

MATRIXx™

Xmath™ Optimization Module User Manual

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599,
Canada (Calgary) 403 274 9391, Canada (Ottawa) 613 233 5949, Canada (Québec) 450 510 3055,
Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530, China 86 21 6555 7838,
Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 385 0 9 725 725 11,
France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427, India 91 80 51190000,
Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400,
Malaysia 603 9131 0918, Mexico 001 800 010 0793, Netherlands 31 0 348 433 466,
New Zealand 0800 553 322, Norway 47 0 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210,
Russia 7 095 783 68 51, Singapore 65 6226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197,
Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227,
Thailand 662 992 7519, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send email to techpubs@ni.com.

© 2000–2004 National Instruments Corporation. All rights reserved.

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THEREOF PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

MATRIX[™], National Instruments[™], NI[™], ni.com[™], SystemBuild[™], and Xmath[™] are trademarks of National Instruments Corporation.

Product and company names mentioned herein are trademarks or trade names of their respective companies.

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or ni.com/patents.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Conventions

The following conventions are used in this manual:

» The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.



This icon denotes a note, which alerts you to important information.

bold

Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.

italic

Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.

`monospace`

Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.

`monospace bold`

Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.

`monospace italic`

Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

Contents

Chapter 1

Introduction

| | |
|---------------------------------------|-----|
| Using This Manual..... | 1-1 |
| Document Organization..... | 1-1 |
| Commonly Used Nomenclature | 1-2 |
| Related MATRIXx Publications | 1-2 |
| Additional Related Publications | 1-3 |
| MATRIXx Help..... | 1-3 |
| Getting Started | 1-3 |

Chapter 2

Nonlinear Programming

| | |
|--|------|
| Optimize() | 2-1 |
| Running Optimize()..... | 2-2 |
| Practical Considerations | 2-5 |
| Finding a Feasible Solution | 2-5 |
| Specifying the Penalty Parameter..... | 2-5 |
| Evaluating Results | 2-6 |
| Reducing Constraints to Improve Performance and Efficiency | 2-7 |
| Avoiding Discontinuities..... | 2-8 |
| Controlling the Numbers of Major and Minor Iterations | 2-10 |
| Optimize Algorithm..... | 2-11 |
| Application Examples..... | 2-13 |
| Box Design | 2-14 |
| Formulation | 2-15 |
| Optimization..... | 2-15 |
| Trajectory Optimization (Zermelo’s Problem)..... | 2-17 |
| Problem Definition..... | 2-17 |
| Formulation | 2-17 |
| Optimization..... | 2-21 |
| Analysis..... | 2-23 |
| Feedback Control Design | 2-24 |
| Problem Definition..... | 2-24 |
| Optimization..... | 2-28 |
| Analysis..... | 2-30 |

Chapter 3 Quadratic Programming

| | |
|---|-----|
| QPOPT() Function..... | 3-1 |
| QPOPT Algorithm..... | 3-2 |
| Application Example..... | 3-3 |
| Curve Fitting with Quadratic Programming..... | 3-3 |
| Solving the Curve Fitting Problem..... | 3-5 |

Chapter 4 Linear Programming

| | |
|----------------------------|-----|
| LPOPT() Function..... | 4-1 |
| LPOPT Algorithm..... | 4-2 |
| Application Example..... | 4-3 |
| Refinery Optimization..... | 4-3 |
| Formulation..... | 4-4 |
| Optimization..... | 4-5 |
| Analysis..... | 4-6 |

Appendix A Technical Support and Professional Services

Index

Introduction

This chapter starts with an outline of the manual and some use notes. It also provides information about getting started with the Optimization Module.

Using This Manual

This manual is a guide to performing engineering optimization using the Xmath Optimization Module.

To get the most out of the Optimization Module, you will need a working knowledge of Xmath and its MathScript analysis language. If you have never used Xmath, refer to the *Xmath User Guide*.

Document Organization

Chapters 2 through 4 of this manual focus on a single function. They define the mathematics of the problem solved by the function and provide step-by-step examples of problem solving sessions.

This manual includes the following chapters:

- Chapter 1, *Introduction*, starts with an outline of the manual and some use notes. It also provides information about getting started with the Optimization Module.
- Chapter 2, *Nonlinear Programming*, details the `optimize` function for general nonlinear programming. The `optimize` function is the most general and powerful function. It is fully integrated with Xmath and SystemBuild.
- Chapter 3, *Quadratic Programming*, details the quadratic programming function `qpopt`.
- Chapter 4, *Linear Programming*, details the linear optimization programming function `lpopt`.

Commonly Used Nomenclature

This manual uses the following general nomenclature:

- Matrix variables are generally denoted with capital letters; vectors are represented in lowercase.
- $G(s)$ is used to denote a transfer function of a system where s is the Laplace variable. $G(q)$ is used when both continuous and discrete systems are allowed.
- $H(s)$ is used to denote the frequency response, over some range of frequencies of a system where s is the Laplace variable. $H(q)$ is used to indicate that the system can be continuous or discrete.
- A single apostrophe following a matrix variable, for example, x' , denotes the transpose of that variable. An asterisk following a matrix variable (for example, A^*) indicates the complex conjugate, or Hermitian, transpose of that variable.

Related MATRIXx Publications

For a complete list of MATRIXx publications, refer to Chapter 2, *MATRIXx Publications, Help, and Customer Support*, of the *MATRIXx Getting Started Guide*. The following MATRIXx publications are particularly useful for topics covered in this manual:

- *MATRIXx Getting Started Guide*
- *Xmath User Guide*
- *Control Design Module*
- *Interactive Control Design Module*
- *Interactive System Identification Module, Part 1*
- *Interactive System Identification Module, Part 2*
- *Model Reduction Module*
- *Optimization Module*
- *Robust Control Module*
- *X μ Module*

Additional Related Publications

The following additional references on optimization also are useful for topics covered in this manual:

- “An Extension of Karmarkar’s Algorithm and the Trust Region Method for Quadratic Programming,” *Progress in Mathematical Programming*, Y. Y. Ye, N. Megiddo, editor, Springer-Verlag, 1989.
- *Linear and Nonlinear Programming*, Luenberger, D. G., Addison Wesley Publishing Company, 1987.
- Ph.D. Dissertation, Y. Y. Ye, Department of Engineering, Economic Systems, Stanford University, 1987.
- *Practical Optimization*, Gill, P. E., Murray, W. and Wright, M. H., Academic Press, 1981.

MATRIXx Help

Optimization Module function reference information is available in the *MATRIXx Help*. The *MATRIXx Help* includes all Optimization functions. Each topic explains a function’s inputs, outputs, and keywords in detail. Refer to Chapter 2, *MATRIXx Publications, Help, and Customer Support*, of the *MATRIXx Getting Started Guide* for complete instructions on using the Help feature.

Getting Started

Before using the Optimization Module, you should feel comfortable with the following:

- Creating/editing text files in your computer’s operating system
- Creating Xmath MathScript functions (MSFs)
- Creating, editing, and addressing vectors and matrices
- Saving and loading data
- Plotting

If you intend to use the `optimize()` function with nonlinear dynamic systems, you also should familiarize yourself with SystemBuild. You will want to know how to do the following:

- Build system models
- Linearize nonlinear models
- Generate time responses

When you know the Xmath and SystemBuild basics, you are ready to begin optimization. Complete function syntax and examples are available online. This document discusses details pertaining to each algorithm and suggests problem solving techniques. Application examples are provided for each function.

Each example includes the following procedures:

- Problem Setup
- Syntax Options
- Analysis

Nonlinear Programming

The `optimize()` function solves the general nonlinear programming problem shown in Equation 2-1.

$$\begin{aligned}
 & \min_p F(p) \\
 & G(p) = 0 \\
 & h_l \leq H(p) \leq h_u \\
 & p_l \leq p \leq p_u
 \end{aligned} \tag{2-1}$$

where p is the $n \times 1$ vector of optimization parameters must be real.

$F(p)$ is the scalar valued cost (objective) function.

$G(p)$ is the vector valued equality constraint function.

$H(p)$ is the vector valued inequality constraint function.

h_l and h_u are the lower and upper limits for the inequality function.

p_l and p_u are the lower and upper limits for the optimization parameters.

In general, $F(\cdot)$, $G(\cdot)$, and $H(\cdot)$ are any nonlinear functions that can be specified and computed using `Xmath` or `SystemBuild`. The constraint equations and parameter bounds need not be specified. This flexibility results in a wide variety of problem solving capabilities, which are illustrated in the examples in this chapter.

Optimize()

```
[P, Jh, L, H, IC, Ph]=optimize(P0, {Pmin, Pmax, ICmin, ICmax, L0,
H0, IC0, rho, majit, minit, delta, tol})
```

Conditions and parameters for operation of the function that invokes the `optimize()` function are set up in two places:

- The user furnishes the values of the cost and constraints through a MathScript function (MSF) named `COST()`, which is executed by `optimize()` whenever it needs the appropriate functions for a set of

candidate parameters evaluated. `optimize()` computations include the cost function $F(p)$, an optional equality constraint function $G(p)$, and an optional inequality constraint function $H(p)$.

- All other parameters are specified as parameters of the `optimize()` function call, including:
 - Initial values and bounds for the optimization parameters
 - Bounds for the inequality constraint equations
 - A penalty parameter controlling the search for a feasible solution
 - Maximum numbers of major and minor iterations
 - Numerical gradient perturbation parameters
 - A tolerance parameter on optimality and feasibility

For a detailed description of each `optimize()` input, output, and keyword, refer to the *Xmath Help*.

Running Optimize()

The Optimization Module provides an interface between MathScript and graphical modeling in SystemBuild. As shown in Figure 2-1 and in the following procedure, running `optimize()` in Xmath is a straightforward process.

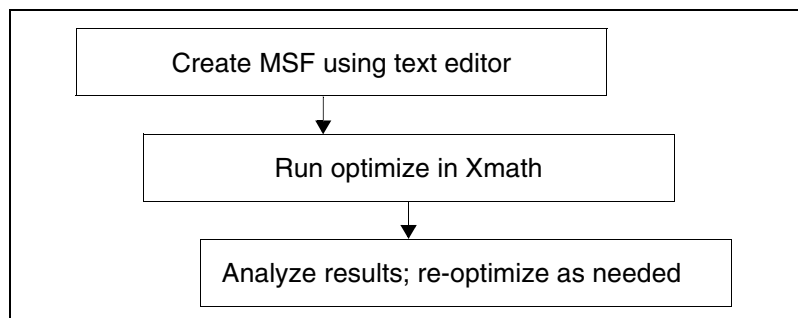


Figure 2-1. Running Optimize()

1. Create a MathScript function (MSF) named `cost.msf()` using any text editor. Your function computes the cost that is to be minimized and also calculates the constraint functions.
2. Specify input variables and execute the `optimize()` function. `optimize()` calls the `cost.msf()` as many times as its operations require.

3. As the algorithm executes, you can monitor its progress by displaying intermediate values and/or creating plots in the `cost` function. Upon completion, all results are returned to Xmath's command level for quick access to the Xmath system analysis, post processing, and simulation capabilities. To save any of the variables computed by the `cost` function, use the Xmath `SAVE` command or select **File»Save All** from the **Xmath Commands** menu.

The objective function and any constraint functions are specified in `cost.msf()`. This means that any dynamic or static system performance measure can be defined quickly and easily. Waveform math, matrix algebra, frequency response analysis, and time simulations are some of the operations that can be used in defining problems. Example 2-1 is a template for `cost.msf()`.

Example 2-1 Template for `cost.msf()`

```
# cost function syntax:
function [out]=cost(p,it)
#{
    p is the list of parameter vectors cost.msf uses to evaluate
    the cost and constraint function values.
    it (iteration) informs the cost function that optimize() has
    completed a major or minor iteration:
        it<0 a minor iteration has completed a gradient evaluation
            occurs once per minor cycle)
        it=0 a gradient or search evaluation
        it>0 major iteration completed
}#
cost_at_p = ( ) # insert cost to be minimized (scalar):
equality_at_p = ( ) #{ compute value(s) of equality function(s)
    and place them in the variable equal (a column vector). Skip this step
    if equality constraints are unnecessary.
}#
inequality_at_p = ( ) #{compute value(s) of inequality
    function(s) and place them in the variable inequal (a column vector).
    If inequality constraints are specified, the number of values must
    match the number of upper and lower inequality constraints specified
    in the optimize call. Skip this step if inequality constraints are
    unnecessary.
}#
#{
    put results in out, omitting any variables that were not
    computed above.
}#
```

```

out=[cost_at_p;equality_at_p;inequality_at_p];
#{    display or store any intermediate results here using the
      variable it to determine points of major and minor
      iterations.
}#
endfunction

```

The `cost()` function accepts the column vector of optimization parameters p and the scalar variable it , and returns the argument out . In the body of `cost()`, the user specifies how p will be used to compute the `cost()` function and (optionally) any equality and/or inequality constraints. Any command or function that can be issued at the Xmath prompt can be used to compute these performance measures. `optimize()` calculates the number of constraints by finding the size of out (the variable returned by the cost function) and determines the number of inequality constraints by checking $IC0$. If m equality constraints and n inequality constraints are specified, the out vector returns $(m + n + 1)$ elements ordered as shown in Table 2-1.

Table 2-1. Out Vector

| Position | Description | Optional (y/n) |
|-----------------------|------------------------|----------------|
| 1 | Objective (COST) | n |
| 2: $m + 1$ | Equality Constraints | y |
| $m + 2$: $m + n + 1$ | Inequality Constraints | y |

The `iter` input parameter informs `cost()` of the completion of a major or minor iteration. `iter` signals any one of three possible conditions in the optimization process after that event has occurred as shown in Table 2-2.

Table 2-2. Iter Possibilities

| Condition | Situation |
|--------------------------|-------------------------------|
| <code>iter < 0</code> | The gradient evaluation |
| <code>iter = 0</code> | Gradient or search evaluation |
| <code>iter > 0</code> | Major iteration completed |

The `iter` input parameter can be used to customize the run-time information displayed on the screen (or in the log file, if the optimization job is run in batch mode). Data can be analyzed, displayed, plotted, or saved

to the disk at minor and/or major iterations. The `optimize()` process is illustrated in the examples given in the *Application Examples* section. Both unconstrained and constrained nonlinear optimization problems are set up, implemented, and solved. The *Optimize Algorithm* section details how the `optimize()` algorithm works and the *Practical Considerations* section discusses some practical approaches that aid in using `optimize()` effectively and efficiently. Detailed demo files supporting the examples in the *Application Examples* section are included with the software so you can follow along with the text and quickly come up to speed with optimization.

Practical Considerations

The `optimize()` algorithm has been designed to handle a broad range of engineering problems with as few restrictions on problem definition as possible. The following subsections discuss several points to consider when using `optimize()`. In most situations, following these guidelines will mean faster and more efficient convergence.

Finding a Feasible Solution

A feasible candidate solution should at least meet all the constraint criteria, even if it is not optimal or near optimal. If `optimize()` is having difficulties converging to a feasible minimum for a constrained optimization problem, try altering the original problem to help `optimize()` solve it more efficiently. One way to do this is to modify `cost()` so that it keeps the cost constant whenever the error in the constraint equation—that is, the difference between the current candidate parameter and the feasible range—is large. This means that the true cost will be computed only when `optimize()` has found a feasible (or near feasible) solution. Forcing `optimize()` to concentrate on finding a feasible region before searching for an optimal solution will get the best results from the algorithm.

Specifying the Penalty Parameter

The penalty parameter, ρ , is used to control the distance that the optimization search is allowed to travel outside the feasible range to seek an optimal solution. A value of this parameter greater than 1 forces the search to stay close to the known feasible range; a smaller value of ρ allows a broader range of searching. The default value, 1, allows an intermediate range of search.

Another way of understanding the role of the penalty parameter is as a weighting factor. The higher the value of ρ , the more emphasis will be put on bringing (or keeping) the parameters inside the feasible region. Be cautious when using large values of ρ , because a large ρ might make the problem ill-conditioned. NI suggests not exceeding $\rho = 1,000$. In most situations, the default value of ρ will result in good performance of the nonlinear optimization routine. In other cases, it may be necessary to specify a good initial guess for the penalty parameter.

For smooth problems, several guidelines may help you find a good initial guess. If a good solution to a similar problem is known, the value of the penalty parameter from the successful optimization will provide a good first guess for the current problem. If you have absolutely no idea about how to find an initial penalty parameter, try $\rho = 1$. In most cases, difficulties associated with a poor initial choice for the penalty parameter can be avoided by specifying reasonable bounds on the parameters.

For non-smooth problems, the difficulties associated with selecting a good initial penalty parameter are not as severe. In this case, a good strategy is to choose a large value for the initial parameter. This will help ensure that the subproblem will have the desired local minimum. If the penalty parameter from a successful optimization of a similar problem is available, start the optimization with a slightly smaller value of the successful penalty parameter. If no Help is available, try 10. Try to place reasonable bounds on the variables to reduce the problem's sensitivity to the penalty parameter.

In general, there is no systematic way of choosing the best value for the penalty parameter. Some problems can be extremely sensitive to this parameter, thus making subproblems poorly conditioned whenever the penalty parameter is too small or too large. It is important to remember what effect this parameter can have, and that it may only be possible to find a good value through trial and error. For more details, refer to the *Penalty Functions* section and the *Penalty Parameters* section of *Practical Optimization*.¹

Evaluating Results

After the algorithm has completed the optimization, be careful to check that the optimal value meets any additional criteria implied by common sense. For example, the final result should be a true minimum; at least, it should be considerably smaller than the initial result.

¹ Gill, P. E., Murray, W., and Wright, M. H., *Practical Optimization*, Academic Press, 1981.

Reducing Constraints to Improve Performance and Efficiency

Constraints add a great deal of complexity to the general optimization problem. It is always advantageous for the user to simplify, reduce, or completely eliminate constraints. With general formulation of the `optimize()` function, several types of constraints are possible. Equality constraints and inequality constraints are the most general classification. These constraints can be linear or nonlinear in the optimization parameters. Optimization with inequality constraints is the most difficult problem to solve, and nonlinear constraints pose greater difficulties than linear constraints. With careful specification, constraints can be simplified and the optimization will be performed more efficiently and with fewer problems.

Always check the constraint equation and first eliminate any redundant or unnecessary constraints. Next, try to replace or simplify any constraints, or replace the constraints with bounds on the parameters. In some cases, a transformation of variables will be helpful. The following problem,

$$\begin{aligned} \min & (x_1 + 2)^2 + (x_2 + 1)^2 \\ & x_1 \geq 0 \end{aligned}$$

can be converted into an unconstrained minimization problem by letting:

$$x_1 = z_1^2 \text{ and } z_2$$

The problem then becomes to minimize:

$$(z_1^2 + 2)^2 + (z_2 + 1)^2$$

If this is not possible, then try to turn nonlinear constraints into linear constraints. Finally, try to combine and simplify constraint equations to keep the same limitation on the problem using a smaller number of simpler equations. The effort of simplifying the constraints beforehand is a wise investment toward making the optimization process much more efficient and robust, but often it is difficult to decide when or how to make the modifications.

Transformations can introduce very undesirable properties in the optimization problem—for example, discontinuities, periodicity in the optimization parameters, singularities, poor scaling, and a greater number of minima. For this reason, simplify the constraints only after full

consideration of the pitfalls which may result. When in doubt, do not simplify.

Eliminating redundant parameters is as important as eliminating redundant constraints. The following transfer function is an example:

$$h(s) = \frac{as + b}{cs^2 + ds + e}$$

This expression has five parameters: a , b , c , d , and e . However, the same transfer function can be expressed as

$$h(s) = \frac{k_1s + k_2}{s^2 + k_3s + k_4}$$

where

$$k_1 = \frac{a}{c}, \quad k_2 = \frac{b}{c}, \quad k_3 = \frac{d}{c}, \quad \text{and} \quad k_4 = \frac{e}{c}$$

This reduces the number of parameters from five to four.

Avoiding Discontinuities

Like most optimization algorithms, convergence of the `optimize()` function can be guaranteed only when the objective function and any constraint functions are sufficiently smooth. Discontinuities should be avoided whenever possible. Discontinuities can arise from seemingly insignificant sources in problem formulation and definition. Be careful to avoid these common pitfalls:

- **Table Lookup**—Linear interpolation table lookup functions are a common, but avoidable, source of discontinuities in the formulation of an optimization problem. Table lookups are continuous in the function along all interior points, but the first derivative of the function will be piecewise constant with linear interpolation. Because table lookups are often used to approximate continuous data, there is little or no loss of accuracy involved with providing a smooth approximation—for example, by using the `spline()` function in Xmath to smooth a linear interpolation.
- **Piecewise Constant Functions**—Piecewise constant functions such as non-interpolated table lookup and sampled data time histories can pose significant problems in optimization. If the resolution of a computation is coarse, numerical gradients will not be accurate.

This results in slow convergence or poor algorithm performance. Two courses of corrective action may improve the performance of the algorithm:

- Interpolation (preferably with a continuously differentiable algorithm)
- Finer resolution (more samples) in the function
- **Function Switching**—Avoid situations where the objective function or any constraint functions contain computations that change in structure (switching from one formula to another) as a function of the parameter values. When such a function must be included, make sure that the function outputs and first derivatives match at all switching conditions.
- **Iterative Algorithms**—If you use an iterative algorithm to compute the objective and/or constraint functions and you have specified tolerances considerably larger than the machine constants, nontrivial discontinuities can occur. A common example occurs when a variable step algorithm is used to evaluate an integral; successive iterations of the optimization algorithm may result in varying local accuracy along the output trajectory. The function and/or its gradients will not be smooth functions of the parameters. In certain cases, the error can be substantial. This error can be avoided by using fixed step methods or a very small local error tolerance for the variable step method. Another option is to increase the size of the perturbation used in gradient step-size whenever this retains the integrity of the gradient computation. In most cases, it will be difficult to determine the proper balance between integration algorithm accuracy and the gradient perturbation, so try a fixed step integration algorithm.

NI advises that you define step size for numerical derivatives so changes due to perturbing the parameters are greater than the errors in the integration, or else you will end up with derivative of integration noise with respect to your parameters. For example, if the numerical integration is good to ± 0.1 and you expected results to change by $.001$ when you perturb the parameter, you cannot tell the difference. You either can make the integration error specification tighter (a precision of $.0001$) or boost the perturbation so the expected change is one-tenth ($.1$).

When using SystemBuild, avoid using integration blocks or saturation blocks as limiters. If saturation is reached, `optimize()` does not get anything useful. Rather, make the signals sim outputs so that the optimization algorithm can use constraints to do the limiting.

These and other important considerations are discussed in greater detail in the [Specifying the Penalty Parameter](#) section.

Controlling the Numbers of Major and Minor Iterations

In an attempt to achieve convergence, the `optimize()` function divides the general nonlinear programming problem into a series of linearly constrained nonlinear programming steps to approximate the actual nonlinear problem; these are the major iterations of the function. The major iteration linearizes the constraints and tries to find a feasible solution for the linearized problem. It then passes it to the minor iteration(s).

The minor iterations use quadratic methods to find a solution within the linear constraints. When the minor iterations are complete or a solution is found, `optimize()` returns to the major iteration. These ideas are illustrated in Figure 2-2.

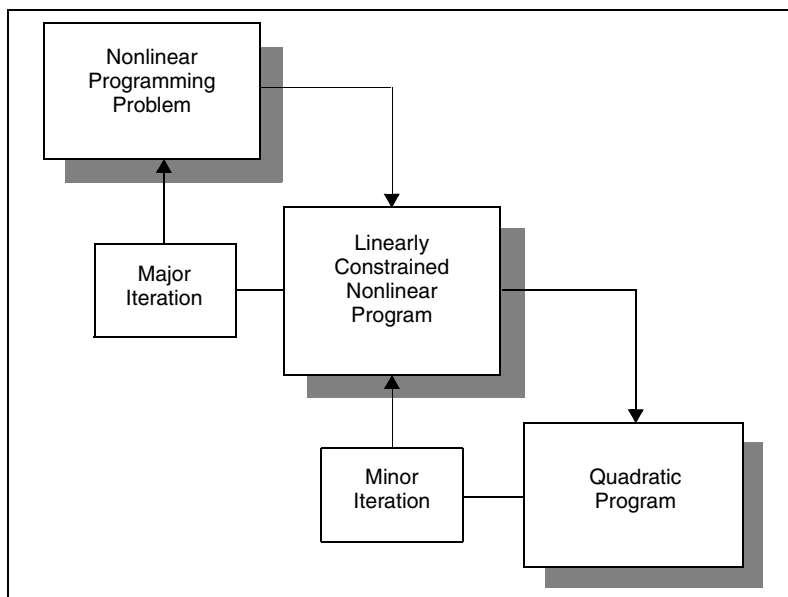


Figure 2-2. Major and Minor Iterations

Many optimization problems are sensitive to the number of either major or minor iterations, but it is difficult to give general rules for assigning these variables, because they are intensely problem-dependent. Ten is the default setting for the number of both major and minor iterations. This is intentionally generous; often performance can be enhanced with no loss of accuracy by using smaller numbers for either or both parameters. It is easy to study this effect, because the calls to the `COST MSF` contain the indication that this call is for a major or a minor iteration, and one can see if, for example, more minor iterations give an evidently better local solution, or if they are just a waste of time.

For some problems, the constraints influence the iterations greatly. This is true if the solution from the minor iterations falls beyond the true nonlinear constraints. If the solution to the constraints is twice as bad as the solution from the previous major iteration, ρ will be increased. If it is safely within the tolerances, ρ will be decreased.

The higher ρ is, the more important the constraints are. If ρ is small, it is more tolerant. If your problem is poorly behaved when the constraints are violated, you want to push the solution towards the constraints by increasing ρ and tightening the constraints.

The constraint used in the minor iteration quadratic program is only a linear approximation. This means some inaccuracy may result because of nonlinearities in the actual constraints. If your constraints are highly nonlinear, you may want to reduce the number of minor iterations so that the linear approximation will be updated more often.

Optimize Algorithm

When given the general nonlinear programming problem, `optimize()` converts Equation 2-1 into Equation 2-2.

$$\begin{aligned} \min \quad & F(p) \\ \hat{p} \\ \hat{G}(\hat{p}) = 0 \\ \hat{p}_l \hat{\phi} \hat{\phi}_u \end{aligned} \quad (2-2)$$

where G is a combination of G and H :

$$\hat{G} = \begin{pmatrix} H(p) - s \\ G(p) \end{pmatrix} \quad \text{Let} \quad \begin{aligned} \hat{p} &= \begin{pmatrix} p \\ s \end{pmatrix} \\ \hat{p}_u &= \begin{pmatrix} p_u \\ h_u \end{pmatrix} \\ \hat{p}_l &= \begin{pmatrix} p_l \\ h_l \end{pmatrix} \end{aligned}$$

`optimize()` solves the problem by first constructing a linearly-constrained optimization problem with an Augmented Lagrangian objective function shown in Equation 2-3,

$$\begin{aligned} \min_{\hat{p}} \quad & F(\hat{p}) - \hat{y}_k \hat{G}(\hat{p}) + \frac{\rho}{2} \hat{G}(\hat{p})^T \hat{G}(\hat{p}) \equiv \Phi(\hat{p}) \\ J_k(\hat{p} - \hat{p}_k) = -\hat{G}(\hat{p}_k) \quad & \hat{p}_l \leq \hat{p}_l \leq \hat{p}_u \end{aligned} \quad (2-3)$$

where J is a numerical approximation to the Jacobian:

$$J_k = \left. \frac{\partial \hat{G}(\hat{p})}{\partial \hat{p}} \right|_{\hat{p} = \hat{p}_k} \quad (2-4)$$

and \hat{y}_k is the vector of Lagrange multipliers at step k (the first step, $\hat{y}_0 = 0$ by default). This problem is solved with an iterative algorithm for the variables \hat{p}_{k+1} and \hat{y}_{k+1} . For the new problem, the first step involves checking to see if \hat{p}_k satisfies the linear constraints.

$$\begin{aligned} J(\hat{p} - \hat{p}_k) = -\hat{G}(\hat{p}_k) \\ \hat{p}_l \leq \hat{p}_l \leq \hat{p}_u \end{aligned}$$

If \hat{p}_k is not a feasible solution, Equation 2-4 is solved to generate a valid initial condition for subsequent iterations.

Next, sequential quadratic programming (SQP) is implemented to solve Equation 2-3. You calculate the gradient vector \hat{g}_k and update the Hessian matrix (Equation 2-5) of the Augmented Lagrangian objective of Equation 2-3 to obtain the second-order approximation to the objective function $\Phi(\hat{p})$ in the quadratic programming (QP) problem in Equation 2-6.

The Hessian matrix is updated as follows (Broyden-Fletcher-Goldfarb-Shanno):

where

$$\begin{aligned} \Delta p &= \hat{p}_{k+1} - \hat{p}_k \\ \Delta g &= \hat{g}_{k+1} - \hat{g}_k \end{aligned}$$

then

$$H = H - \frac{1}{\Delta p^T H \Delta p} H \Delta p \Delta p^T H + \frac{1}{\Delta g^T \Delta p} \Delta g \Delta g^T \quad (2-5)$$

$$\min_{\hat{p}} \frac{1}{2}(\hat{p} - \hat{p}_k)^T H_k(\hat{p} - \hat{p}_k) + \hat{g}_k(\hat{p} - \hat{p}_k) \quad (2-6)$$

$$J_k(\hat{p} - \hat{p}_k) = -\hat{G}(\hat{p}_k)$$

$$\hat{p}_l \leq \hat{p}_u$$

where H_k is a Hessian for $\Phi(\hat{p}_k)$ and (\hat{g}_k) is a gradient for the Augmented Lagrangian objective function $\Phi(\hat{p}_k)$.

The `optimize()` function uses the interior trust region method¹ to reach an approximate solution for QP (Equation 2-6). A line search technique provides a step size for Equation 2-3 and is used to generate the minimal solution \hat{p} .

If the optimal QP solution \hat{p} is both feasible and optimal for Equation 2-3, `optimize()` returns to the beginning. Otherwise, it updates the Hessian matrix (Equation 2-5) for the subproblem Equation 2-6 and solves the QP problem. This repeats until the optimal local solution is found or until the maximum number of minor iterations is computed.

The final solution \hat{p} and multiplier \hat{y} that result from the QP process are \hat{p}_{k+1} and \hat{y}_{k+1} . These values are returned to solve for the next subproblem (Equation 2-3). When Equation 2-3 converges to a local minimum \hat{p} for Equation 2-2, the algorithm is complete.

Application Examples

This section illustrates a variety of applications for the `optimize()` function. The first example shows how to `optimize()` the volume of a box, the second solves a trajectory optimization problem, and the last optimizes the response of a third order nonlinear system. This group of applications demonstrates the approaches and procedures you will need to begin solving your own optimization problems. Each application has four phases:

- **Problem Definition**—Define the system and performance objectives.
- **Formulation**—Specify the system cost and constraint equations using Xmath and SystemBuild commands and functions in the `cost.msf` file.

¹ Y. Y. Ye, Ph.D. Dissertation, Dept. of Engineering-Economic Systems, Stanford University, 1987.

- **Optimization**—Use the `optimize()` function to compute and solve optimization problems.
- **Analysis**—Examine the preliminary solution. If the algorithm does not converge in the specified number of iterations, the `optimize()` function can be restarted. The algorithm can be run with different parameters or from another initial condition to test the assumptions about a local minimum.

To get the most out of these application examples, you should know how to create a MathScript function (MSF). For instructions on creating an MSF, refer to the *MATRIXx Help* or *Xmath Basics*.

Box Design

The objective of this example is to find the height, width, and length that give a box a total volume of 100 m^3 and minimize the amount (surface area) of cardboard required. The situation is illustrated in Figure 2-3. The only restrictions are that each box side must be greater than 1 m and less than 10 m in length.

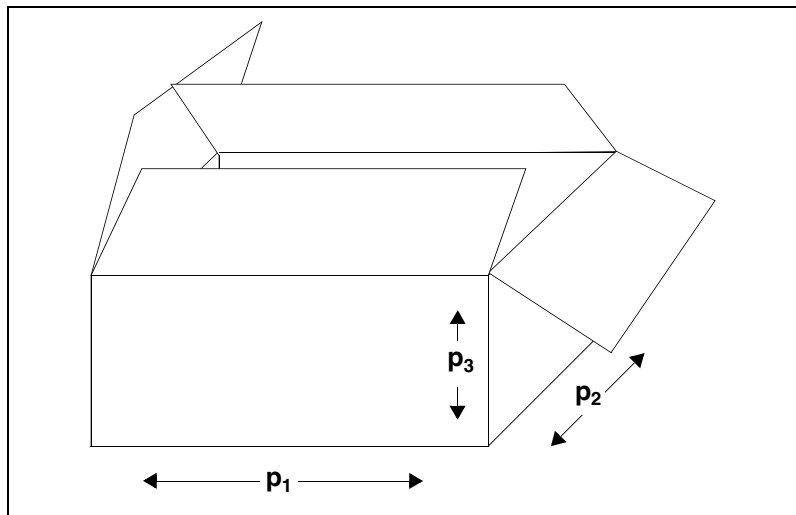


Figure 2-3. Box Optimization

Formulation

As shown in Figure 2-3, the dimensions of the box are p_1 , p_2 , and p_3 . The top and bottom of the box have double flaps so that the surface area of the top and bottom is $4p_1p_2$. The total surface area of cardboard required is as follows:

$$J(p) = 4p_1p_2 + 2p_2p_3 + 2p_1p_3$$

This is the quantity to be minimized and it will be returned as the first value from the `cost.msf` file. The second argument to be returned is the difference between the actual and desired volume: $G(p) = p_1p_2p_3 - 100$.

This argument defines the equality constraint that requires the volume of the box to be exactly 100 m^3 . These equations are coded in a MathScript file named `cost.msf`.

```
#Cost function for the box optimization problem.
function out=cost(p,iter)
out=[
4*p(1)*p(2)+2*p(2)*p(3)+2*p(1)*p(3) # surface
p(1)*p(2)*p(3)-100];# vol -100
endFunction
```

In the `cost.msf` file, $J(p) = \text{out}(1)$ and $G(p) = \text{out}(2)$. In a directory of your choice, create `cost.msf` as shown in the [Optimization](#) section.

Now you are ready to run the optimization in Xmath.

Optimization

To run `optimize()` on this example, complete the following steps:

1. Enter Xmath and define the parameter bounds for each dimension of the box:

```
pmin=ones(3,1);pmax=10*pmin;
```

2. Make `p0` equal to the average of the upper and lower bounds, thus:

```
p0=(pmax+pmin)/2;
```

3. Execute the `optimize()` function:

```
[p,jh,l]= opti(p0,{pmin=pmin,pmax=pmax,
rho=1,majit=5,init=10,delta=1e-6,tol=1e-6})
```

The `optimize()` function computes successive estimates of the parameter dimensions and the error in the volume of the box. While running, `optimize()` displays the current major iteration and current minor iteration. The current value of the `cost()` function (J) is also displayed. These are output to the screen with each iteration until the maximum number of major iterations is reached or `optimize()` is completed.

In Example 2-2, `optimize()` returns the final parameter estimate `p` and the history of cost values (surface areas) `jh` after five major iterations are computed. `l` is the Lagrange multiplier.

Example 2-2 Running `optimize()` on the Box Dimensions

```
Beginning minor iteration 1
Beginning minor iteration 2
Beginning minor iteration 3
Updated parameters:    3.51798    3.51798    7.26982
Beginning major iteration 2, J=151.805
Beginning minor iteration 1
Beginning minor iteration 2
Beginning minor iteration 3
Beginning minor iteration 4
Beginning minor iteration 5
Beginning minor iteration 6
Updated parameters:    3.75321    3.75319    7.1079
Beginning major iteration 3, J=163.055
Beginning minor iteration 1
Beginning minor iteration 2
Updated parameters:    3.74731    3.74728    7.12137
Beginning major iteration 4, J=162.912
Beginning minor iteration 1
Updated parameters:    3.74724    3.74721    7.12165
Beginning major iteration 5, J=162.912
Beginning minor iteration 1
Updated parameters:    3.74723    3.74721    7.12166
Completed in 5 iterations

p (a column vector) =
    3.74723
    3.74721
    7.12166

jh (a row vector) =    242    151.805    163.055...

l (a scalar) =    1.09409
```

Trajectory Optimization (Zermelo's Problem)

Problem Definition

In this example, a ship must travel through water where the current varies with (x,y) location. Given any initial point (x_o, y_o) , you want to find the trajectory of the ship that will place it at the origin $(0,0)$ in minimum time. In this example, assume that the ship travels at a constant speed V , that the current flows in the x direction, as shown in Figure 2-4, and that the speed of the current varies linearly with y position.

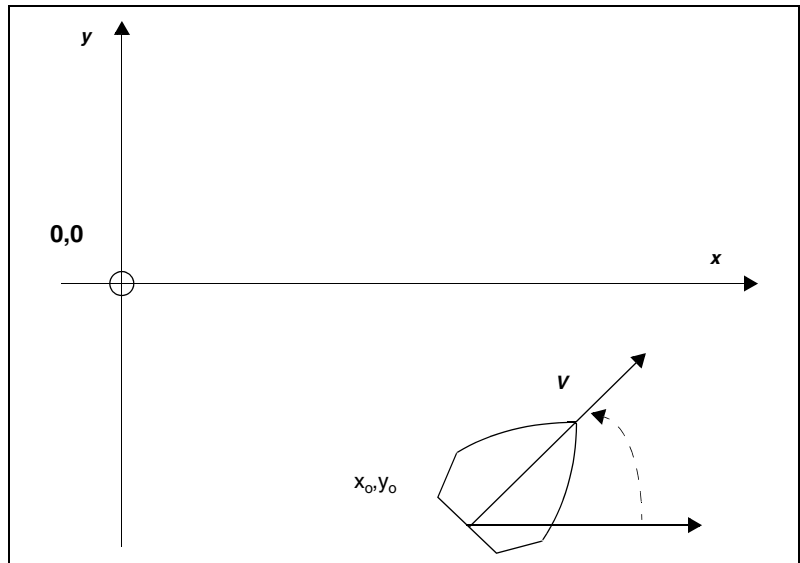


Figure 2-4. Ship Trajectory Optimization Problem

Formulation

The problem solving strategy is to constrain the final (x,y) position of the ship to $(0,0)$ and minimize the time it takes the ship to reach this origin. The `cost()` function parameters are the total time and the value of the steering angle at eleven evenly-spaced time points. The `cost()` function takes the inputs and simulates the motion of the ship over the time interval. It computes the final position and returns the total time employed and the final distance from the origin.

The first step is to write the equations of motion for the ship in the coordinate system of Figure 2-4.

$$\begin{aligned} \text{For } V = 1, \quad \dot{x} &= V \cos \theta + ky \\ \dot{y} &= V \sin \theta \\ k &= -1 \end{aligned} \qquad \begin{aligned} \dot{y} &= \sin \theta \\ \dot{x} &= \cos \theta - y \end{aligned}$$

Recall that the current acts only in the x direction and is a linear function of the ship's y position.

The next step is to implement these equations in a SystemBuild block diagram where you can simulate the (x,y) ship position as the input angle $\theta(t_k)$ varies.

The block diagram for the equations of motion is shown in Figure 2-5. It takes θ , calculates the right side of the differential equation, and integrates the velocities to give x and y positions.

Before proceeding with the example, start SystemBuild and load the data file for this model from the demo directory appropriate to your operating system.

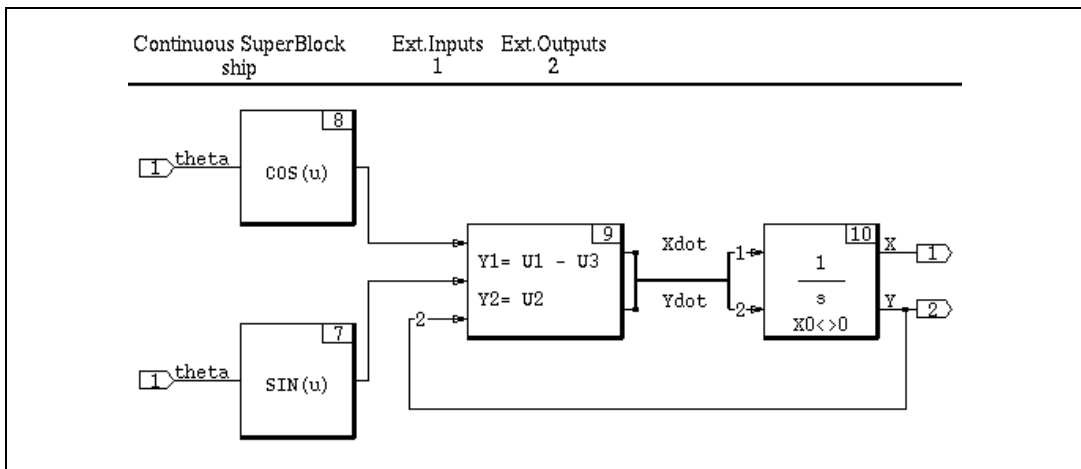


Figure 2-5. Ship Block Diagram

build

load "\$XMATH/demos/ship/ship.dat"

Initial conditions for this example are set in the SystemBuild integrator block at:

$$(x_o, y_o) = (3.5, -1.8)$$

The `cost()` function for this example is shown in Example 2-3.

Example 2-3 `cost()` function

```
{For clarity we use unnecessary calculations and temporary values in the
code below. This slows down the optimization procedure.}#
function [out]=cost(p,iter)
theta=p(1:11);
tf=p(12);# final time
time=[0:.1:1]'*tf;# time vector
xy=sim("ship",time,theta,{simclock=0,simmessage=0});
temp=makematrix(xy);
xf=temp(1,11);
yf=temp(2,11);      # final x and y positions
r=xf**2+yf**2;# distance to the origin
out=[tf;r];# out=[objectv;eq_constr]
x=temp(1,:);y=temp(2,:);
if iter>0# plot at major iteration
plot(x,y,{xmin=-1,xmax=5,ymin=-2,ymax=3,keep})?
endif
endFunction
```



Note It is more efficient to define the single constraints than to have two constraints.

$$G_1(p) = x(t_f)^2 + y(t_f)^2 = 0$$

You can copy the previous cost function to your working directory with the following Xmath command:

```
fprintf("cost.msf", "%s", read("$XMATH/demos/ship/cost.ms
f"), {reset});
```

This MSF takes an input steering angle (as a function of time) and computes the trajectory and final (x,y) position of the ship through the `sim()` function. The only tricky part of the `cost()` function is the specification of the course heading as a function of time.

The set of steering angles (θ) is specified at 11 evenly-spaced time points (`theta=p(1:11)`).

To determine the total time (the goal is to minimize the time to reach the origin), recompute the simulation time vector from the normalized time column vector `time` before each simulation: `time=[0:.1:1]'*tf`.

This vector is the scaled time vector for the simulation. This gives the algorithm complete control over the total time needed to reach the origin. `tf(p(12))` is returned as the first element of the `out` vector. This parameter will be minimized by the `optimize()` function. The sum squared of the terminal position (x,y coordinates) is returned as the equality constraint because you want the terminal position to be at the origin (0,0).



Note It is more efficient to define the single constraints $G_1(p) = x(t_f)^2 + y(t_f)^2 = 0$ than to have two constraints.

The final line of the `cost()` function checks to see if a major iteration has been completed. Upon completion of an iteration, the `cost()` function plots the ship's x,y trajectory.

1. After the `cost.msf` file has been created, create the parameter bounds $-2\pi \leq \theta(t) \leq \pi$. As arbitrarily large numbers, they are not expected to have much effect on the convergence process.

```
p_lower = [-2*pi*ones(11,1);0];
p_upper = [2*pi*ones(11,1);20];
```

It is tempting to constrain the steering angle θ to be between 0 and 2π , or between $-\pi$ and π , but this would prevent the ship from completing more than one circle.

2. For an initial guess at the correct steering trajectory, assume that the ship maintains a constant heading in the general direction of the origin.

```
p0 = ones(11,1)*pi*2/3;
```

3. We guess that the ship will take five seconds to reach the origin:

```
p0 = [p0;5];
```

4. Run the `cost()` function with the initial parameters. The graphics window displays a plot of trajectory from the initial parameter (refer to Figure 2-6).

```
cost(p0,1);
```

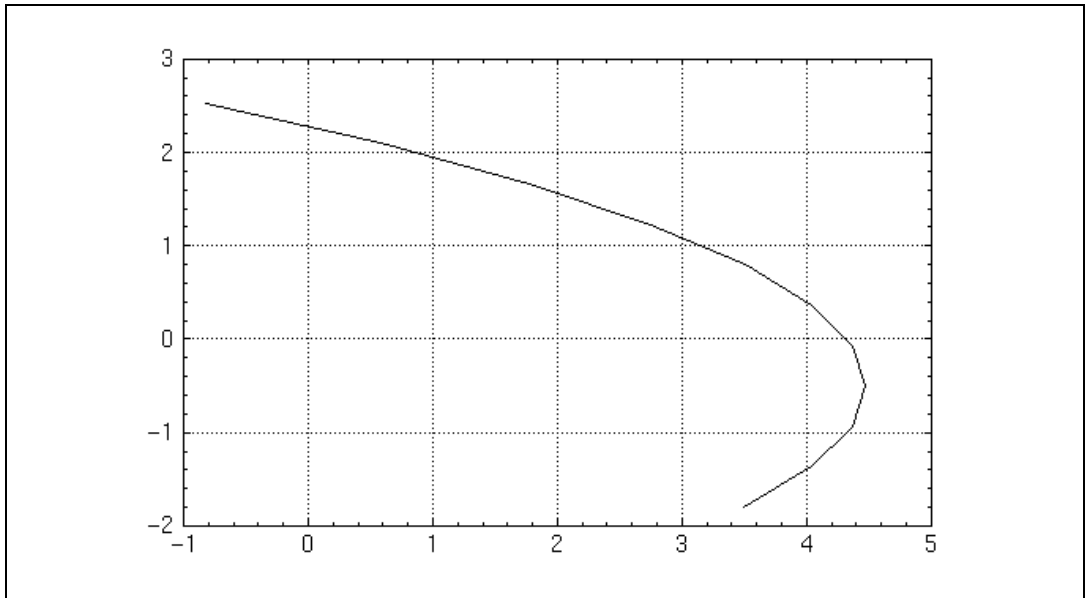


Figure 2-6. Plot of Trajectory

Optimization

For the optimization phase, the penalty parameter `rho` is set to 1. The number of major and minor iterations are set empirically to 10 and 3. The tuning parameters have been reduced to cut the running time of the example.

```
[p, jh, l] = opti(p0, {pmin=plower, pmax=pupper,
rho=1, majit=10, minit=3, delta=1e-4, tol=1e-2})
```

As the optimization proceeds, intermediate cost and constraint values are displayed, and updated parameters are output at each major iteration. A plot of the ship's trajectory is also created at each major iteration. Optimization is completed in six iterations as shown in Example 2-4.

The final plot is shown in Figure 2-7. The initial parameter plot is included in the graph because `cost.msf()` uses the `keep` keyword with `plot()`. If you do not want the plots combined, type `erase` in the **Commands** window command area before running `optimize()`.

Example 2-4 Optimization of Ship Trajectory

Completed in 6 iterations

p (a column vector) =

- 1.54072
- 1.7487
- 2.31885
- 2.39624
- 1.68404
- 2.19942
- 2.91508
- 3.23675
- 3.5521
- 4.06592
- 3.71046
- 5.36374

jh (a row vector) = 5 4.41321 4.77413 ...

l (a scalar) = -16.5989

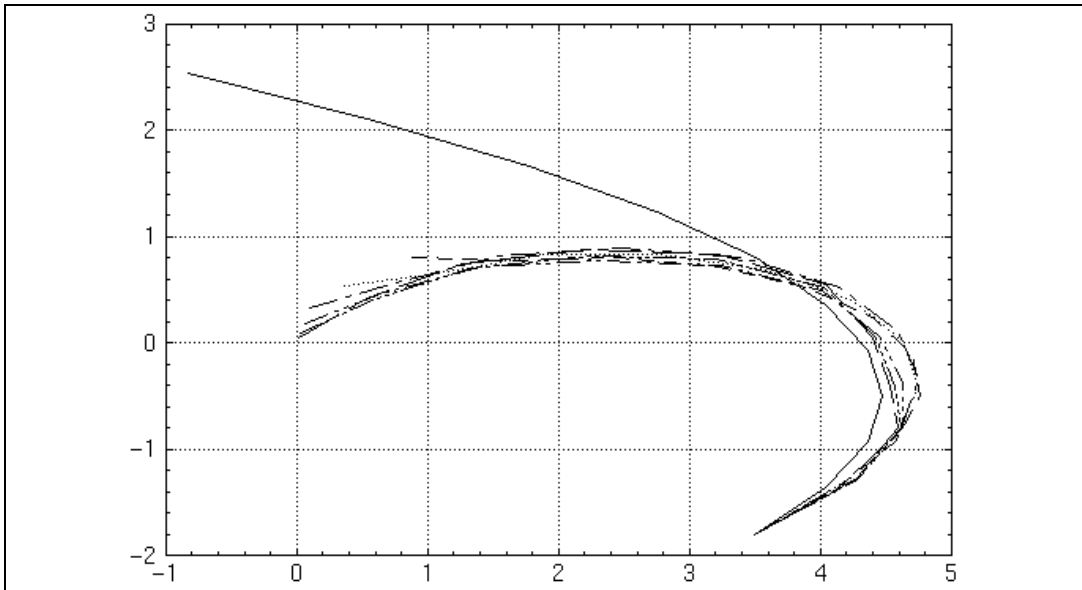


Figure 2-7. Ship Trajectory Plot

Analysis

The final time value is 5.36374. Figure 2-8 shows the initial response plotted with the final time value. To create this plot, type:

```
erase
cost (p0, 1)
cost ([p; 5.36374], 1)
```

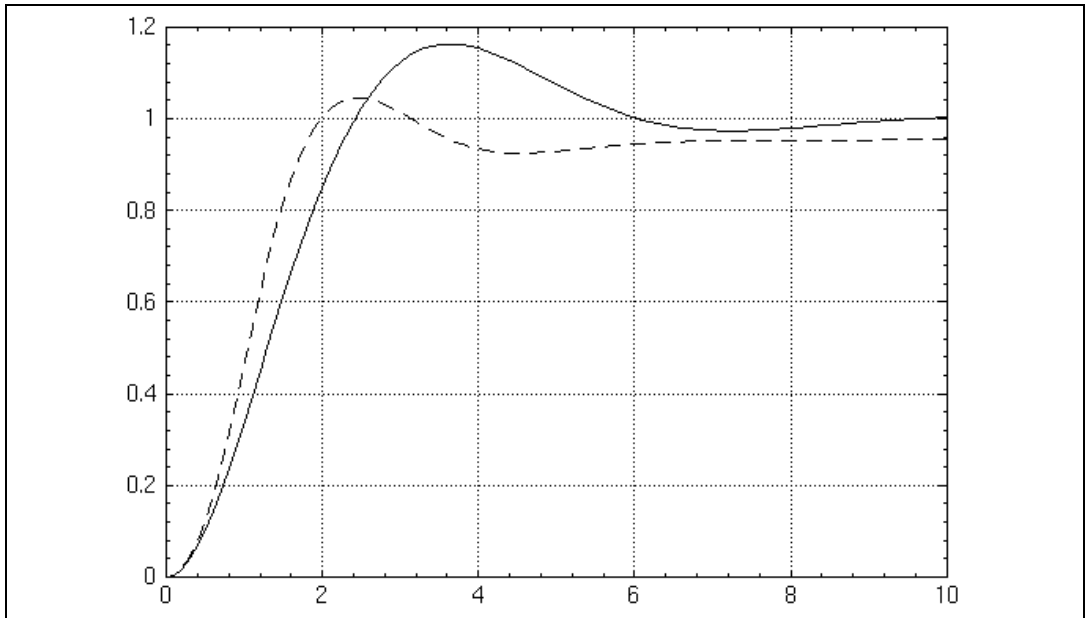


Figure 2-8. Initial versus Optimal Trajectory

This was a concise but very nonlinear trajectory optimization problem. Had you chosen a greater number of minor iterations, you might have obtained a faster or better solution. If the tolerance had been reduced, or allowed to run at default, the problem would have taken considerably longer to converge.

As previously illustrated, one useful feature of `optimize()` is the ability to change the iteration parameters and tolerances to help convergence at any intermediate point in the optimization process, without having to start over again or go through extensive work to compile, redefine, or resubmit the optimization job.

Also, observe that the value of the constraint has been reduced by the last iteration, and the divergence from the origin of the last two iterations is scarcely visible in Figure 2-7.

Feedback Control Design

Problem Definition

In this example, you use `optimize()` to specify feedback control logic for the closed-loop system shown in Figure 2-9.

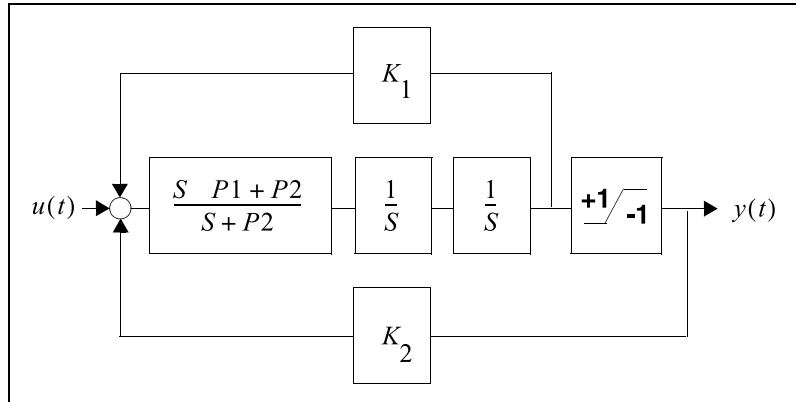


Figure 2-9. Closed-Loop Control System Model

The objective is to find the values of the feedback gain parameters K_1 and K_2 , along with the compensator coefficient parameters $p1$ and $p2$ such that the response at y from a step input at u is as good as the saturation specified in the actuator will allow.

You want the output signal to follow the input step as closely as possible while keeping the maximum percentage overshoot less than 5% (refer to Figure 2-10). Your working strategy is to constrain the overshoot and hope that an acceptable settling time results. If the overshoot criteria are met, but the settling time is unacceptable, the problem can be reformulated.

The first step is to create a SuperBlock diagram based on the model (refer to Figure 2-9). The model system has a single input where the step input is applied, and a single output where the position response is measured.

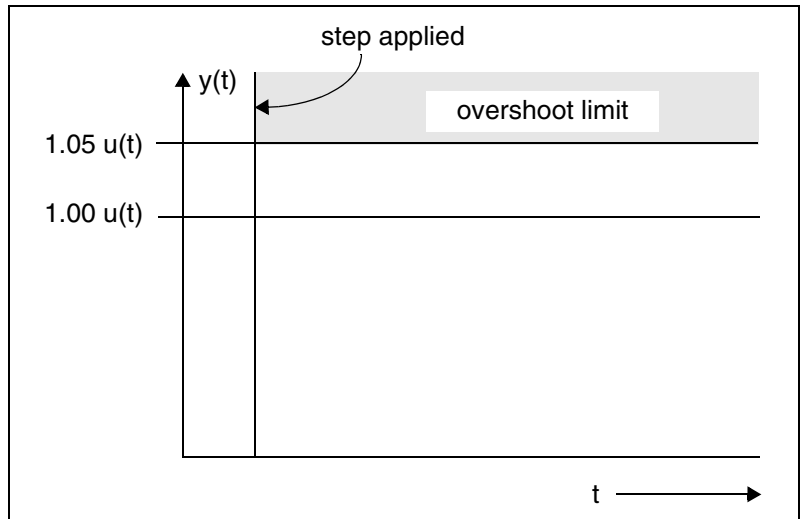


Figure 2-10. Desired Step Response

Table 2-3 shows the parameters and their initial values and how they will be used in the SuperBlock.

Table 2-3. Parameters and Initial Values

| Block | Parameter | Value | Parameter |
|---------------|---------------|-------|------------------------------------|
| compensator | NUM= [p1, p2] | [1,1] | vector of numerator coefficients |
| compensator | DEN= [1, p2] | [1,1] | vector of denominator coefficients |
| rate gain | K_1 | 1 | rate gain |
| position gain | K_2 | 1 | position gain |

You can load the model shown in Figure 2-11 from the demo directory with the following command:

```
load = "$XMATH/demos/system/system.dat"
```

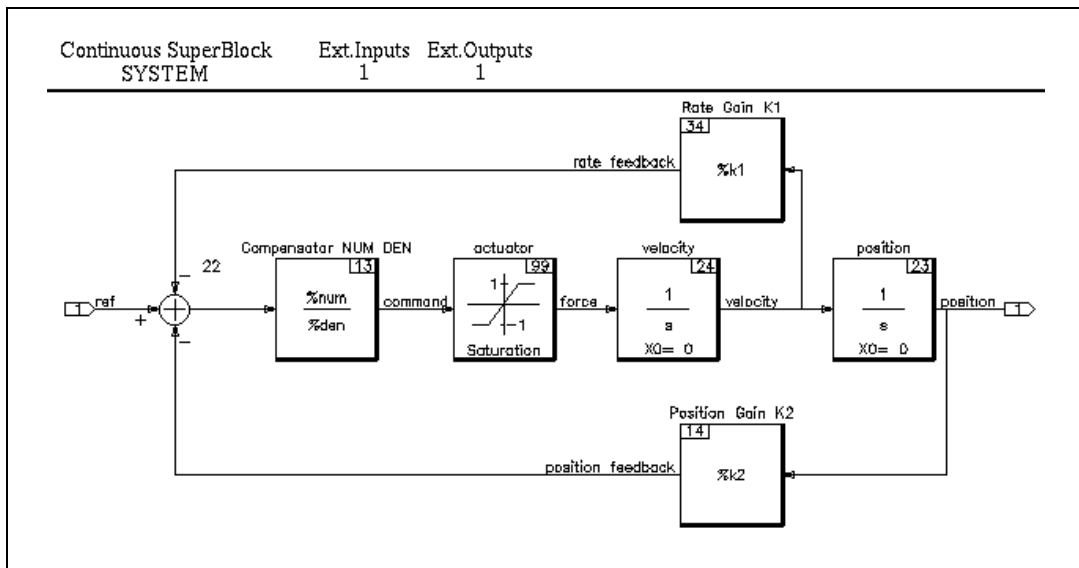


Figure 2-11. Closed-Loop SuperBlock

Alternatively, you can define a continuous SuperBlock called `SYSTEM` and build the model as shown in Figure 2-11. When defining the block named `Compensator NUM DEN`, use the values of the variables `NUM` and `DEN`. The form for the block compensator should duplicate the values shown in Figure 2-12 (from the Parameters tab of the SystemBuild Editor). When defining the gain blocks named `rate gain` and `position gain`, use the appropriate values of K_1 and K_2 . Finally, parameterize `NUM`, `DEN`, K_1 , and K_2 in the appropriate blocks.

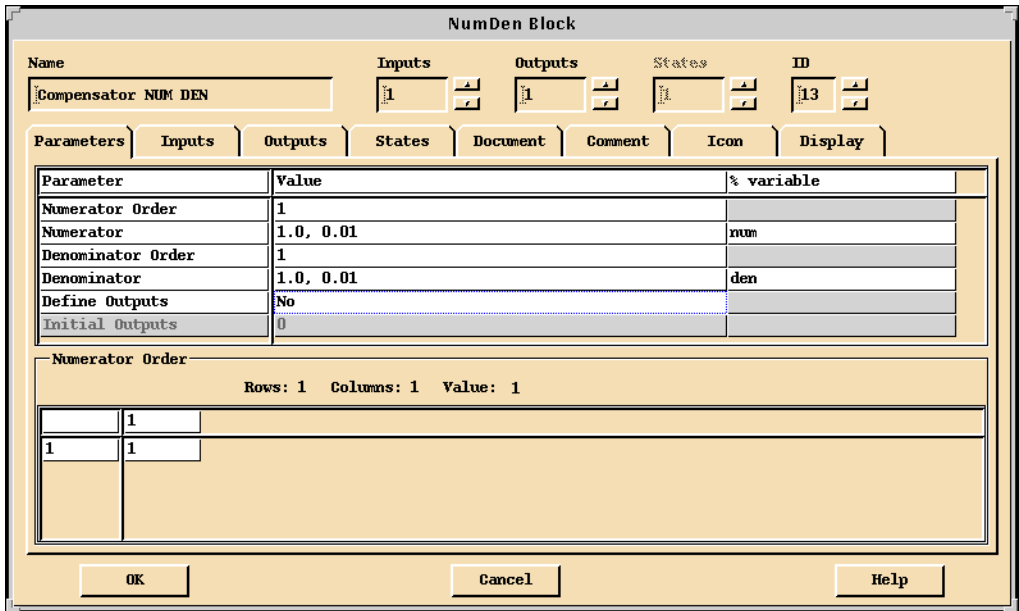


Figure 2-12. Designating Parameters

The SuperBlock `SYSTEM` contains a saturation block. You know in advance that saturation will not be reached, so this is acceptable for this example. However, saturation blocks, or any blocks that might introduce nonlinearities, should be avoided. If saturation is reached, optimization stops. Another way to handle this problem is to remove the saturation block and limit the signal with `icmin=-1, icmax=1` as described in the [Avoiding Discontinuities](#) section.

After the `SYSTEM` model is built, create `cost.msf`, which uses the model to compute the performance parameters. You can copy the cost function to your working directory with the following Xmath command:

```
fprintf("cost.msf", "%s", read("$XMATH/demos/system/cost.msf"), {reset});
```

Because the `SYSTEM` model uses parameterized values as shown in Example 2-5, `cost.msf` must specify the partition name along with the variable name—for example, `main.num`—so that SystemBuild can find and update the values.

Example 2-5 Specifying Feedback Control Logic

```

Function [out]=cost(p,it)
main.num=[p(1), p(2)];
main.den=[1, p(2)];
main.k1=p(3);
main.k2=p(4);
t=[0:.1:10]';
u=ones(t);
[,y]=sim("SYSTEM",t,u,{simclock=0,simmessage=0});
# minimize the difference between the command and the output:
out(1)=norm(y-ones(y));
out(2)=100*(max(y)-1); # percent overshoot
# plot at major iterations:
if it>0
    plot(t,y,{keep})?
endif
endFunction

```

The `cost.msf()` function accepts model parameters in the K vector. First, the p vector is converted into the SystemBuild model parameters `NUM`, `DEN`, K_1 , and K_2 . Next, the time and simulation vectors are created and used to simulate the step response. After the output signal is returned, `cost()` computes the rise time and overshoot. Whenever a major iteration is completed, `cost()` displays the step response. The `plot()` function keeps the plots for final comparison.

Optimization

1. To view the initial step response, type the following values from Table 2-1:

```

p0=[1, 1, 1, 1]';
num=[p0(1), p0(2)];
den=[1, p0(2)];
k1=p0(3); k2=p0(4);

```

2. Analyze `SYSTEM` to verify it:

```
analyze("SYSTEM")
```

3. Try `cost()` with the initial parameters:

```
cost(p0,1)
```

The result in Figure 2-13 shows that the step response is stable, but the overshoot is greater than the required 5%.

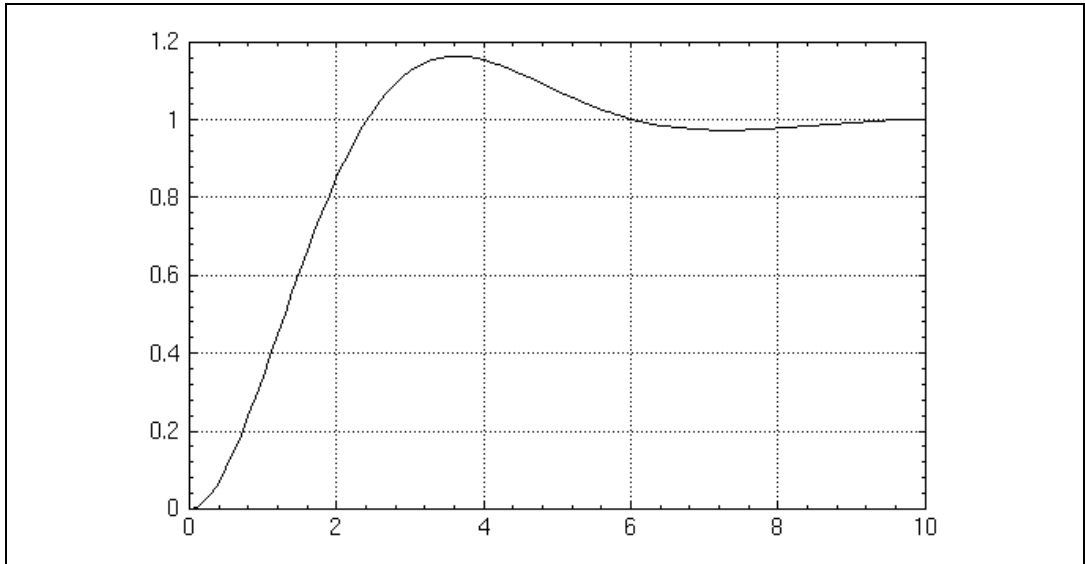


Figure 2-13. Step Response with Initial Parameters

4. Start `optimize()`.

Use the default value of ρ , 1. You are arbitrarily guessing you might need a large number of major iterations (20), but you are reducing the number of minor iterations to 5 in hopes of shortening running time. Set the perturbation parameter `delta` to 1×10^{-4} and the tolerance to 1×10^{-3} .

```
[P,JH,L] = opti(p0,{icmin=0,icmax=5,
pmin=zeros(4,1),pmax=ones(4,1)*100,
rho=1, majit=20, minit=5,delta=1e-4,tol=1e-3})
```

As the optimization proceeds, you will see the following warning in the commands window message area:

MINOR OPTIMIZATION ROUTINE DID NOT CONVERGE IN THE SPECIFIED NUMBER OF MINOR ITERATIONS. YOU MAY NEED TO INCREASE THE NUMBER OF MINOR ITERATIONS.

This message indicates an intermediate point in the processing. It does not imply that the algorithm has failed or that the solution is invalid. Optimization completes in seven iterations as shown in Example 2-6.

Example 2-6 Optimization Iterations

```
Beginning major iteration 7, J=2.93744
Beginning minor iteration 1
Updated parameters: 2.88236 0.476158 0.527184...
Completed in 7 iterations
```

P (a column vector) =

```
2.88236
0.476158
0.527184
1.03561
```

JH (a row vector) = 3.24032 3.14864 3.13624 ...

L (a scalar) = -0.0115143

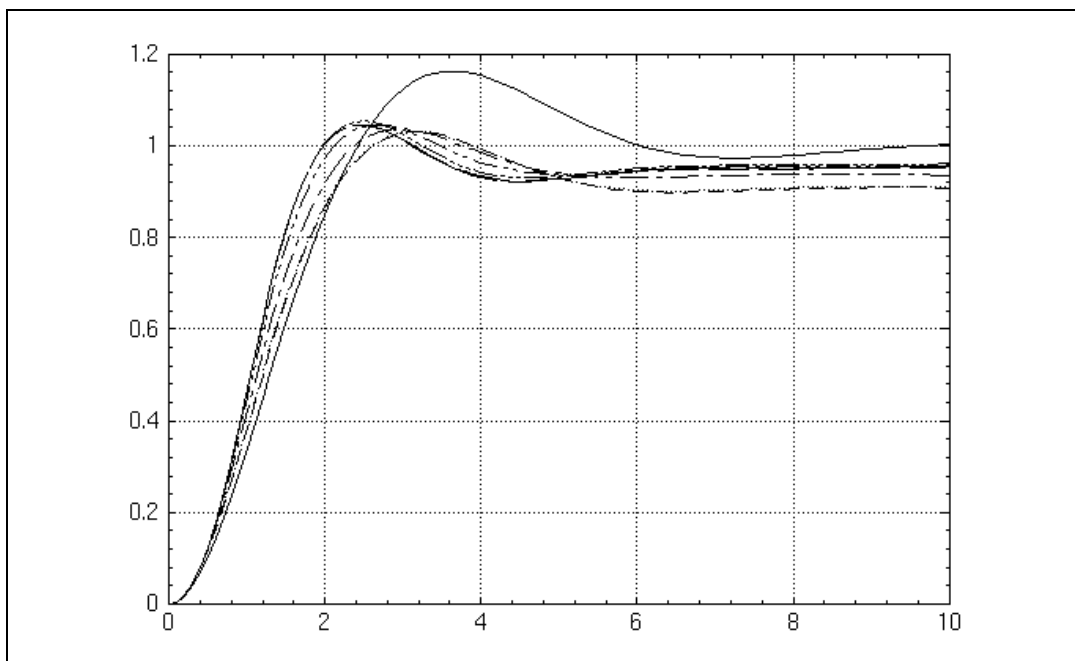


Figure 2-14. System Responses

Analysis

`optimize()` computes a parameter set that drastically improves the step response of the closed-loop system. The overshoot is limited to 5% as shown in Figure 2-14.

Figure 2-15 plots the initial step response against the final response.

```
erase  
cost (p0, 1)  
cost ([P; 2.93744], 1)
```

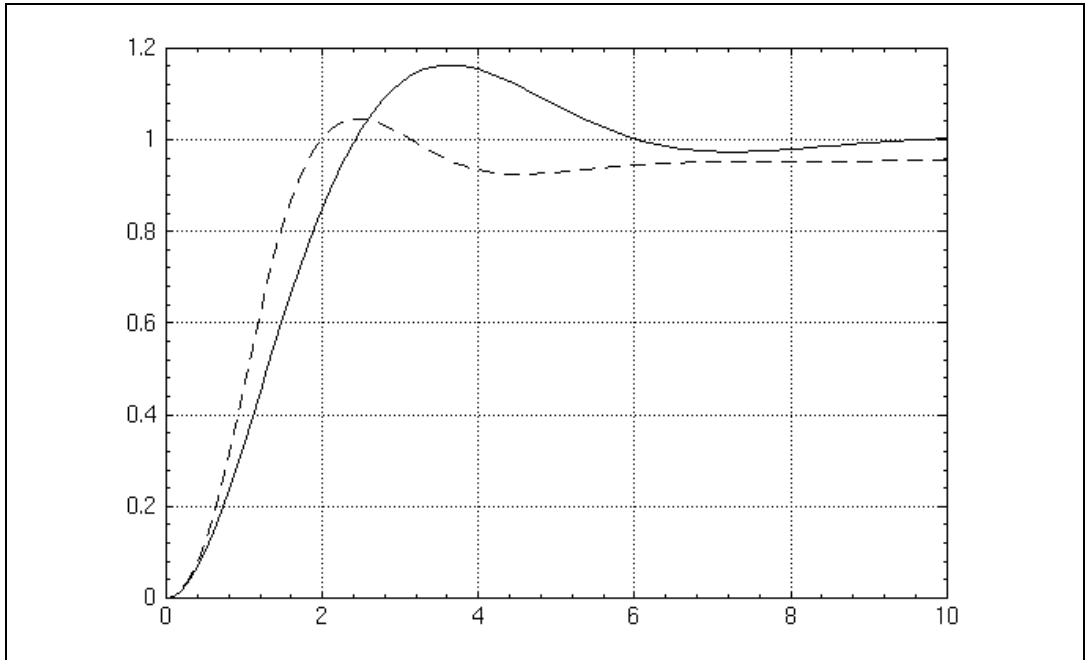


Figure 2-15. Initial Step Response versus Optimal Response

Quadratic Programming

Quadratic Programming (QP) finds a number of useful applications in engineering, finance, and operations research. Over the years, a number of techniques have been used, but recent algorithmic advances have made QP computation very efficient and robust. The Xmath Optimization Module employs this new class of algorithms in the function `qpopt()`.

This chapter describes the structure and implementation of the QP solver. The *QPOPT() Function* section covers the mathematical formulation of the QP problem and `qpopt()` syntax. The *QPOPT Algorithm* section highlights the algorithmic details that make up `qpopt()`. The application example in the *Application Example* section uses `qpopt()` to solve a nonlinear curve fitting problem.

QPOPT() Function

```
[p, y, jh] = qpopt(q, {c, a, b, xmin, xmax, tol})
```

The `qpopt()` function solves the quadratic programming problem.

$$\begin{aligned} \min_p \quad & \frac{1}{2}(p^T Q p) + c^T p \\ \text{subject to} \quad & A p = b \\ & p_l \leq p \leq p_u \end{aligned} \tag{3-1}$$

where

- p is the $n \times 1$ vector of real optimization parameters
- Q is the $n \times n$ quadratic cost matrix (symmetric) in the objective function
- c^T is the $1 \times n$ linear cost vector
- A is the $m \times n$ linear equality constraints matrix (optional)
- b is the $m \times 1$ constraint vector
- p_l is the lower parameter bound
- p_u is the upper parameter bound

Regarding the outputs:

| | |
|------|---|
| p | is the optimal solution |
| y | is the optimal shadow price (vector of Lagrange multipliers for equality constraints) |
| jh | is the history of objective function values over iterations |

QPOPT Algorithm

The QP solver in the Xmath Optimization Module is an extension of Karmarkar's interior point algorithm for solving the linear programming problem. The algorithm uses affine scaling transformation and optimization over a trust ellipsoid region. It creates a series of interior feasible points that converge to the optimal solution.

Because the QPOPT algorithm is based on the interior point algorithm, it is more efficient than most common QP algorithms. Enhancements to the basic algorithm allow the solution to be computed to any user-defined accuracy. This also can mean substantial savings in situations where you require only five digits of accuracy instead of full machine precision. Together, these differences make the `qpopt ()` utility an attractive and efficient means of solving the QP problem.

The QP problem is defined as shown in Equation 3-1. Assume that the interior of the feasible region is not empty. From an interior solution p_k , the `qpopt` algorithm first forms $D = \text{diag}(d)$ as follows:

$$\begin{aligned} d^j &= \min(p_k^j - p_l^j, p_u^j - p_k^j) \text{ for } j = 1, \dots, n \\ D &= \text{diag}(d) \end{aligned} \quad (3-2)$$

and then solves the Quadratic Programming optimality conditions for p and y :

$$\begin{aligned} \min_p \quad & Qp + \mu_k D^{-2} p - A^T y + c^T - \mu_k D^{-1} e \\ & Ap = b \end{aligned} \quad (3-3)$$

where e is a vector of ones the same size as the p vector
 y is the Lagrange multiplier for the equality constraint
 m is the Lagrange multiplier at step k for the interior ellipsoidal trust region constraint

If p is not feasible or a satisfactory solution, set $\mu_k = 2\mu_k$ and solve Equation 3-3 again for p and y . Otherwise, return $p_k = p$. For more information about this algorithm, its convergence properties, and how it compares with other strategies, refer to the references listed in the [Additional Related Publications](#) section of Chapter 1, [Introduction](#).

For a complete description of each `qpopt ()` input, output, and keyword, refer to the `qpopt` topic of the *MATRIXx Help*.

Application Example

Curve Fitting with Quadratic Programming

This example shows how `qpopt ()` is used to fit a polynomial equation to test data. Specific heat measurements for superheated steam are given as a function of absolute temperature in Table 3-1.

Table 3-1. Specific Heat Measurements for Superheated Steam

| i | T(k) | $\frac{\text{cal}}{(\text{deg} - \text{mol})} C_p$ |
|----------|-------------|--|
| 1 | 400 | 8.4944 |
| 2 | 425 | 8.5589 |
| 3 | 450 | 8.5800 |
| 4 | 475 | 8.6728 |
| 5 | 500 | 8.6924 |
| 6 | 525 | 8.8236 |
| 7 | 550 | 8.9816 |

You want to fit this data to a second order polynomial:

$$\bar{C}_p = \theta_1 T^2 + \theta_2 T + \theta_3$$

The problem is to find $\theta = [\theta_1, \theta_2, \theta_3]^T$ such that the sum of the squared estimation error is minimized. More formally,

$$\sum_{i=1}^7 [C_p(i) - \bar{C}_p(\theta, T(i))]$$

noting that:

$$\begin{aligned} \bar{C}_p(\theta, T(i)) &= A\theta \\ &= \begin{bmatrix} T_1^2 T_1 & 1 \\ T_2^2 T_2 & 1 \\ \dots & \dots \\ T_7^2 T_7 & 1 \end{bmatrix} \end{aligned}$$

You can rewrite Equation 3-1 as:

$$\min_{\theta} C_p - \langle A\theta, C_p - A\theta \rangle$$

or,

$$\min_{\theta} C_p^T C_p - (2C_p^T A\theta + \theta^T A^T A\theta)$$

which is the same as:

$$\min_{\theta} \frac{1}{2} \theta^T (Q\theta + c^T \theta)$$

This is the unconstrained quadratic programming problem with:

$$c^T = -2C_p^T A$$

$$Q = 2A^T A$$

Solving the Curve Fitting Problem

To solve the problem, set up the cost variables in Xmath by completing the following steps:

1. Enter the following:

```
set format shorte
T=[400:25:550]';
CP=[8.4944, 8.5589, 8.5800,8.6728, 8.6924, 8.8236,
8.9816];
A=[T^2, T, ones(T)];
C=(-2*CP*A)';
Q=2*A'*A;
```



Note All QP vectors must be column vectors.

2. Call `qpopt ()`:

```
[THETA, Y, JH]=qpopt(Q, {C=C})
```

The following results appear:

Problem is unconstrained.

Solvable unbounded, unconstrained problem.

The unique stationary solution has been found

THETA (a column vector) =

1.660190e-05

-1.276695e-02

1.096324e+01

Y (a scalar) = 0.000000e+00

JH (a scalar) = -5.283228e+02

- Find the predicted values of C_p (CPHAT):

```
CPHAT=A*THETA
```

```
CPHAT (a column vector) =
```

```
8.512760e+00
8.536000e+00
8.579993e+00
8.644738e+00
8.730236e+00
8.836486e+00
8.963488e+00
```

- Plot the predicted values of C_p against the test data to check the validity of the fit. The results are shown in Figure 3-1.

```
plot(T,CPHAT,{xlab="T (deg K)",ylab="Cp"})
```

```
plot(T,CP,{keep,markerstyle=5,
```

```
legend=["predicted values","observed values"]})
```

The experimental data and the quadratic curve fit show good agreement.

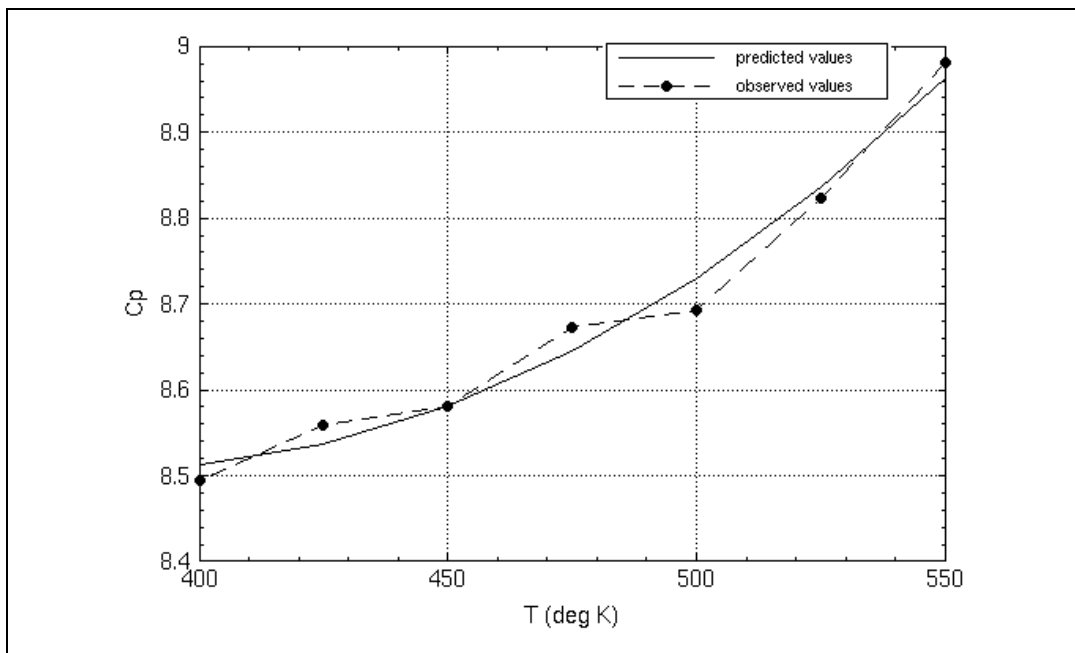


Figure 3-1. Predicted versus Observed Specific Heat

Linear Programming

Linear Programming (LP) solves a number of interesting and practical problems. It is used to solve network flow problems as well as inventory control and investment portfolio problems. For general linear programming problems of moderate size (less than 80 variables), the `lpopt()` function in the Xmath Optimization Module performs very well.

For this class of problems, the Karmarkar based algorithm in the `lpopt()` function offers substantial advantages over the commonly used Simplex algorithm. An important difference is that, unlike Simplex, the `lpopt()` algorithm computes optimal search directions in the feasible set rather than using exhaustive search techniques. Just as importantly, the Linear Programming algorithm in the Xmath Optimization Module lets you control the accuracy of the solution. This is possible because the problem solving strategy computes upper and lower solution bounds at each step. You can control how closely the bounds approach each other. By increasing the value of `tol` (tolerance), you can obtain a less accurate solution with less computational effort. Conversely, decreasing `tol` gives a more accurate, but computationally costly solution. You decide how much effort you want to devote to your problem.

LPOPT() Function

```
[p, y, jh, zh] = lpopt(a, b, c, {z1, tol, beta})
```

The `lpopt()` function solves the general linear programming problem:

$$\begin{aligned} \min \quad & c^T p \\ & p \\ & Ap = b \\ & p \geq 0 \end{aligned} \tag{4-1}$$

where

- c^T is the $1 \times n$ cost vector
- A is the $m \times n$ coefficient matrix of linear constraints
- b is an $m \times 1$ vector of the right side of the constraint equation
- p is the $n \times 1$ vector of parameters

LPOPT Algorithm

The `lpopt()` solver in the Xmath Optimization Module uses the Karmarkar-type algorithms to compute the solution to the classical linear programming problem as shown in Equation 4-1.

At each iteration, the algorithm transforms p to the center of a Simplex and computes the feasible solution to the dual linear programming problem. It uses this updated dual problem solution to compute the parameter for the interior ellipsoid optimization problem. Finally, the algorithm transforms the interior point solution back into the correct parameter space. This procedure is repeated until the termination tolerance on upper and lower cost bounds is met.

More formally, the algorithm proceeds as follows:

$$\begin{array}{ll}
 \text{set} & (k = 0) \text{ given } (p_0, z_0) \\
 & W = \text{diag}(p_k) \\
 \text{solve for } y_1 & (AW^2A^T + bb^T)y_1^T = b \\
 \text{solve for } y_2 & (AW^2A^T + bb^T)y_2^T = AW^2c \\
 \text{compute} & u = (c^TW - y_2AW, y_2b)^T \\
 & v = (y_1AW, 1 - y_1b)^T \\
 \text{then let} & \theta(z) = \min\{u - zv\} \\
 & z_1 = \sup\{z: \theta(z) \leq 0\} \\
 & z_{k+1} = \begin{cases} z_1 & \text{if } z_1 \geq z_k \\ z_k & \text{otherwise} \end{cases} \\
 \text{then} & d = u - z_{k+1}v - \frac{c^T p_k - z_{k+1}}{n+1} \\
 & a = e\beta \frac{d}{\|d\|} \\
 \text{then} & P_{k+1} = \frac{Wa(1:n)}{a(n+1)}
 \end{array}$$

where b is a constant
 e is a vector of ones the same size as the p vector
 n is the dimension of the parameter vector

The algorithm continues until $(c^T p_k - z_k) < \text{tol}$.

Setting up and running the `LPOPT()` function is straightforward in the Xmath environment. Create the A, B, and C matrices as Xmath variables and run the linear programming function with any desired option. For a complete description of the function syntax, refer to the *MATRIXx Help*.

For problems with inequality constraints, you first must map the problem into one with equality constraints using slack and/or surplus variables. This procedure¹ is illustrated in the *Application Example* section.

Application Example

Refinery Optimization

An oil refinery in Placitas, New Mexico, must order crude oil to manufacture its three main products: kerosene, gasoline, and jet fuel. Two types of crude oil are available as feedstock material: crude 1, which sells for \$27 per barrel, and crude 2, which sells for \$25 per barrel (current prices will vary). The fractional amounts of product that can be obtained from each barrel of the two types of crude are given in Table 4-1.

Table 4-1. Fractional Amounts of Products in Crude Oil

| Feedstock | Kerosene | Gasoline | Jet fuel |
|-----------|----------|----------|----------|
| crude 1 | 0.35 | 0.2 | 0.25 |
| crude 2 | 0.29 | 0.4 | 0.25 |

The refinery must produce the following numbers of barrels for each product:

Kerosene 900,000

Gasoline 800,000

Jet Fuel 500,000

How many barrels of each type of crude should the refinery buy to meet the production requirements and minimize its cost?

¹ Luenberger, D. G., *Linear and Nonlinear Programming*, Addison Wesley Publishing Company, 1987.

Formulation

If you let $p = [p_1 \ p_2]^T$, where p_1 is the number of barrels of crude 1 and p_2 is the number of barrels of crude 2, the problem can be expressed as:

$$\begin{aligned} & \min \\ & \quad p \\ & Ap \geq b \\ & p \geq 0 \end{aligned} \quad c^T p$$

where $c^T = [27, 25]$

$$A = \begin{bmatrix} 0.35 & 0.29 \\ 0.20 & 0.40 \\ 0.25 & 0.25 \end{bmatrix}$$

$$B = \begin{bmatrix} 900 & 000 \\ 800 & 000 \\ 500 & 000 \end{bmatrix} \quad (4-2)$$

To convert the problem into standard form with equality constraints only, introduce the surplus variables ($y = [y_1 \ y_2 \ y_3]^T$) and then formulate the problem as:

$$\begin{aligned} & \min \quad c_1^T p \\ & \quad p \\ & A_1 z = b \\ & z \geq 0 \end{aligned}$$

where

$$\begin{aligned} & \min \quad c_1^T p \\ & \quad p \\ & A_1 z = b \\ & z \geq 0 \end{aligned}$$

where $z = \begin{bmatrix} p \\ y \end{bmatrix}$ optimization parameters and,

$$A_1 = \begin{bmatrix} -1 & 0 & 0 \\ A & 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

$$c_1^T = [c^T \ 0 \ 0 \ 0]$$

and b is as given in Equation 4-2. Solve the problem with an optimization tolerance of 1×10^{-3} , and a relative step size of 0.9.

Optimization

To save calculation time, let the b values represent thousand barrel lots and c values represent price per thousand barrels. To define the cost/constraint variables in Xmath, enter the following:

```
set format shorte
a = [
    0.35, 0.29, -1.00, 0.00, 0.00
    0.20, 0.40, 0.00, -1.00, 0.00
    0.25, 0.25, 0.00, 0.00, -1.00 ];
b = [ 900, 800, 500 ]';
c = [ 27000, 25000, 0, 0, 0 ]';
```

Call `lpopt ()`:

```
[p, y, jh, zh]=lpopt(a,b,c)
```

As the solution proceeds, `lpopt ()` displays the following:

```
Starting search for a feasible solution...
Equality constraint error: 5.016295e-01
Equality constraint error: 5.224094e-01
Equality constraint error: 6.532657e-01
Equality constraint error: 0.000000e+00
Feasible solution found.
```

As `lpopt ()` searches for the feasible solution set, it displays the amount of error in the equality constraints. When it is found, the algorithm approaches the optimal solution with successive upper and lower bounds to the cost.

Starting to search for the optimal solution...

Objective Value: 8.245058e+07 Lower Bound: 4.7 ...

Objective Value: 7.590557e+07 Lower Bound: 7.0 ...

Objective Value: 7.354306e+07 Lower Bound: 7.0 ...

Objective Value: 7.284624e+07 Lower Bound: 7.2 ...

Objective Value: 7.268279e+07 Lower Bound: 7.2 ...

Objective Value: 7.264562e+07 Lower Bound: 7.2 ...

Objective Value: 7.263681e+07 Lower Bound: 7.2 ...

Objective Value: 7.263477e+07 Lower Bound: 7.2 ...

Optimal solutions and shadow prices have been found.

p (a column vector) =

1.560839e+03

1.219685e+03

2.205234e-03

4.169563e-02

1.951309e+02

y (a column vector) =

7.073171e+04

1.121951e+04

1.081438e-03

jh (a row vector) = 2.645557e+08 8.245058e+07...

zh (a row vector) = 4.700973e+07 4.700973e+07...

Analysis

The final solution is returned in the `p` vector. To meet the production requirements with minimum cost, 1.5608×10^6 barrels of crude 1 and 1.2197×10^6 barrels of crude 2 will be required. The total cost is the last element of the `jh` vector, \$72.63 million dollars. The `y` vector contains the shadow prices for the three products produced by the refinery. Inspection of `y` shows that the cost to produce barrel number 900,001 of kerosene is \$70.73 and that the incremental cost for a barrel of gasoline is \$11.22.

The `obh` vector contains the history of the objective function during the algorithm, and the `zh` vector shows the lower bound on the optimal cost for each step of the algorithm. Plot the vectors to examine convergence:

```
plot([zh',jh'],
     {xlabel="Iteration Number",ylabel="$ Cost"})
```

Figure 4-1 shows how the upper and lower bounds converge quickly to an optimal estimate.

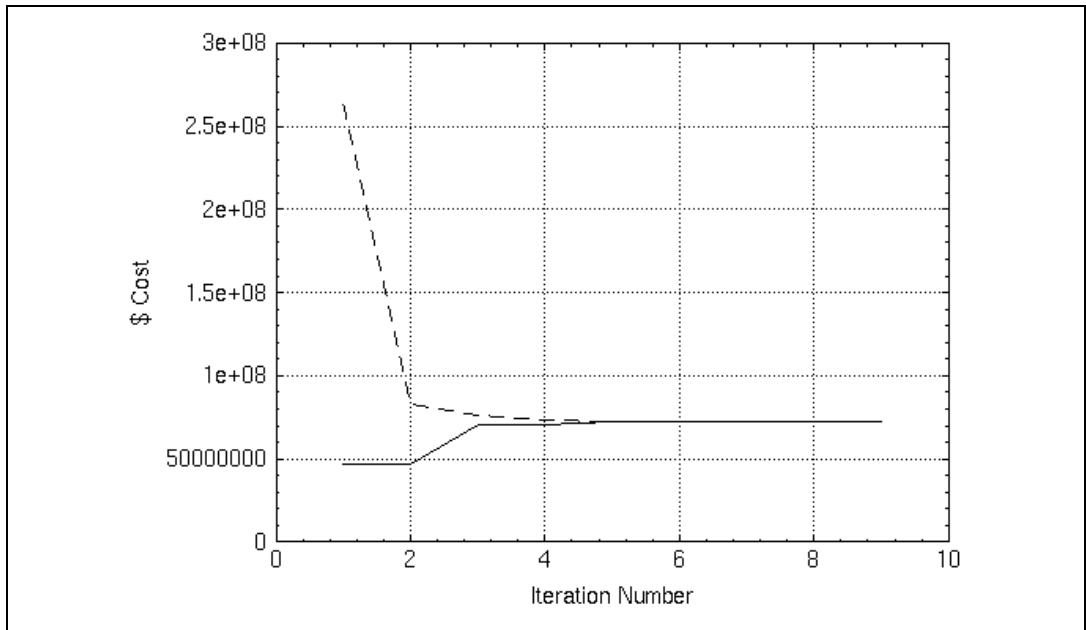


Figure 4-1. Convergence of Upper and Lower Bounds for an Optimal Estimate



Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources at ni.com/support include the following:
 - **Self-Help Resources**—For immediate answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
 - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Developer Exchange at ni.com/exchange. National Instruments Application Engineers make sure every question receives an answer.
- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Index

A

algorithm

iterative, 2-9

Karmarkar (interior point), 3-2

linear programming, 4-1

lpopt(), 4-2

optimization, 2-11

quadratic programming, 3-2

simplex, 4-1

C

constraints, adjusting iterations to, 2-11

conventions used in the manual, *iv*

cost and feasible solution, 2-5, 2-13

cost MSF, 2-3

computation specifications, 2-4

constraints, 2-2

parameterized variables in, 2-27

parameters to function call, 2-2

template, 2-3

cost.msf(), 2-2, 2-3, 2-5

D

diagnostic tools (NI resources), A-1

documentation

conventions used in the manual, *iv*

NI resources, A-1

drivers (NI resources), A-1

E

example

box design, 2-14

curve fitting, 3-3

feedback control design, 2-24

refinery optimization, 4-3

trajectory optimization, 2-17

examples (NI resources), A-1

F

function switching, 2-9

H

help, technical support, A-1

I

instrument drivers (NI resources), A-1

K

KnowledgeBase, A-1

L

linear programming (LP)

classical problem, 4-2

procedure, 4-3

linear programming problem, 4-1

lpopt(), 4-1

M

major and minor iterations, 2-10
MATRIXx Help, 1-3
MSF, cost(), 2-2, 2-3

N

NI support and services, A-1
nomenclature, 1-2

O

optimization
 constraints, 2-7
 convergence, 2-8
 discontinuities, 2-8
 evaluation, 2-6
 example, 2-15
 feasible solution, 2-5
 parameters, 2-2
optimize, 2-1

P

penalty parameter. *See* rho
piecewise constant functions, 2-8
programming examples (NI resources), A-1

Q

quadratic programming optimization
 algorithm, 3-2
 solution precision, 3-2
quadratic programming problem, 3-2

R

rho, 2-5
 changing when problem is poorly
 behaved, 2-11
 initial guess, 2-6

S

save, 2-3
software (NI resources), A-1
support, technical, A-1

T

technical support, A-1
training and certification (NI resources), A-1
troubleshooting (NI resources), A-1

W

Web resources, A-1
weighting factor (ρ). *See* rho