.

Suplim

Superbasics limit
Type: integer
Default: $n_1 + 1$, *where* n_1 *in the nonlinear case is the number of nonlinear variables,*

in the quadratic case is the number of leading nonzero columns of the Hessian, and in the linear case is 0.
Applicable for: All solvers

This parameter places a limit on the storage allocated for superbasic variables. Ideally, `suplim` should be set slightly larger than the "number of degrees of freedom" expected at an optimal solution.

For linear programs, an optimum is normally a basic solution with no degrees of freedom. (The number of variables lying strictly between their bounds is no more than m, the number of general constraints.) The default value of `suplim` is therefore 1.

The number of degrees of freedom is often called the "number of independent variables."

For quadratic problems, `suplim` normally need not be greater than the number of leading nonzero columns of H. For many problems, `suplim` might be considerably smaller than that, which saves storage if the number of leading nonzero columns is very large.

For nonlinear problems, `suplim` normally need not be greater than $n_1 + 1$, where n_1 is the number of nonlinear variables. For many problems it might be considerably smaller than n_1. This saves storage if n_1 is very large.

Totitlim

Total iterations limit
Type: numerical
Default: $\max\{10000, 20m\}$*, where* m *is the number of general constraints*
Applicable for: Nonlinear solvers

This is the maximum number of minor iterations allowed (that is, iterations of the simplex method or the QP algorithm), summed over all major iterations.

Verify

Verification level
Type: integers -1, 0, 1, 2, *or* 3
Default: 0
Applicable for: Nonlinear solvers

This option refers to finite-difference checks on the derivatives computed by user-provided routines. Derivatives are checked at the first point that satisfies all bounds and linear constraints.

TABLE 6-4: THE VERIFY OPTION

VERIFY	DESCRIPTION
-1	Derivative checking is disabled.
0	Only a "cheap" test is performed, requiring 2 calls to user functions.
1	Individual objective gradients are checked (with a more reliable test).
2	Individual columns of the problem Jacobian are checked.
3	Options 2 and 1 both occur (in that order).

Verify level 3 is intended mainly for use when developing a new function routine. Missing derivatives are not checked, so they result in no overhead.

Viollim

Violation limit
Type: numeric
Default: 10
Applicable for: Nonlinear solvers

This keyword defines an absolute limit on the magnitude of the maximum constraint violation after the line search. On completion of the line search, the new iterate x_{k+1} satisfies the condition

$$v_i x_{k+1} \leq \tau \max \{1, v_i(x_0)\}$$

where x_0 is the point at which the nonlinear constraints are first evaluated, and $v_i(x)$ is the ith nonlinear constraint violation $v_i(x) = \max(0, l_i - f_i(x), f_i(x) - u_i)$, where l_i and u_i are the lower and upper bounds, respectively.

The effect of this violation limit is to restrict the iterates to lie in an expanded feasible region whose size depends on the magnitude of τ. This makes it possible to keep the iterates within a region where the objective is expected to be well defined and bounded below. If the objective is bounded below for all values of the variables, then τ can be any large positive value.

7

Glossary

This glossary contains important terms in the Optimization Lab.

Glossary of Terms

bound constraints Constraints that bound the variables: $x_{\mathrm{lb}} \le x \le x_{\mathrm{ub}}$.

cost function See *objective function*.

goal function See *objective function*.

linear constraints Constraints that bound a linear transformation of the variables: $b_{\mathrm{lb}} \le Ax \le b_{\mathrm{ub}}$.

nonlinear constraints Constraints that bound an arbitrary function of the variables: $d_{\mathrm{lb}} \le c(x) \le d_{\mathrm{ub}}$.

objective function The value of this function is what the optimization process attempts to minimize.

Opt structure A COMSOL Script structure that contains the entire optimization problem and also solution data after the Opt structure is passed to one of the optimization routines.

INDEX