This book is printed on acid-free paper. ♾

Published simultaneously in Canada.

Library of Congress Cataloging in Publication Data:

Venkataraman, P.
Applied optimization with MATLAB® Programming / P. Venkataraman.
p. com.
"A Wiley-Interscience publication."
ISBN 0-471-34958-5 (cloth : alk. paper)
1. Mathematical optimization--Data processing. 2. MATLAB. I. Title.

QA402.5.V42 2001
519.3—dc21 2001026938

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To coexistence of peace and harmony between the different melodies of people everywhere

CONTENTS

PREFACE

The subject of optimization is receiving serious attention from engineers, scientists, managers, and most everybody else. This is driven by competition, quality assurance, cost of production, and finally, the success of the business enterprise. Ignoring the practice of optimization is not an option during current times.

Optimization is practiced through software programs and requires significant computer resources. The techniques of optimization have not changed significantly in recent years, but the areas of applications have mushroomed at a significant rate. Successfully embedding the use of optimization in professional practice requires at least three prerequisites. They include mathematical modeling of the design problem, knowledge of computer programming, and knowledge of optimization techniques. Many special-purpose optimization software packages that relax the required knowledge of programming are available today. To use them efficiently, the remaining two areas still have to be addressed.

There are several excellent books on the subject of optimization, a few of them released recently. Most of them cover the subject in depth, which is necessary because the mathematical models are nonlinear and require special techniques that are usually not part of any core curriculum. All of the books assume that the reader is familiar with a programming language, traditionally FORTRAN, and recently, C. Another assumption frequently made is that the optimization techniques will be implemented in a mainframe computing environment. Such a combination is extremely difficult for the self-learner even with enormous motivation.

An explosion of inexpensive desktop computing resources aids engineering design practice today. Paralleling this development is the availability of extensive resources on the Internet, both for learning and deployment. This is joined by the availability of software systems that provide an opportunity for handling mathematics, graphics, and programming in a consistent manner. These software systems are significantly easy to master compared to the higher-level programming languages of the previous years. This book primarily seeks to harness this triangulation of services to provide a practical approach to the study of design optimization. The book uses MATLAB® to illustrate and implement the various techniques of optimization. MATLAB is a product from MathWorks, Inc.

This book attempts to accomplish two important objectives. The first is to assist the reader in developing MATLAB programming skills. It introduces the reader to the use

of symbolic, numerical, and graphical features of MATLAB. It integrates this powerful combination during the translation of many algorithms into applied numerical techniques for design optimization. There is a constant enhancement of the programming skills throughout the book. The second objective is the primary task of the book—*to communicate and demonstrate various numerical techniques that are currently used in the area of optimal design.*

All of the numerical techniques are supported by MATLAB code available as computer files. These files are available on a companion web site www.wiley.com/venkat. It is necessary to visit the web site to download the files to follow all of the examples. The book will be essentially incomplete without these files. At the web site you will also find useful links to other resources, a web course on MATLAB programming, and updated information about *Applied Optimization with MATLAB® Programming.*

The decision to locate the files on a web site provides an avenue to shadow the dynamic changes in MATLAB software itself, which is being transformed in a major way almost every year. It provides a mechanism to correct bugs in the code in a timely manner. It establishes an opportunity to improve the code through reader suggestions and provides a way for the author to keep the reader engaged with new developments. It does cut down the size of the book enormously by locating all of the pages of code outside the book.

An unusual feature of the book is the inclusion of *discrete optimization* and *global optimization*. Traditional continuous design optimization techniques do take up a significant portion of the book. Continuous problems generally allow the algorithms to be mathematically developed in a convincing manner, and these ideas could be the core of original algorithms developed by the readers to address their own special needs. Every chapter includes additional nontrivial examples that often present a novel use or extension of the optimization techniques in the chapter. The mathematical algorithms and examples are accompanied by MATLAB code available at the companion web site. The numerical techniques usually include strong graphical support to illustrate the operation of the techniques.

Discrete and global optimization techniques are based on very different algorithms, and each of them can easily justify full-fledged independent courses devoted exclusively to their study. In a comprehensive book such as this, exposure to these areas of optimization cannot escape being brief. The discussion, however, is accompanied by numerical implementation and nontrivial examples. There is a caveat with respect to all the code presented in the book—it has only been tested for the example under discussion. These pieces of code are not likely to supplant the *Optimization Toolbox* (part of the MATLAB family of software) that extends the capability of MATLAB. The code in the book does provide a starting point or a seed for exploration and development of a personal approach to the subject. The reader is strongly urged to take advantage of this possibility. The author is hopeful that readers will be able to share their experiences through the web site.

The book can be used as a formal text on design optimization. The preferred setting is a computational laboratory with the possibility of hands-on computation. A formal

classroom setting without computational experience is also feasible. In this case the algorithms can be presented and the numerical results illustrated. The senior/graduate students in various disciplines, especially engineering, are the target audience. Optimization techniques are an important tool to solve design problems in all professional areas of study. Many illustrations are from the area of mechanical engineering reflecting the experience of the author.

Independent learners, particularly professionals who need to understand the subject of optimization, should also find the book very useful. A reader who has access to MATLAB software can use the book to its best advantage. The book is largely self-contained and develops all necessary mathematical concepts when needed. Abstract mathematical ideas of optimization are introduced graphically and through illustrative examples in the book. In many instances, the operation of the numerical technique is animated to show how the design is changing with iterations. This has been possible because MATLAB graphics are rich in features and simple to incorporate, making them effective for conveying the ideas. It is for this reason the reader is exposed to graphical programming early on, including a chapter on *graphical optimization*. To balance the development of programming skills and the presentation of optimization concepts, programming issues are incrementally explored and implemented. Investment of time in developing programming skills is the most effective way to imbibe them. The author is not aware of shortcuts. Students are expected to program independently, make mistakes, debug errors, and incorporate improvements as part of their learning experience delivered through this book.

The book proceeds at a brisk pace to provide experience in MATLAB programming and communicate ideas in optimization in a reasonable number of pages. This could not be possible without taking advantage of the accompanying web site to locate much of the code for algorithms and examples. Merely running the code will not significantly enhance the reader's programming skills. It is important for the student to understand the code also. To assist in this process, the code carries liberal comments often corresponding to the algorithm that is being implemented. The author expects the reader to have read the comments. The accompanying code allows the instructor to assign both programming assignments as well as solutions to design problems. The most effective of such assignments would be to explore the same design problem using several techniques. The author has attempted to make this book user friendly, particularly through the code on the web site.

For academic programs that provide prior exposure to MATLAB programming, all the topics in the book can be covered in a single term. It is recommended that the course include a final open-ended design project to provide an avenue for application of the ideas learned in the course. It would be a good learning experience for the students to discover and formulate such a project themselves following a simple problem statement or identifying a design need expressed by the instructor. For programs where this book will also provide a formal exposure to MATLAB programming, it is likely that only continuous optimization problems can be covered effectively in a single term. The remaining can be assigned for independent self-study. Since discrete and global optimization are important contemporary optimization

approaches, the instructor can choose to include only a subset of continuous optimization algorithms to allow the inclusion of these popular techniques. If the student has access to the *Optimization Toolbox* from MATLAB, then it can be integrated into the course for handling the final design project. The last chapter provides an overview on the use of the toolbox.

This book started out using MATLAB *Version 5*. About halfway through the book the transition to MATLAB *Version 5.3* was complete. At the completion of the manuscript MATLAB *Version 6* was shipping for some time. As of this time, *Version 6.1* is available. Most institutional MATLAB licenses are usually based on subscription so that they will have the latest version of the software. In a sense, and this is true of all books that depend on particular software systems, the book may appear out of date with respect to the software. This is not really so. While the usage of MATLAB functions may change between versions, there should be suitable warnings and instruction about the different usage. An important insurance is built into the book, particularly to handle such changes if they arise. Sprinkled throughout the book is the habit of the using the *online help*. In most instances this was primarily done to expose the reader to alternative use of the same command. Another significant change the manuscript had to endure was the loss of access to a UNIX version because of the university's change to distributed computing on PC clusters. The author's experience with the early chapters suggests that this should not matter. Except for external files and resources, the use of MATLAB as illustrated in the book is indifferent to the various operating systems (Windows, Mac, UNIX, or Linux).

The topics chosen for inclusion and the simplicity of presentation of the topics are directly related to the experience of teaching the course on optimization at the senior/graduate level in the Department of Mechanical Engineering for over ten years. Experience proved that comprehension improved with simple illustrations, however complicated the mathematical ideas. On the other hand, straightforward mathematical expressions elicited no interest or understanding. The emphasis on application was important to keep the students' attention. In this connection the author would like to thank all his students for pushing him for simple explanations, for providing positive interaction, and for their contribution of original and useful ideas in the discussions on many techniques.

This book would not be possible without the work published by scores of authors and researchers in the area of design optimization over the years. The purpose of this book is to bring readers up to speed in generating and translating their own ideas into promising numerical techniques and deploying them through MATLAB. The book does not make any significant compromise relating to the mathematical ideas necessary to present the subject of optimization. The focus is on applications and the recognition that optimization techniques can be used in a wide variety of situations. Sometimes, practical discussion and numerical experiments are substituted for higher-order mathematical analysis and rigor. A few of the algorithms are simplified so that they can be better appreciated. Some of the examples are explored through different techniques to illustrate the impact of different algorithms and approaches.

The book was made possible through support from John Wiley and Sons, Inc., and MathWorks, Inc. Sincere thanks are owed to Bob Argentieri, senior editor at John Wiley for accepting the proposal, and who all along displayed a lot of patience in getting the book moving forward. Same is due to Bob Hilbert, associate managing editor at John Wiley, for his impressive work at cleaning up the manuscript. Brian Snapp, New Media editor at John Wiley, created the companion web site (www.wiley.com/venkat) and will also be maintaining it. Naomi Fernandes from MathWorks, Inc., saw to it that I had the latest version of MATLAB as soon as it was available. My regard for Dr. Angelo Miele, Professor Emeritus, at Rice University is more than can be expressed in these lines. It was he who introduced me to the subject of optimization and demonstrated the effectiveness of simple presentation. I will always regard him as a great teacher. Of course, my family deserves special mention for putting up with all the "Not now," "Later," "How about tomorrow?" during debugging the code. Special thanks are to Archana and Vinayak, my offspring, for their patience, understanding, and encouragement. The author apologizes for any shortcomings on the presentation and welcomes comments, criticisms, and suggestions for improvement at all times.

P. VENKATARAMAN

Rochester, New York

1

INTRODUCTION

Optimization has become a necessary part of design activity in all major disciplines. These disciplines are not restricted to engineering. The motivation to produce economically relevant products or services with embedded quality is the principal reason for this inclusion. Improved production and design tools, with a synergistic thrust through inexpensive computational resources, have aided the consideration of optimization methods in new developments, particularly engineering products. Even in the absence of a tangible product, optimization ideas provide the ability to define and explore problems while focusing on solutions that subscribe to some measure of usefulness. Generally, the use of the word optimization implies the best result under the circumstances. This includes the particular set of constraints on the development resources, current knowledge, market conditions, and so on. Every one of us has probably used the term at some time to describe the primary quality of our work or endeavor. It is probably the most used or abused term in advertising and presentations. Nevertheless, the ability to make the best choice is a perpetual desire among us all.

Optimization is frequently associated with design, be it a product, service, or strategy. Aerospace design was among the earliest disciplines to embrace optimization in a significant way driven by a natural need to lower the tremendous cost associated with carrying unnecessary weight in aerospace vehicles. Minimum mass structures are the norm. Optimization forms part of the psyche of every aerospace designer. Saving on fuel through trajectory design was another problem that suggested itself. Very soon the entire engineering community could recognize the need to define solutions based on merit. Recognizing the desire for optimization and actually implementing were two different issues.

Until recently, for much of the time, optimization was usually attempted only in those situations where there were significant penalties for generic designs. The application of optimization demanded large computational resources. In the nascent years of digital computation these were available only to large national laboratories and research programs. These resources were necessary to handle the nonlinear problems that are associated with engineering optimization. As a result of these constraints most of the everyday products were designed without regard to optimization. This includes everything you see around you or use in your daily life. It is inconceivable that the new generation of replacement products, like the car, the house, the desk, or the pencil, redesigned and manufactured today are not designed optimally in one sense or another.

Today, you would definitely explore procedures to optimize your investments by tailoring your portfolio. You would optimize your business travel time by appropriately choosing your destinations. You can optimize your commuting time by choosing your time and route. You can optimize your necessary expenditure for living by choosing your day and store for shopping. You can optimize the useful time you connect to the Internet by determining your time of connection. You can buy software that will optimize your connection to the Internet. You can buy books or read articles that tell you how to perform these various optimizations. The above activities primarily relate to services or strategy. It is now apparent that every activity, except aesthetic, provides the scope for optimization. This justifies looking at the study of optimization as a tool that can be applied to a variety of disciplines. If so, the myriad of optimization problems from many disciplines can be described in a common way. This is the emphasis of the book.

The partnership between design and optimization activity is often found in engineering. This book recognizes that connection and many of the problems used for illustrations and practice are from engineering, primarily mechanical, civil, and aerospace design. Nevertheless the study of optimization, particularly applied optimization, is not an exclusive property of any specific discipline. It involves the discovery and design of solutions through appropriate techniques associated with the formulation of the problem in a specific manner. This can be done for example in economics, chemistry, and business management. In this book we will use the word *optimization* to also imply the techniques used in the practice of optimization.

1.1 OPTIMIZATION FUNDAMENTALS

Optimization can be applied to all disciplines. Qualitatively, this assertion implies multiple decision choices; implicitly recognizing the necessity of choosing among alternatives. This books deals with optimization in a quantitative way. This means that an outcome of applying optimization to the problem, design, or service must yield numbers that will define the solution, or in other words, numbers or values that will characterize the particular design or service. Quantitative description of the solution requires a quantitative description of the problem itself. This description is called a

mathematical model. The design, its characterization, and its circumstances must be expressed mathematically. Consider the design activity in the following cases:

- New consumer research indicates that people like to drink about 0.5 liter of soda pop at a time during the summer months. The fabrication cost of the redesigned soda can is proportional to the surface area, and can be estimated at $1.00 per square meter of the material used. A circular cross section is the most plausible given current tooling available for manufacture. For aesthetic reasons, the height must be at least twice the diameter. Studies indicate that holding comfort requires a diameter between 6 and 9 cm.
- A cantilever beam needs to be designed to carry a point load F at the end of a beam of length L. The cross section of the beam will be in the shape of the letter I (referred to as an I-beam). The beam should meet prescribed failure criteria. There is also a limit on its deflection. A beam of minimum mass is required to be designed.
- MyPC Company has decided to invest $12 million in acquiring several new Component Placement Machines to manufacture different kinds of motherboards for a new generation of personal computers. Three models of these machines are under consideration. Total number of operators available is 100 because of the local labor market. A floor space constraint needs to be satisfied because of the different dimensions of these machines. Additional information relating to each of the machines is given in Table 1.1. The company wishes to determine how many of each kind is appropriate to maximize the number of boards manufactured per day.

The above list represents three problems that will be used to define formal elements of an optimization problem. Each problem requires information from the specific area or discipline to which it refers. To recognize or design these problems assumes that the designer is conversant with the particular subject matter. The problems are kept simple to focus on optimization issues. Problems of such variety are important today. Recent advertisements in general consumer magazines illustrate that Alcoa (an aluminum manufacturer) is quite happy to have reduced the weight of the standard soda pop can by over 30% in recent years. A similar claim by the plastics industry with respect to the standard milk jug (gallon) is also evident in these magazines, although in this case the number is 40%. Roof collapses in the Northeast due to excessive snow

Table 1.1 Component Placement Machines

Machine Model	Board Types	Boards/Hour	Operators/Shift	Operable Hours/Day	Cost/Machine
A	10	55	1	18	400,000
B	20	50	2	18	600,000
C	18	50	2	21	700,000

during the 1998–1999 winter will have structural designers and homebuilders exploring the second problem. The vibrant stock market in these times has made balancing the investment portfolio more challenging. The third case may suggest a mathematical model appropriate for such decision making.

1.1.1 Elements of Problem Formulation

In this section, we will introduce the formal elements of the optimization problem. In this book, the term *product* also refers to a service or a strategy. It should be understood that optimization presupposes the knowledge of the design rules for the specific problem, primarily the ability to describe the design in mathematical terms. These terms include *design variables*, *design parameters*, and *design functions*. Traditional design practice, that is, design without regard to optimization, includes all of these elements although they were not formally recognized as such. This also justifies the prerequisite that you must be capable of designing the object if you are planning to apply the techniques of optimization. It is also a good idea to recognize that optimization is a procedure for searching the best design among candidates, each of which can produce an acceptable product. The need for the object or product is not questioned here, but this may be due to a decision based on optimization applied in another discipline.

Design Variables: Design variables are entities that identify a particular design. In the search for the optimal design, these entities will change over a prescribed range. The values of a complete set of these variables characterize a specific design. The number and type of entities belonging to this set are very important in identifying and setting up the quantitative design problem. It is essential that this choice capture the essence of the object being designed and at the same time provide a quantitative characterization of the design problem. In applied mathematical terminology, design variables serve as the unknowns of the problem being solved. Borrowing an analogy from the area of system dynamics and control theory, they are equivalent to defining the state of the system, in this case, the state of design. Typically, design variables can be associated with describing the object's size such as its length and height. In other cases, they may represent the number of items. The choice of design variables is the responsibility of the designer guided by intuition, expertise, and knowledge. There is a fundamental requirement to be met by this set of design variables, namely, they must be linearly independent. This means that you cannot establish the value of one of the design variables from the values of the remaining variables through basic arithmetic (scaling or addition) operations. For example, in a design having a rectangular cross section, you cannot have three variables representing the length, width, and area. If the first two are prescribed, the third is automatically established. In complex designs, these relationships may not be very apparent. Nevertheless, the choice of the set of design variables must meet the criterion of linear independence for applying the techniques of optimization. From a practical perspective, the property of linear independence identifies a minimum set of variables that can completely describe the

design. This is significant because the effort in obtaining the solution varies as an integer power of the number of variables, and this power is typically greater than 2. Meeting the requirement ensures reduced difficulty in mathematically exploring the solution.

The *set* of design variables is identified as the *design vector*. This vector will be considered a column vector in this book. In fact, all vectors are column vectors in the text. The length of this vector, which is n, is the number of design variables in the problem. The design variables can express different dimensional quantities in the problem, but in the *mathematical model*, they are distinguished by the character x. All of the techniques of optimization in this book are based on the *abstract mathematical model*. The subscript on x, for example, x_3, represents the third design variable, which may be the height of an object in the characterization of the product. This abstract model is necessary for mathematical convenience. This book will refer to the design variables in one of the following ways:

(1) $[\mathbf{X}]$—referring to the vector of design variables

(2) $\mathbf{X}$ or $\mathbf{x}$—referring to the vector again, omitting the square brackets for convenience if appropriate

(3) $[x_1, x_2, \ldots, x_n]^T$—indicating the vector through its elements. Note the superscript transposition symbol to identify it as a column vector.

(4) $x_i, i = 1, 2, \ldots, n$—referring to all of the elements of the design vector.

The above notational convenience is extended to all vectors in the book.

Design Parameters: In this book, these identify constants that will not change as different designs are compared. Many texts use the term *design parameters* to represent the design variables we defined earlier and do not formally recognize design parameters as defined here. The principal reason is that parameters have no role to play in determining the optimal design. They are significant in the discussion of modeling issues. Examples of parameters include material property, applied loads, and choice of shape. The parameters in the abstract mathematical model are represented in a similar form as the design vector, except that we use the character p. Therefore, $[\mathbf{P}]$, $\mathbf{P}$, $[p_1, p_2, \ldots, p_q]$ represent the parameters of the problem. Note that the length of the parameter vector is q. Except in the discussion of modeling, the parameters will not be explicitly referred to, as they are primarily predetermined constants in the design.

Design Functions: Design functions define meaningful information about the design. They are evaluated using the *design variables* and *design parameters* discussed earlier. They establish the *mathematical model* of the design problem. These functions can represent *design objective(s)* and/or *constraints*. As its name implies, design objective drives the search for the optimal design. The satisfaction of the *constraints* establishes the validity of the design. If not explicitly stated, the designer is responsible for identifying the objective and constraints. Minimize the mass of the

structure will translate to an *objective function*. The stress in the material must be less than the yield strength will translate to a *constraint function*. In many problems, it is possible for the same function to switch roles to provide different design scenarios.

Objective Function(s): The traditional design optimization problem is defined using a single objective function. The format of this statement is usually to minimize or maximize some quantity that is calculated using some design function. This function must depend, explicitly or implicitly, on the design variables. In the literature, this problem is expressed exclusively, without loss of generality, as a minimum problem. A maximum problem can be recast as a minimization problem using the negative or the reciprocal of the function used for the objective function. In the first example introduced earlier, the objective is to minimize cost. Therefore, the design function representing cost will be the objective function. In the second case, the objective is to minimize mass. In the third case, the objective is to maximize machine utilization. The area of single objective design is considered mature today. No new solution techniques for classical problems have been advanced for some time now. Today, much of the work in applied optimization is directed at expanding applications to practical problems. In many cases, this has involved creative use of the solution techniques. In the abstract mathematical model, the objective function is represented by the symbol f. To indicate its dependence on the design variables, it is frequently expressed as $f(x_1, x_2, \ldots, x_n)$. A more concise representation is $f(\mathbf{X})$. Single objective problems have only one function denoted by f. It is a scalar (not a vector). Note that although the objective function depends on **P** (parameter vector), it is not explicitly included in the format.

Multiobjective and *multidisciplinary* designs are important developments today. Multiobjective design, or multiple objective design, refers to using several different design functions to drive the search for optimal design. Generally, they are expected to be conflicting objectives. They could also be cooperating objectives. The current approach to the solution of these problems involves standard optimization procedures applied to a single reconstructed objective optimization problem based on the different multiple objectives. A popular approach is to use a suitably weighted linear combination of the multiple objectives. A practical limitation with this approach is the choice of weights used in the model. This approach has not been embraced widely. An alternative approach of recognizing a premier objective, and solving a single objective problem with additional constraints based on the remaining objective functions can usually generate an acceptable solution. In multiobjective problems f will be a vector **f**. Multidisciplinary optimization problems is the hot topic of the day. They can be either a single objective or a multiobjective problem. Essentially, they permit a noticeably larger set of design solutions by permitting the mathematical model to represent design functions from several disciplines. This is again driven by the aerospace design community. To design an aircraft wing to provide the best performance from the point of view of the aerodynamic designer as well as the structural engineer remains a committed goal of this investigation. This is an exciting area, and definitely not for the resource poor. While raising some new issues of coupling, it has not altered the standard techniques of single objective optimization. This book will focus primarily on single objective optimization.

Constraint Functions: As design functions, these will be influenced by the design variables. The format of these functions requires them to be compared to some numerically limiting value that is established by design requirement, or the designer. This value remains constant during the optimization of the problem. A well-described design problem is expected to include several such functions, which can be represented as a vector. The comparison is usually set up using the three standard relational operators: $=$, $\leq$, and $\geq$. Consider our first example. Let $fun1(\mathbf{X})$ represent the function that calculates the volume of the new soda can we are designing. The constraint on the design can be expressed as

$$fun1(\mathbf{X}) = 500 \text{ cm}^3$$

In the second example, let $fun2(\mathbf{X})$ be the function that calculates the deflection of the beam under the applied load. The constraint can be stated as

$$fun2(\mathbf{X}) \leq 1 \text{ mm}$$

The constraint functions can be classified as *equality* constraints [like $fun1(\mathbf{X})$ above] or *inequality* constraints [like $fun2(\mathbf{X})$].

Problems without constraints are termed unconstrained problems. If constraints are present, then meeting them is more paramount than optimization. Constraint satisfaction is necessary before the design established by the current value of the design variables is considered valid and acceptable. If constraints are not satisfied, then there is no solution. A feasible design is one in which all of the constraints are satisfied. An optimal solution is one that has met the design objective. An optimal design must be feasible. The design space enclosed by the constraints is called the feasible domain. Design space is described a few paragraphs below.

Equality Constraints: Equality constraints are mathematically neat and easy to handle. Numerically, they require more effort to satisfy. They are also more restrictive on the design as they limit the region from which the solution can be obtained. The symbol representing equality constraints in the abstract model is h. There may be more than one equality constraint in the design problem. A vector representation for equality constraints is introduced through the following representation. [**H**], $[h_1, h_2, \ldots, h_l]$, and h_k: $k = 1, 2, \ldots, l$ are ways of identifying the equality constraints. The dependence on the design variables **X** is omitted for convenience. Note that the length of the vector is l. An important reason for distinguishing the equality and inequality constraints is that they are manipulated differently in the search for the optimal solution. The number n of design variables in the problem must be greater than the number of equality constraints l for optimization to take place. If n is equal to l, then the problem will be solved without reference to the objective. In mathematical terms the number of equations matches the number of unknowns. If n is less than l, then you have an overdetermined set of relations which could result in an inconsistent problem definition. The set of equality constraints must be linearly independent. Broadly, this

implies that you cannot obtain one of the constraints from elementary arithmetic operations on the remaining constraints. This serves to ensure that the mathematical search for solution will not fail. These techniques are based on methods from linear algebra. In the standard format for optimization problems, the equality constraints are written with a 0 on the right-hand side. This means that the equality constraint in the first example will be expressed as

$$h_1(\mathbf{X}): fun1(\mathbf{X}) - 500 = 0$$

In practical problems, equality constraints are rarely encountered.

Inequality Constraints: Inequality constraints appear more naturally in problem formulation. Inequality constraints also provide more flexibility in design selection. The symbol representing inequality constraints in the abstract model is g. There may be more than one inequality constraint in the design problem. The vector representation for inequality constraints is similar to what we have seen before. Thus, $[\mathbf{G}]$, $[g_1, g_2, \ldots, g_m]$, and g_j: $j = 1, 2, \ldots, m$ are ways of identifying the inequality constraints. m represents the number of inequality constraints. All design functions explicitly or implicitly depend on the design (or independent) variable $\mathbf{X}$. g is used to describe both less than or equal to ($\leq$) and greater than or equal to ($\geq$) constraints. Strictly greater than ($>$) and strictly less than ($<$) are not used much in optimization because the solutions are usually expected to lie at the constraint boundary. In the standard format, all problems are expressed with the $\leq$ relationship. Moreover, the right-hand side of the $\leq$ sign is 0. The inequality constraint from the second example $fun2(\mathbf{X})$ is set up as

$$g_1(\mathbf{X}): fun2(\mathbf{X}) - 1 \leq 0$$

In the case of inequality constraints a distinction is made as to whether the design variables lie on the constraint boundary or in the interior of the region bounded by the constraint. If the set of design variables lie on the boundary of the constraint, mathematically, this expresses the fact that constraint is satisfied with strict equality, that is, $g = 0$. The constraint acts like an equality constraint. In optimization terminology, this particular constraint is referred to as an *active* constraint. If the set of design variables do not lie on the boundary, that is, they lie inside the region of the constraints, they are considered *inactive* constraints. Mathematically, the constraint satisfies the relation $g < 0$. An inequality constraint can therefore be either active or inactive.

Side Constraints: Side constraints are a necessary part of the solution techniques, especially numerical ones. They express the range for the design variables. Each design variable must be bound by numerical values for its lower and upper limit. The designer makes this choice based on his anticipation of an acceptable design.

Design Space: The design space, the space that will be searched for optimal design, is the Euclidean or Cartesian n-dimensional space generated by the n independent design variables $\mathbf{X}$. This is a generalization of the three-dimensional physical space with which we are familiar. For ten design variables, it is a ten-dimensional space. This is not easy to imagine. It is also not easy to express this information through a figure or graph because of the limitation of the three-dimensional world. However, if the design variables are independent, then the n-dimensional considerations are mere extrapolations of the three-dimensional reality. Of course, we cannot geometrically define them though we will be working with the numbers. The side constraints limit the search region, implying that only solutions that lie within a certain region will be acceptable. They define an n-dimensional rectangular region (hypercube) from which the feasible and optimal solutions must be chosen. Later, we will see that the mathematical models in optimization are usually described by nonlinear relationships. The solutions to such problems cannot be analytically predicted as they are typically governed by the underlying numerical technique used to solve them. It is necessary to restrict the solutions to an acceptable region. The side constraints provide a ready mechanism for implementing this limit. Care must be taken that these limits are not imposed overzealously. There must be reasonable space for the numerical techniques to operate.

The Standard Format: The above definitions allow us to assemble the general abstract mathematical model as

$$\text{Minimize} \quad f(x_1, x_2, \ldots, x_n) \tag{1.1}$$

$$\begin{aligned} \text{Subject to:} \quad & h_1(x_1, x_2, \ldots, x_n) = 0 \\ & h_2(x_1, x_2, \ldots, x_n) = 0 \\ & h_l(x_1, x_2, \ldots, x_n) = 0 \end{aligned} \tag{1.2}$$

$$\begin{aligned} & g_1(x_1, x_2, \ldots, x_n) \leq 0 \\ & g_2(x_1, x_2, \ldots, x_n) \leq 0 \\ & g_m(x_1, x_2, \ldots, x_n) \leq 0 \end{aligned} \tag{1.3}$$

$$x_i^l \leq x_i \leq x_i^u, \; i = 1, 2, \ldots, n \tag{1.4}$$

The same problem can be expressed concisely using the following notation:

$$\text{Minimize} \quad f(x_1, x_2, \ldots, x_n) \tag{1.5}$$

$$\text{Subject to:} \quad h_k(x_1, x_2, \ldots, x_n) = 0, \; k = 1, 2, \ldots, l \tag{1.6}$$

$$g_j(x_1, x_2, \ldots, x_n) \leq 0, \; j = 1, 2, \ldots, m \tag{1.7}$$

$$x_i^l \le x_i \le x_i^u, \quad i = 1, 2, \ldots, n \tag{1.8}$$

Exploiting vector notation the mathematical model is

$$\text{Minimize} \quad f(\mathbf{X}), [\mathbf{X}]_n \tag{1.9}$$

$$\text{Subject to:} \quad [h(\mathbf{X})]_l = 0 \tag{1.10}$$

$$[g(\mathbf{X})]_m \le 0 \tag{1.11}$$

$$\mathbf{X}^{\text{low}} \le \mathbf{X} \le \mathbf{X}^{\text{up}} \tag{1.12}$$

The above mathematical model expresses the following standard format of the optimization problem expressed in natural language:

> Minimize the objective function f, subject to l equality constraints, m inequality constraints, with the n design variables lying between prescribed lower and upper limits.

The techniques in this book will apply to the problem described in the above format. To solve any specific design problem it is required to reformulate the problem in the above manner so that the methods can be applied directly. Also in this book, the techniques are developed progressively by considering the standard model with reduced elements. For example, the unconstrained problem will be explored first. The equality constraints are considered next, followed by the inequality constraints, and finally the complete model. This represents a natural progression as prior knowledge is used to develop additional conditions that need to be satisfied by the solution in these instances.

1.1.2 Mathematical Modeling

In this section, the three design problems introduced earlier will be translated to the standard format. The mathematical model will first be identified. The second problem requires information from a course in mechanics and materials. This should be within the purview of most engineering students.

Example 1.1 New consumer research indicates that people like to drink about 0.5 liter of soda pop at a time during the summer months. The fabrication cost of the redesigned soda can is proportional to the surface area and can be estimated at \$1.00 per square meter of the material used. A circular cross section is the most plausible given current tooling available for manufacture. For aesthetic reasons, the height must be at least twice the diameter. Studies indicate that holding comfort requires a diameter between 6 and 9 cm.

Figure 1.1 shows a sketch of the can. In most product designs, particularly in engineering, the model is easier to develop using a figure. The diameter d and the height h are sufficient to describe the soda can. What about the thickness t of the

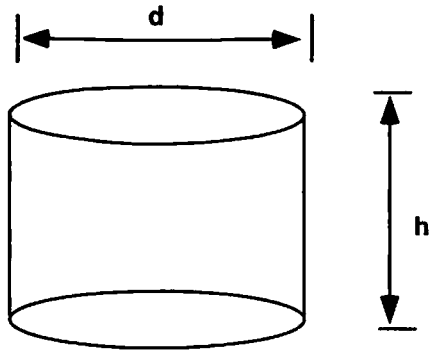

Figure 1.1 Example 1.1—Design of a new beverage can.

material of the can? What are the assumptions for the design problem? One of the assumptions could be that t is small enough to be ignored in the calculation of the volume of soda in the can. Another assumption is that the material required for the can is only the cylindrical surface area. The top and bottom of the can will be fitted with end caps that will provide the mechanism by which the soda can be poured. This is not part of this design problem. In the first attempt at developing the mathematical model we could start out by considering the quantities identified above as design variables:

Design variables: d, h, t

Reviewing the statement of the design problem, one of the parameters is the cost of material per unit area that is given as 1¢ per square centimeter. This is identified as C. During the search for the optimal solution this quantity will be held constant at the given value. Note that if this value changes, then the cost of the can will correspondingly change. This is what we mean by a design parameter. Typically, change in parameters will cause the solution to be recomputed.

Design parameter: C

The design functions will include the computation of the volume enclosed by the can and the surface area of the cylindrical section. The volume in the can is $\pi d^2 h/4$. The surface area is πdh. The aesthetic constraint requires that $h \ge 2d$. The side constraints on the diameter are prescribed in the problem. For completeness the side constraints on the other variables have to be prescribed by the designer. We can formally set up the optimization problem as

$$\text{Minimize} \quad f(d,h,t): \quad C\pi dh \tag{1.13}$$

$$\text{Subject to:} \quad h_1(d,h,t): \quad \pi d^2 h/4 - 500 = 0 \tag{1.14}$$

$$g_1(d,h,t): \quad 2d - h \le 0$$

$$6 \le d \le 9; \quad 5 \le h \le 20; \quad 0.001 \le t \le 0.01 \qquad (1.15)$$

Intuitively, there is some concern with the problem as expressed by Equations (1.13)–(1.15) even though the description is valid. How can the value of the design variable t be established? The variation in t does not affect the design. Changing the value of t does not change f, h_1, or g_1. Hence, it cannot be a design variable. (Note: the variation in t may affect the value of C.) If this were a practical design problem, then the cans have to be designed for impact and stacking strength. In that case, t will probably be a critical design variable. This will require several additional structural constraints in the problem. It could serve as an interesting extension to this problem for homework or project. The optimization problem after dropping t and expressing $[d, h]$ as $[x_1, x_2]$ becomes:

$$\text{Minimize} \quad f(x_1,x_2): C\pi x_1 x_2 \qquad (1.16)$$

$$\text{Subject to:} \quad h_1(x_1,x_2): \pi x_1^2 x_2/4 - 500 = 0 \qquad (1.17)$$

$$g_1(x_1,x_2): 2x_1 - x_2 \le 0 \qquad (1.18)$$

$$6 \le x_1 \le 9; \quad 5 \le x_2 \le 20$$

The problem represented by Equations (1.16)–(1.18) is the mathematical model for the design problem expressed in the standard format. For this problem no extraneous information was needed to set up the problem except for some geometrical relations.

Example 1.2 A cantilever beam needs to be designed to carry a point load F at the end of the beam of length L. The cross section of the beam will be in the shape of the letter I (referred to as an I-beam). The beam should meet prescribed failure criteria. There is also a limit on its deflection. A beam of minimum mass is required to be designed.

Figure 1.2 shows a side view of the beam carrying the load and Figure 1.3 a typical cross section of the beam. The I-shaped cross section is symmetric. The symbols d, t_w, b_f, and t_f correspond to the depth of the beam, thickness of the web, width of the flange, and thickness of the flange, respectively. These quantities are sufficient to define the cross section. A handbook or textbook on strength of materials can aid the development of the design functions. An important assumption for this problem is that we will be working within the elastic limit of the material, where there is a linear relationship between the stress and the strain. All of the variables identified in Figure 1.3 will strongly affect the solution. They are design variables. The applied force F will also directly affect the problem. So will its location L. How about the material? Steel is definitely superior to copper for the beam. Should F, L, and the material properties be included as design variables?

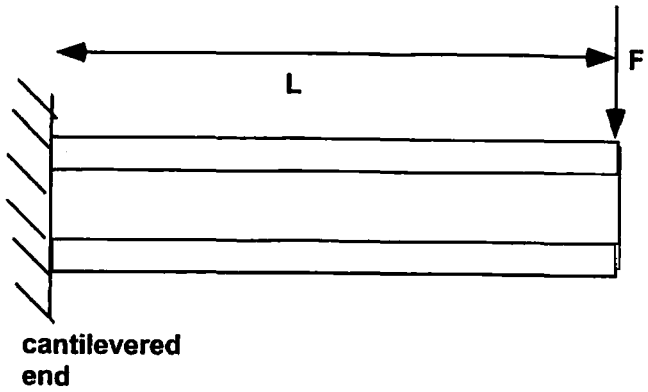

Figure 1.2 Example 1.2—Cantilever beam.

Intuitively, the larger the value of F, the greater is the cross-sectional area necessary to handle it. It can be easily concluded that if F were a design variable, then it should be set at its lower limit, as the mass of the beam will be directly proportional to the area of cross section. No techniques are required for this conclusion. By our definition, F is a good candidate as a parameter rather than as a design variable. Similarly, the larger the value of L, the greater is the moment that F generates about the cantilevered end. This will require an increase in the area of cross-section. Therefore, L is not a good choice for the design variable either as the best choice for it will be its lower limit. To represent material as a design variable, we need to use its structural properties in the formulation of the problem. Typically, a material is characterized by its specific

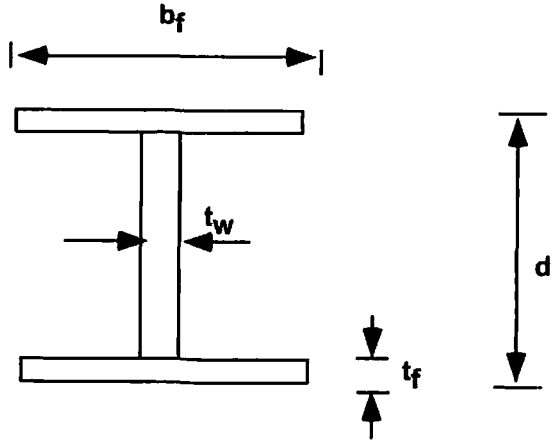

Figure 1.3 Example 1.2—Cross-sectional shape.

weight, γ, its value of the modulus of elasticity, E, its modulus of rigidity, G, its yield limit in tension and compression, σ_{yield}, its yield in shear, τ_{yield}, its ultimate strength in tension, σ_{ult}, and its ultimate strength in shear, τ_{ult}. Optimization techniques identify different values of the design variables. If E is the variable used to represent the material design variable in a problem, then it is possible the solution can require a material that does not exist, that is, a value of E still undiscovered. For all intents and purposes, this is beyond the scope of the designer. Again, material as a *parameter* makes a lot of sense, and the optimization problem should be reinvestigated if F, L, or the material changes. Concluding from this discussion on modeling, the following list can be established for this example:

Design parameters: $F, L, \{\gamma, E, G, \sigma_{yield}, \tau_{yield}, \sigma_{ult}, \tau_{ult}\}$
Design variables: d, t_w, b_f, t_f

In the development of the mathematical model, standard technical definitions are used. Much of the information can be obtained from a mechanical engineering handbook. The first design function is the weight of the beam. This is the product of γ, L, and A_c, where A_c is the area of cross section. The maximum stress due to bending σ_{bend} can be calculated as the product of $FLd/2I_c$, where I_c is the moment of inertia about the centroid of the cross section along the axis parallel to the flange. The maximum shear stress in the cross section is expressed as $FQ_c/I_c t_w$, where Q_c is first moment of area about the centroid parallel to the flange. The maximum deflection (δ_L) of the beam will be at the end of the beam calculated by the expression $FL^3/3EI_c$. A_c of the cross section of the beam is $2b_f t_f + t_w(d - 2t_f)$. The first moment of area Q_c can be calculated as $0.5b_f t_f(d - t_f) + 0.5t_w(0.5d - t_f)^2$. Finally, the moment of inertia I_c is obtained from $(b_f d^3/12) - ((b_f - t_w)(d - 2t_f)^3/12)$. In the following, for convenience, we will continue to use A_c, Q_c, and I_c instead of detailing their dependence on the design values. Associating x_1 with d, x_2 with t_w, x_3 with b_f, and x_4 with t_f, so that $\mathbf{X} = [x_1, x_2, x_3, x_4]$, the problem in standard format is

$$\text{Minimize} \quad f(\mathbf{X}): \gamma L A_c \tag{1.19}$$

$$\begin{aligned} \text{Subject to:} \quad & g_1(\mathbf{X}): FLx_1/2I_c - \sigma_{yield} \le 0 \\ & g_2(\mathbf{X}): FQ_c/I_c x_2 - \tau_{yield} \le 0 \\ & g_3(\mathbf{X}): FL^3/3EI_c - \delta_{max} \le 0 \\ & 0.01 \le x_1 \le 0.25; \quad 0.001 \le x_2 \le 0.05; \\ & 0.01 \le x_3 \le 0.25; \quad 0.001 \le x_4 \le 0.05 \end{aligned} \tag{1.20}$$

To solve this problem, F must be prescribed (10,000 N), L must be given (3 m). Material must be selected (steel : $\gamma = 7860$ kg/m^3; $\sigma_{yield} = 250$ E+06 N/m^2; $\tau_{yield} = 145$ E+06 N/m^2). Also the maximum deflection is prescribed ($\delta_{max} = 0.005$ m).

This is an example with four design variables, three inequality constraints, and no equality constraints. Other versions of this problem can easily be formulated. It can be reduced to a two-variable problem. Standard failure criteria with respect to combined stresses or principal stresses can be formulated. If the cantilevered end is bolted, then additional design functions regarding bolt failure need to be examined.

Example 1.3 MyPC Company has decided to invest \$12 million in acquiring several new Component Placement Machines to manufacture different kinds of motherboards for a new generation of personal computers. Three models of these machines are under consideration. Total number of operators available is 100 because of the local labor market. A floor space constraint needs to be satisfied because of the different dimensions of these machines. Additional information relating to each of the machines is given in Table 1.1. The company wishes to determine how many of each kind is appropriate to maximize the number of boards manufactured per day.

It is difficult to use a figure in this problem to set up our mathematical model. The number of machines of each model needs to be determined. This will serve as our design variables. Let x_1 represent the number of Component Placement Machines of Model A. Similarly, x_2 will be associated with Model B, and x_3 with Model C.

Design variables: x_1, x_2, x_3

The information in Table 1.1 is used to set up the design functions in terms of the design variables. An assumption is made that all machines are run for three shifts. The cost of acquisition of the machines is the sum of the cost per machine multiplied by the number of machines (g_1). The machines must satisfy the floor space constraint (g_2) which is defined without details in Eq. (1.23). The constraint on the number of operators is three times the sum of the product of the number of machines of each model and the operators per shift (g_3). The utilization of each machine is the number of boards per hour times the number of hours the machine operates per day. The optimization problem can be assembled in the following form:

$$\text{Maximize} \quad f(\mathbf{X}): 18 \times 55 \times x_1 + 18 \times 50 \times x_2 + 21 \times 50 \times x_3 \tag{1.21}$$

or

$$\text{Minimize} \quad f(\mathbf{X}): -18 \times 55 \times x_1 - 18 \times 50 \times x_2 - 21 \times 50 \times x_3$$

$$\text{Subject to:} \quad g_1(\mathbf{X}): 400{,}000x_1 + 600{,}000x_2 + 700{,}000x_3 \le 12{,}000{,}000 \tag{1.22}$$

$$g_2(\mathbf{X}): 3x_1 - x_2 + x_3 \le 30 \tag{1.23}$$

$$g_3(\mathbf{X}): 3x_1 + 6x_2 + 6x_3 \le 100 \tag{1.24}$$

$$x_1 \ge 0; \quad x_2 \ge 0; \quad x_3 \ge 0$$

Equations (1.21)–(1.24) express the mathematical model of the problem. Note that in this problem there is no product being designed. Here a strategy for placing order for the number of machines is being determined. Equation (1.21) illustrates the translation of the maximization objective into an equivalent minimizing one. The inequality constraints in Equations (1.22)–(1.24) are different from the previous two examples. Here the right-hand side of the ≤ operator is nonzero. Similarly, the side constraints are bound on the lower side only. This is done deliberately. There is a significant difference between this problem and the previous two. All of the design functions here are linear. This problem is classified as a linear programming problem. The solutions to these types of problems are very different than the solution to the first two examples, which are recognized as nonlinear programming problems.

1.1.3 Nature of Solution

The three examples in the previous section are used to discuss the nature of solutions to the optimization problem. Examples 1.1 and 1.2 describe the nonlinear programming problem, which forms the major part of this book. Example 1.3 is a linear programming problem, which is very significant in decision sciences, but rare in product design. Its inclusion here, while necessary for completeness, is also important for understanding contemporary optimization techniques for nonlinear programming problems. Examples 1.1 and 1.2 are dealt with first. Between the two, we notice that Example 1.1 is quite simple relative to Example 1.2. Second, the four variables in Example 1.2 make it difficult to use illustration to establish some of the discussion.

SOLUTION TO EXAMPLE 1.1 The simplest determination of nonlinearity is through a graphical representation of the design functions involved in the problem. This can be done easily for one or two variables. If the function does not plot as a straight line or a plane, then it is nonlinear. Figure 1.4 shows a three-dimensional plot of Example 1.1. The figure is obtained using MATLAB. Chapter 2 will provide detailed instructions for drawing such plots for graphical optimization. The three-dimensional representation of Example 1.1 in Figure 1.4 does not really enhance our understanding of the problem or the solution. Figure 1.5 is an alternate representation of the problem. This is the one encouraged in this book. The horizontal axis represents the diameter (x_1), and the vertical axis the height (x_2). In Figure 1.5 the *equality constraint* is marked appropriately as h1. Any pair of values on this line will give a volume of 500 cm^3. The *inequality constraint* is shown as g1. The pair of values on this line exactly satisfies the aesthetic requirement. Hash lines on the side of the inequality constraint establish the disallowed region for the design variables. The constraints are drawn thicker for emphasis. The scaled *objective function* is represented through several labeled contours. Each contour is associated with a fixed value of the objective function and these values are shown on the figure. The range of the two axes establishes the *side constraints*. The objective function f and the equality constraint h_1 are nonlinear since they do not plot as straight lines. Referring to Equations (1.16) and (1.17), this is substantiated by the products of the two unknowns (design variables) in

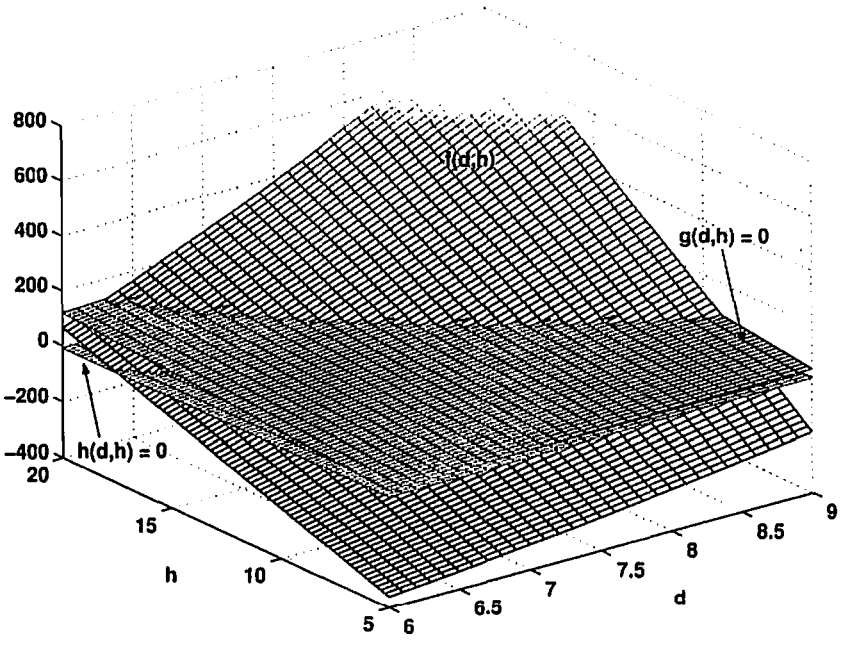

Figure 1.4 Graphical representation of Example 1.1.

both of these functions. The equality constraint g_1 is linear. In Equation (1.18) this is evident because the design variables appear by themselves without being raised to a power other than 1. As we develop the techniques for applying optimization, this distinction is important to keep in mind. It is also significant that graphical representation is typically restricted to two variables. For three variables, we need a fourth dimension to resolve the information and three-dimensional contour plots are not easy to illustrate. The power of imagination is necessary to overcome these hurdles.

The problem represented in Figure 1.5 provides an opportunity to identify the graphical solution. First, the feasible region is identified. In Figure 1.5 this is the equality constraint above the inequality constraint. Both *feasible* and *optimal* must occur from this region. Any solution from this region is an *acceptable* (*feasible*) solution. There are usually a large number of such solutions—*infinite* solutions. If optimization is involved, then these choices must be reduced to the best one with respect to some criteria—the *objective*. In this problem/figure, the smallest value of f is desired. The lowest value of f is just less than the contour value of 308. It is at the intersection of the two constraints. While the value of f needs to be calculated, the optimal values of the design variables, read from the figure, are about 6.75 and 13.5, respectively. Another significant item of information obtained from Figure 1.5 is that g_1 is an *active* constraint. While there are infinite feasible solutions to the problem, the optimal solution is *unique*.

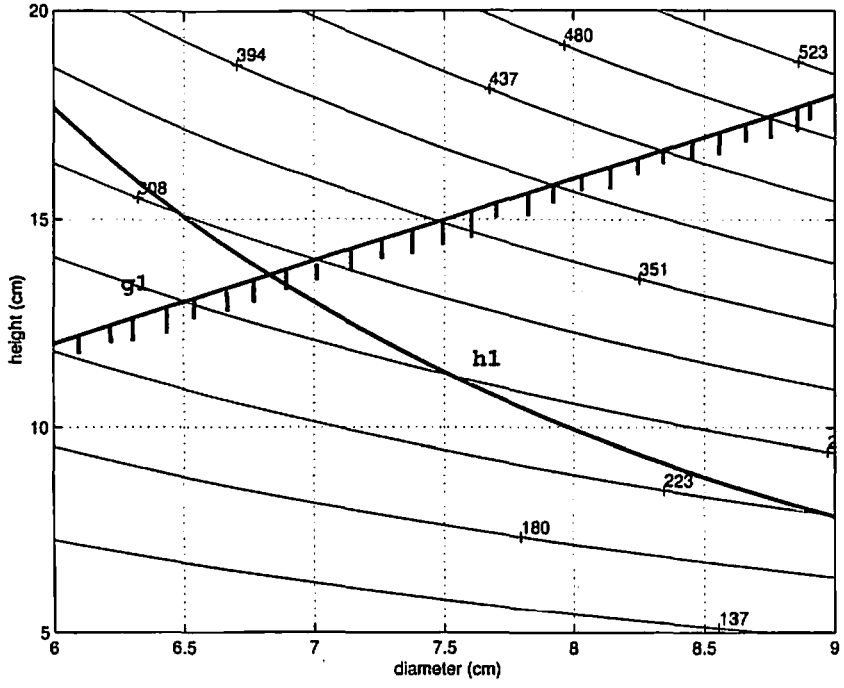

Figure 1.5 Contour plot of Example 1.1.

SOLUTION TO EXAMPLE 1.3 In contrast to Example 1.1, all of the relationships in the mathematical model, expressed by Equations (1.21)–(1.24), are linear. The word *equation* is generally used to describe the equivalence of two quantities on either side of the = sign. In optimization, we come across mostly *inequalities*. The word *equation* will be used to include both of these situations. In Example 1.1, the nature of solution was explained using Figure 1.5. In Example 1.3, the presence of three design variables denies this approach. Instead of designing a new problem, suppose that the company president has decided to buy five machines of the third type. The mathematical model, with two design variables x_1 and x_2, is reconstructed using the information that x_3 is now a parameter with the value of 5. Substituting for x_3 in the model for Example 1.3, and cleaning up the first constraint:

$$\text{Maximize} \quad f(\mathbf{X}): 990 \times x_1 + 900 \times x_2 + 5250 \tag{1.25}$$

$$\text{Subject to:} \quad g_1(\mathbf{X}): 0.4x_1 + 0.6x_2 \leq 8.5 \tag{1.26}$$

$$g_2(\mathbf{X}): 3x_1 - x_2 \leq 25 \tag{1.27}$$

$$g_3(\mathbf{X}): 3x_1 + 6x_2 \leq 70 \tag{1.28}$$

$$x_1 \geq 0; \quad x_2 \geq 0$$

An interesting observation in the above set of equations is that there are only two design variables but three constraints. This is a valid problem. If the constraints were equality constraints, then this would not be an acceptable problem definition as it would violate the relationship between the number of variables and the number of equality constraints. Therefore, the number of *inequality* constraints are not related to the number of design variables used. This fact also applies to nonlinear constraints. Figure 1.6 is a scaled plot of the functions (1.25)–(1.28). Choosing the first quadrant satisfies the side constraints. The linearity of all of the functions is indicated by the straight-line plots on the figure. Again, several contours of f are shown to aid the recognition of the solution. All of the inequalities are drawn with hash marks indicating the nonfeasible side of the constraint. The feasible region is enclosed by the two axes along with constraints g_2 and g_3. The objective is to increase the value of f (which is indicated by the contour levels) without leaving the feasible region. The solution can be

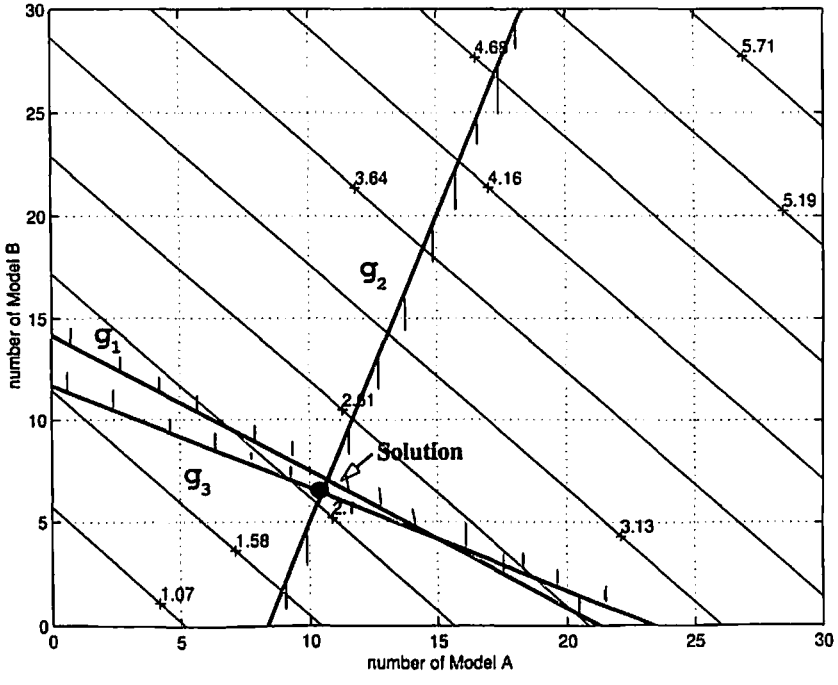

Figure 1.6 Contour plot of modified Example 1.3.

spotted at the intersection of constraints g2 and g3. The values of the design variables at the solution can be read off from the plot. It is not over yet. Before proceeding further it is necessary to acknowledge that a solution of 6.5 machines for Model B is unacceptable. The solution must be adjusted to the nearest integer values. In actual practice, this adjustment has to be made without violating any of the constraints. The need for *integer variables* is an important consideration in design. Variables that are required to have only integer values belong to the type of variables called *discrete variables*. This book mostly considers *continuous variables*. Almost all of the mathematics an engineer encounters, especially academically, belong to the domain of continuous variables. The assumption of continuity provides very fast and efficient techniques for the search of constrained optimum solutions, as we will develop in this text. Many real-life designs, especially those that use off-the-shelf materials and components, determine discrete models. Discrete programming and integer programming are disciplines that address these models. The nature of the search methods, that is, methods used for searching the optimal solution, in discrete programming is very different from that in continuous programming. Many of these methods are based on the scanning and replacement of the solutions through exhaustive search of the design space in the design region. In this book, discrete problems are typically handled as if they were continuous problems. The conversion to the discrete solution is the final part of the design effort and is usually performed by the designer outside of any optimization technique we develop in this book. Chapter 8 discusses discrete optimization with examples.

Getting back to the *linear programming* problem just discussed, it should also be noted that only the boundary of the feasible region will affect the solution. Here is a way that will make this apparent. Note that the objective function contours are straight lines. A different objective function will be displayed by parallel lines with a different slope. Imagine these lines on the figure instead of the actual objective. Note that the solution always appears to lie at the intersection of the constraints, that is, the *corners* of the design space. Solution can never come from inside of the feasible region, unlike in the *nonlinear programming* problem, which accommodates solution on the boundary and from within the feasible region. Recognizing the nature of solutions is important in developing a search procedure. For example, in linear programming problems, a technique employed to search for the optimal design need only search the boundaries, particularly the intersection of the boundaries, whereas in nonlinear problems, the search procedure cannot ignore interior values.

1.1.4 Characteristics of the Search Procedure

The search for the optimal solution will depend on the nature of the problem being solved. For nonlinear problems, it must consider the fact that solution could lie inside the feasible region. Example 1.1 through Figure 1.5 will provide the context for discussing the search procedure for nonlinear problems. Similarly, the characteristics

of the search process for linear problems will be explored using Example 1.3 and Figure 1.6.

Nonlinear Problems: Except for a few classes of problems, solutions to nonlinear problems are obtained through *numerical analysis*. Through computer code, numerical analysis becomes numerical techniques. The methods or techniques for finding the solution to optimization problems are called *search methods*. In applied mathematics and numerical techniques these are referred to as *iterative techniques*. Implicitly this means several *tries* will be necessary before the solution can be obtained. Also implied is that each try or search is executed in a consistent manner. Information from the previous iteration is utilized in the computation of values in the present sequence. This consistent manner or process by which the search is carried out and the solution determined is called an *algorithm*. This term also refers to the translation of the particular search procedure into an ordered sequence of *step-by-step* actions. The advantage is derived mostly in the conversion of these steps, through some computer language, into code that will execute on the computer. Developing the algorithm and translating it into code that will execute in MATLAB will be a standard approach in this book.

The search process is started typically by trying to guess the *initial design* solution. The designer can base this selection on his experience. On several occasions the success of optimization may hinge on his ability to choose a good initial solution. The search method, even when used consistently, responds differently to the particular problem that is being solved. The degree and type of nonlinearity may frequently cause the method to fail, assuming there are no inconsistencies in problem formulation. Often, the number of design variables, the number of constraints, as well as a bad initial guess play a part in unexpected behavior under standard implementation of the method. This has led to the creation of several different techniques for optimization. Practical experience has also suggested that a *class of problems* respond well to certain algorithms.

The methods for the search, or iterative methods, are currently the only way to implement numerical solutions to nonlinear problems, irrespective of the discipline in which these problems appear. These problems are plentiful in design optimization, optimal control, structural and fluid mechanics, quantum physics, and astronomy. While optimization techniques are applied in all of the noted disciplines, in traditional design optimization the problems are mainly characterized by nonlinear algebraic equations. In optimal control they can be nonlinear dynamic systems, which are described by nonlinear differential equations. In structural mechanics there is the possibility of nonlinear integral equations. Computational fluid dynamics deals mostly with nonlinear partial differential equations. Standard implementations of these search methods proceed by attempting to get closer to the solution with every iteration. This is referred to as *converging to the solution*. In optimization techniques this could also mean determining a feasible solution.

Figure 1.7 is the graphical representation of the solution of Example 1.1 (same as Figure 1.5) embellished with several iterations of a *hypothetical* search method. The

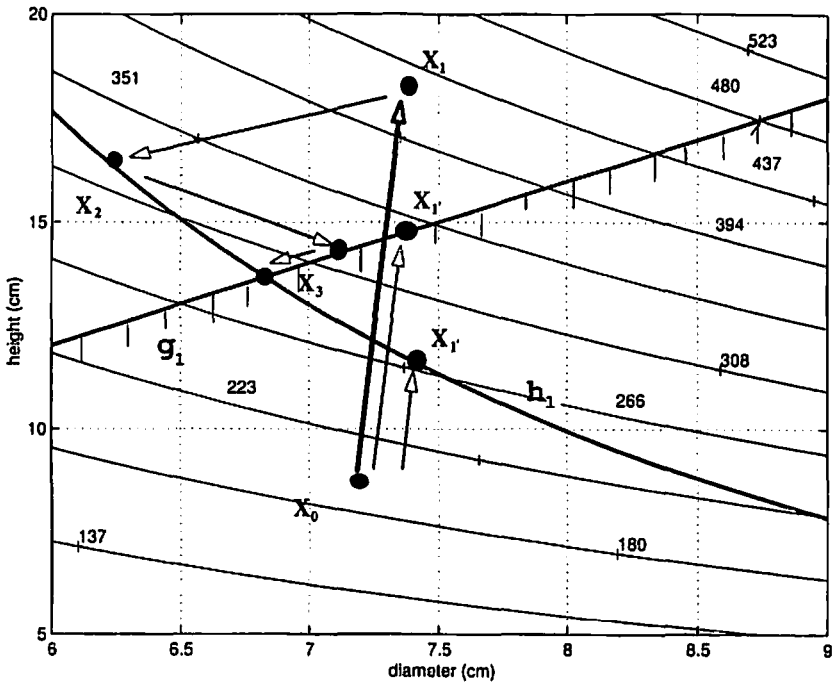

Figure 1.7 A typical search method.

initial starting guess is the point marked X_0. Remember that when solving the optimization problem, such a figure *does not exist*. If it did, then we can easily obtain the solution by inspection. This statement is especially true if there are more than two variables. In the event there are two variables, there is no reason not to obtain a *graphical solution* (Chapter 2). The methods in optimization rely on the direct extrapolation of the concepts and ideas developed for two variables to any number of design variables. That is the reason we will use two variables to discuss and design most of the algorithms and methods. Moreover, using two variables, the concepts can also be reinforced using geometry. The design at the point X_0 is not feasible. It is not useful to discuss whether it is an optimal solution. At least one more iteration is necessary to obtain the solution. Consider the next solution at X_1. Most methods in optimization will attempt to move to X_1 by first identifying a *search direction*, S_1. This is a *vector* (same dimension as design variable vector X) pointing from X_0 to X_1. The actual distance from X_0 to X_1, call it X_0X_1, is some multiple of S_1, say αS_1. X_1 once again is not feasible although it is an improvement on X_0 since it satisfies the inequality constraint g_1 (lying in the interior of the constraint). Alternately, using the same search direction S_1, another technique/method may choose for its next point, $X_{1'}$,

at a distance βS_1. This point satisfies the constraint h_1. A third strategy would be to choose $X_{1'}$. This lies on the constraint boundary of g_1. It is to be noted that all three choices are not feasible. Which point is chosen is decided by the rules of a particular algorithm. Assume point X_1 is the next point in this discussion.

Point X_1 now becomes the new point X_0. A new search direction S_2 is determined. Point X_2 is located. Another sequence of calculations locates X_3. The next iteration yields the solution. The essence of updating the design during successive iteration lies in two major computations, the *search direction* S_i and *stepsize* α. The change in the design vector, defined as ΔX, which represents the vector difference, for example, $X_2 - X_1$, is obtained as αS_i. For each iteration the following sequence of activity can be identified:

Step 0: Choose X_0
Step 1: Identify S
Determine α
$\Delta X = \alpha S$
$X_{new} = X_0 + \Delta X$
Set $X_0 \leftarrow X_{new}$
Go To Step 0

The above is the basic structure of a typical optimization algorithm. A complete one will indicate the manner in which S and α are computed. It will include tests, which will determine if the solution has been reached (convergence), or if the procedure needs to be suspended or restarted. The different techniques to be explored in this book mostly differ in the way S is established. Take a look at Figure 1.7 again. At X_0 there are infinite choices for the search direction. To reach the solution in reasonable time an acceptable algorithm will try not to search along directions that do not improve the current design. There are many ways to accomplish this strategy.

Linear Programming Problem: It was previously mentioned that the solution to these problems would be on the boundary of the feasible region (also called the *feasible domain*). In Figure 1.6, the axes and some of the constraints determine this region. The latter is distinguished also by the points of intersection of the constraints among themselves, the origin, and the intersection of the *binding* (active) constraints with the axes. These are *vertices* or *corners* of the quadrilateral. Essentially the design improves by moving from one of these vertices to the next one through improving feasible designs. To do that you have to start at a feasible corner. This will be explored more fully in the chapter on linear programming. It is to be emphasized that only with two variables can we actually see the geometric illustration of the technique, that is, the selection of vertices as the solution converges.

This part of Chapter 1 has dealt with the introduction of the optimization problem. While it was approached from an engineering design perspective, the mathematical model, in abstract terms, is not sensitive to any particular discipline. A few design rules with respect to specific examples were examined. The standard mathematical model for optimization problems was established. Two broad classes of problems,

linear and nonlinear, were introduced. The geometrical characteristics of the solution for these problems were shown through the graphical description of the problems. An overview of the search techniques for obtaining the solution to these problems was also illustrated. It was also mentioned that the solutions were to be obtained *numerically*, primarily through digital computation.

The numerical methods used in this area are iterative methods. Numerical solutions involve two components: first, an algorithm that will establish the iterative set of calculations, and second, the translation of the algorithm into computer codes, using a *programming language*. This finished code is referred to as *software*. Until recently in engineering education as well as practice, the computer language of choice was predominantly FORTRAN with a little Pascal and C. The computational environment in which it was delivered and practiced was the *mainframe* computing systems. Today, the scene is very different. Individual personal computers (PCs), running Windows, Macintosh, LINUX, or other flavors of UNIX, operating systems are now the means for technical computing. They possess more power than the old mainframe computers and they are inexpensive. The power of the graphical user interface (GUI), the way the user interacts with the computer and the software, has revolutionized the creation of software today. New *programming paradigms*, especially *object oriented programming*, have led to new and improved ways to develop software. Today, engineering software is created by such general programming languages as C, C++, Java, Pascal, and Visual Basic. Many software vendors develop their own extensions to the standard language, providing a complete environment for software development by the user. This makes it very convenient for users who are not necessarily programmers to get their job done in a reasonably short time.

One such vendor is MathWorks, with their flagship product MATLAB along with a complementary collection of toolboxes that deliver specific discipline-oriented information. In this book, MATLAB is used exclusively for numerical support. *No prior familiarity* with MATLAB is expected, though familiarity with the PC is required. Like all higher-level languages, effectiveness with this new programming resource comes with effort, frequency of use, original work, and a consistent manner of application. By the end of this book, it is hoped the reader will be very familiar with the use of MATLAB to develop programs for optimization. The experience gained from this book will be substantial, and sufficient for the use of MATLAB in other settings. Nevertheless, there is a large part of MATLAB that will not be used. It should be possible to explore that part with the experience gained from this book. All of the code in this book is developed only after identifying a need for its implementation. New *commands* will be recognized and explained (commented on) when they appear for the first time. The code will be kept simple, as this book is also a means for understanding MATLAB through writing numerical techniques for optimization. The author welcomes suggestions from new users, and those familiar with MATLAB who have considerable experience in programming, for ways to make the learning process more effective. The next part of this chapter deals with MATLAB, establishing its appropriateness for this book, as well as getting started in its use. The reader is required to have access to MATLAB to be able to use this book effectively.

1.2 INTRODUCTION TO MATLAB

MATLAB is introduced by MathWorks Inc., as the language for technical computing. Borrowing from the description in any one of its manuals, MATLAB integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. The typical uses for MATLAB include

- Math and computation
- Algorithm development
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including GUI building

The use of MATLAB has exploded recently and so have its features. This continuous enhancement makes MATLAB an ideal vehicle for exploring problems in all disciplines that manipulate mathematical content. This ability is multiplied by application-specific solutions through toolboxes. To learn more about the latest MATLAB, its features, the toolboxes, collection of user-supported archives, information about Usenet activities, other forums, information about books, and so on, visit MATLAB at http://www.mathworks.com.

1.2.1 Why MATLAB?

MATLAB is a standard tool for introductory and advanced courses in mathematics, engineering, and science in many universities around the world. In industry, it is a tool of choice for research, development, and analysis. In this book, MATLAB's basic array element is exploited to manipulate vectors and matrices that are natural to numerical techniques. In the next chapter, its powerful visualization features are used for graphical optimization. In the rest of this book, the interactive features and built-in support of MATLAB are utilized to translate algorithms into functioning code, in a fraction of the time needed in other languages like C or FORTRAN. Toward the end, we will introduce the use of its Optimization Toolbox.

Most books on optimization provide excellent coverage on the techniques and algorithms, and some also provide a printed version of the code. The code is required to be used in mainframe or legacy environments that must support a compiler in the language the code is written, which is predominantly FORTRAN. Each student then translates the printed code into a form suitable for computation. The next step is to compile the code, debug compilation errors, run the program, and troubleshoot execution errors before useful results can be obtained. An important consideration is that the student is expected to be conversant with the programming language. While FORTRAN was formally taught to engineers several years ago, this is no longer true.

Many electrical engineering departments have started to embrace C and C++. PCs are great for the use of ready-made applications, especially those with GUI, but its use for programming in FORTRAN has yet to take off.

MATLAB suffers very few of these disadvantages. Its code is compact and it is intuitive to learn. To make problems work requires few lines of code compared to traditional languages. Instead of programming from the ground up, standard MATLAB pieces of code can be threaded together to implement the new algorithm. It avoids the standard separate compilation and link sequence by using an interpreter. Another important reason is code portability. The code written for the PC version of MATLAB does not have to be changed for the Macintosh or the UNIX systems. This will not be true if system-dependent resources are used in the code. As an illustration of MATLAB's learning agility, both MATLAB and applied optimization are covered in this book, something very difficult to accomplish with, say, FORTRAN and optimization, or C++ and optimization. Furthermore, MATLAB, like C or FORTRAN, is a globally relevant product.

1.2.2 MATLAB Installation Issues

This book does not care if you are running MATLAB on an individual PC or Mac, a networked PC or Mac, or individual or networked Unix workstations. MATLAB takes care of this issue. The book assumes you have a working and appropriately licensed copy of MATLAB in the machine in front of you. On your personal machine, for installing MATLAB, follow the instructions that accompanied the box. Typically the system administrator handles network installations.

This book uses MATLAB Version 5.2 (until Chapter 4) and Version 5.3 (the rest). Version 6 has been shipping since the fall of 2000. MATLAB version 5.0 and above should work. If MATLAB has changed any built-in commands in newer versions than the one used in the book, it should provide information on the availability of newer version of the commands (or throw an error, a rare case). The MATLAB code and examples were created on an Intel Pentium Pro 200 running the Windows NT 4 operating system. The earlier codes were also tested on a DEC Alpha server running OSF1 V4. The MATLAB version on the server was 5.1.

In organizations and institutions MATLAB software is usually licensed on a subscription basis. Users therefore have the latest version of the software in these situations. In such a case the new version (at the time the book is first published), will be Version 6 (or Release R 12). This book is based on Version 5 (Release R11). The code and the programming instructions in the book should execute without problem in the new version. In the new Release R12, on the PC platform, the default appearance of MATLAB is a composite of several windows presenting different pieces of information. This is shown in Figure 1.8. This can be customized through the menu bar by *View* → *Desktop Layout* → *Command Window Only*, which also corresponds to the previous release. It is shown in Figure 1.9. This view is simple and less distracting for getting familiar with MATLAB. Note that the MATLAB command prompt is >>. This window delivers standard operating system commands for saving, printing, opening files, and so on, through menus and a *command bar*. In order to

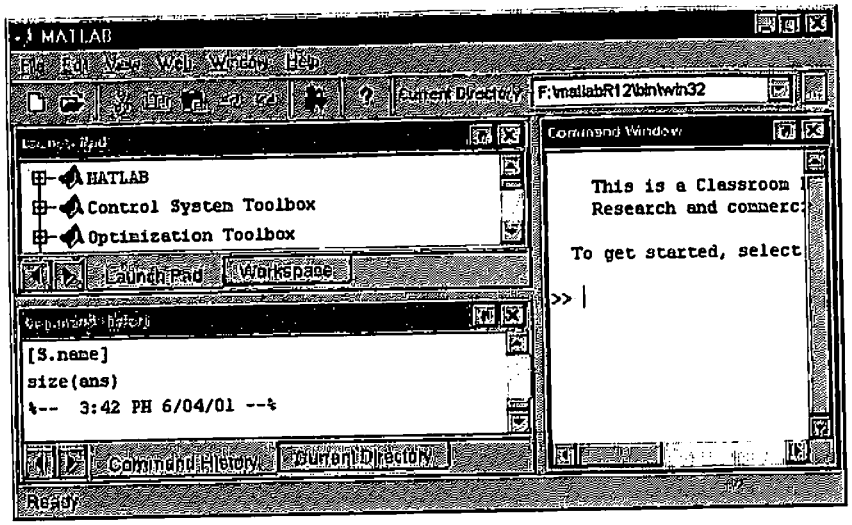

Figure 1.8 MATLAB opening view.

create files (or code) another graphical window is needed. This is the MATLAB M-file *editor/debugger* window and is shown in Figure 1.10. The editor uses color for enhancing code readability. Each of these windows spawn additional dialog boxes (if necessary) for setting features during the MATLAB session, like the *path browser* and *workspace browser*.

While this is not so much as an installation issue, it would be appropriate to discuss the way this book plans to write and execute MATLAB code. Generally, there are two ways to work with MATLAB: *interactively* and through *scripts*. In interactive mode, the commands are entered and executed one at a time in the MATLAB command window. Once the session is over and MATLAB exited, there is no information retained on the set of commands that were used and what worked (unless a choice was made to save the session). Large sequences of commands are usually difficult to keep track of in an interactive session. Another option is to store the sequence of commands in a text file (ASCII), and execute the file in MATLAB. This is called a *script m-file*. Most MATLAB files are called *m-files*. The *.m* is a file extension usually reserved for MATLAB. Any code changes are made through the editor, the changes to the file saved, and the file executed in MATLAB again. Executing MATLAB through script files is predominantly followed in this book. MATLAB allows you to revert to interactive mode any time.

1.2.3 Using MATLAB the First Time

MATLAB will be used interactively in this section. The author very strongly recommends a hands-on approach to understanding MATLAB. This means typing the codes yourself. It also includes understanding the syntax errors, and debugging and

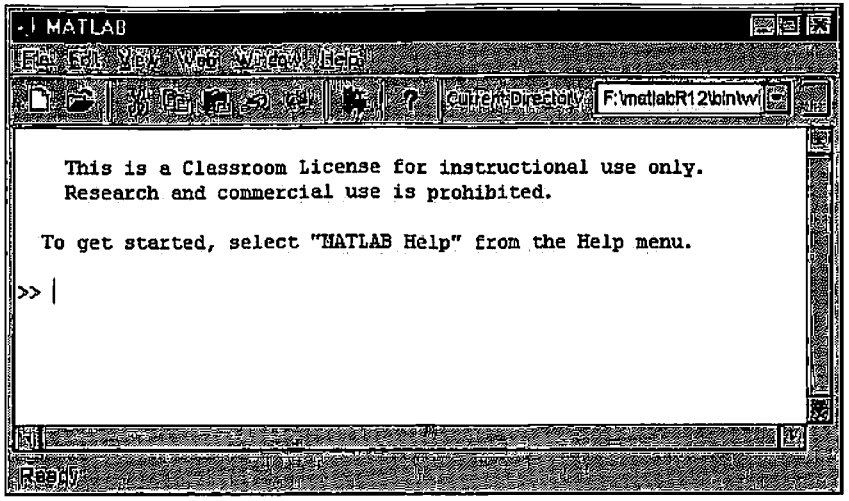

Figure 1.9 MATLAB Command window.

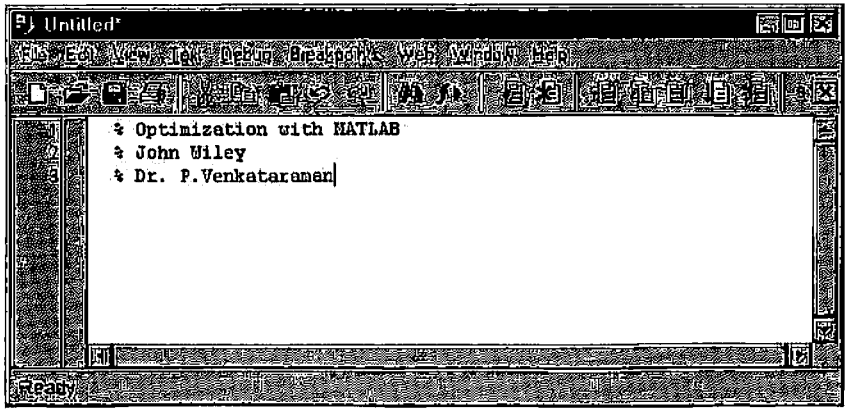

Figure 1.10 Matlab Editor/Debugger window.

rectifying the code. The author is not aware of a single example where programming was learned without the reader typing code. In this book, MATLAB code segments are courier font with the bold style used for emphasizing commands or other pieces of information. Anything else is recommendations, suggestions, or exercises. The placeholder for command is in italic.

Before We Start: The following punctuation and special characters are worth noting:

`%`	All text after the % sign is considered a *comm*ent. MATLAB ignores anything to the right of the % sign. *Comments* are used liberally in this book to indicate why's and what's of command or code usage. Do not overlook them. You do not have to code them in, however, probably two weeks later you will not remember why you developed that particular piece of code.
`>>`	This is the default MATLAB prompt.
`;`	A semicolon at the end of a line prevents MATLAB from echoing the information you enter on the screen.
`,`	A comma will cause the information to echo. Default is a comma.
`...`	A succession of three periods at the end of the line informs MATLAB that code will continue to the next line. You cannot split a variable name across two lines.You cannot continue a comment on another line.
`^c`	You can stop MATLAB execution and get back the command prompt by typing ^c (Ctrl-C)—by holding down "Ctrl" and "c" together.
`>> help command_name`	Will display information on the various ways the command can be used. This is the quickest way to use help.
`>> helpwin`	Opens a help text window that provides more information on all of the MATLAB resources installed on your system.
`>> helpdesk`	Provides help using a browser window.

For those of you programming for the first time, the equal to sign (=) has a very special meaning in all programming languages, including MATLAB. It is called the *assignment operator*. The variable on the left-hand side of the sign is assigned the value of the right-hand side. The actual *equal to* operation in MATLAB is usually accomplished by a double equal to sign (==).

Some Additional Features

- MATLAB is *case sensitive*. An *a* is different than *A*. This is difficult for persons who are used to FORTRAN. All built-in MATLAB commands are in lowercase.
- MATLAB does not need a *type definition* or a *dimension* statement to introduce variables. It automatically creates one on first encounter in code. The type is assigned in context.
- Variable names start with a letter and contain up to 31 characters (only letters, digits, and underscore).
- MATLAB uses some built-in variable names. Avoid using built-in variable names.
- All numbers are stored internally using the long format specified by IEEE floating-point standard. These numbers have roughly a precision of 16 decimal digits. They range roughly between 10E-308 and 10E+308. However, they are *displayed* differently depending on the context.
- MATLAB uses conventional decimal notation for numbers using decimal points and leading signs optionally, for example, 1, –9, +9.0, 0.001, 99.9999.
- Scientific notation is expressed with the letter *e*, for example, 2.0e-03, 1.07e23, –1.732e+03.
- Imaginary numbers use either *i* or *j* as a suffix, for example, 1i, -3.14j, 3e5i.

Operators: The following are the arithmetic operators in MATLAB.

+	Addition (when adding matrices/arrays size must match)
–	Subtraction (same as above)
*	Multiplication (the subscripts of arrays must be consistent when multiplying them)
/	Division
^	Power
'	Complex conjugate transpose (also array transpose)

In the case of arrays, each of these operators can be used with a *period* prefixed to the operator, for example, (.*) or (.^) or (./). This has a special meaning in MATLAB. It implies element-by-element operation. It is useful for quick computation. It has no

relevance in mathematics or anywhere else. We will use it a lot in the next chapter for generating data for graphical optimization.

An Interactive Session: Start MATLAB. On the PC press the **Start** button. Press the **Program** button. Press the MATLAB Program Group. Press **MATLAB**. The MATLAB Command window opens up and after some introductory messages there will be a MATLAB prompt indicated by *double forward arrow*. This is an indication that MATLAB is ready to accept your commands. On the networked stations, typing **matlab** at the window prompt should do the same. Note that the MATLAB program must be in the path as defined in the **.login** or **.cshrc** file. Ask the lab assistant for help. These are standard UNIX procedures. Hit return at the end of the line or before the comment. The comments relate to what is being typed and what is displayed in the MATLAB command window. Feel free to try your own variations. In fact, to understand and reinforce the commands, it is recommended that you make up your own examples often.

```
>> a = 1.0; b = 2.0; c = 3.0, d = 4.0; e = 5.0
>> % why did only c and e echo on the screen?
>> who    % lists all the variables in the workspace
>> a      % gives the value stored in a
>> A = 1.5;      % variable A
>> a, A          % case matters
>> one = a; two = b; three = c;
>> % assigning values to new variables
>> four = d; five = e; six = pi; % value of pi available
>> A1 = [a b c ; d e f] % A1 is a 2 by 3 matrix
>>   % space or comma separates columns
>>   % semi-colon separates rows

>> A1(2,2)       % accesses the Matrix element on the
>>               % second row and second column
>> size(A1)      % gives you the size of the matrix
>>               % (row, columns)
>> AA1 = size(A1)      % What should happen here?
>>       % from previous statement the size of A1
>>       % contains two numbers organized as a row
>>       % matrix. This is assigned to AA1
>> size(AA1)     % AA1 is a one by two matrix
>> A1'     % this transposes the matrix A1
>> B1 = A1'      % the transpose of matrix A1
>>   % is assigned to B1. B1 is a three by two matrix
>> C1 = A1 * B1 % Since A1 and B1 are matrices this
>>       % is a matrix multiplication
>>       % Should this multiplication be allowed?
>>       % consider matrix multiplication in C or
```

```
>>        % Fortran
>>        % the power of MATLAB derives from its
>>        % ability to handle matrices like numbers
>> C2 = B1 * A1 % How about this?
>> C1 * C2      % What about this?
>>        % read the error message
>>        % it is quite informative
>> D1 = [1 2]'  % D1 is a column vector
>> C3 = [C1 D1] % C1 is augmented by an extra column
>> C3 = [C3 ; C2(3,:)] % Note = is an assignment
>>   % means do the right hand side and overwrite the
>>   % old information in C3 with the result
>>   % of the right
>>   % hand side calculation
>>   % On the right you are adding a row to current
>>   % matrix C3. This row has the value of the third
>>   % row of C2 - Notice the procedure of
>>   % identifying the third row. The colon
>>   % represents all the columns
>> C4 = C2 * C3       % permissible multiplication
>>   % Note the presence of a scaling factor
>>   % in the displayed output
>> C5 = C2 .* C3      % seems to multiply!
>>   % Is there a difference between C4 and C5?
>>   % The .* represents the product of each element
>>   % of C2 multiplied with the corresponding
>>   % element of C3
>> C6 = inverse(C2)   % find the inverse of C2
>>   % apparently inverse is not a command in MATLAB
>>   % if command name is known it is easy to obtain
>>   % help
>> lookfor inverse % this command will find all files
>>   % where it comes across the word "inverse" in
>>   % the initial comment lines
>>   % The command we need appears to be INV which
>>   % says Inverse of a Matrix
>>   % The actual command is in lower case. To find
>>   % out how to use it - Now
>> help inv     % shows how to use the command
>> inv(C2)      % inverse of C2
>> for i = 1:20
    f(i) = i^2;
    end
>>   % This is an example of a for loop
```



```
>>   % the index ranges from 1 to 20 in steps of
>>   % 1(default)
>>   % the loop is terminated with "end"
>>   % the prompt does not appear until "end" is
>>   % entered
>> plot(sin(0.01*f),cos(0.03*f))
>> xlabel('sin(0.01*f)') % strings appear in single
>>   % quotes
>> ylabel('cos(0.03*f)')
>> legend ('Example')
>> title ('A Plot Example')
>> grid
>>   % The previous set of commands will create plot
>>   % label axes, write a legend, title and grid the
>>   % plot
>> exit   % finished with MATLAB
```

This completes the first session with MATLAB. Additional commands and features will be encountered throughout the book. In this session, it is evident that MATLAB allows easy manipulation of matrices, definitely in relation to other programming languages. Plotting is not difficult either. These advantages are quite substantial in the subject of optimization. In the next session, we will use the editor to accomplish the same through scripts. This session introduced

- MATLAB Command window and Workspace
- Variable assignment
- Basic matrix operations
- Accessing rows and columns
- Suppressing echoes
- **who, inverse** commands
- .* multiplication
- Basic plotting commands

1.2.4 Using the Editor

In this section, we will use the editor to create and run a MATLAB a script file. Normally, the editor is used to generate two kinds of MATLAB files. These files are termed *script files* and *function files*. Although both of these files contain MATLAB commands like the ones we have used, the second type of files needs to be organized in a specified format. Both file types should have the extension **.m**. Although these files are ASCII text files, the generic **.m** should be used because MATLAB searches for this extension. This extension is unique to MATLAB. The script file contains a list of MATLAB commands that are executed in sequence. This is different from the interactive session of the previous section where

MATLAB responded to each command immediately. The script file is more useful when there are many commands that need to be executed to accomplish some objective, like running an optimization technique. It is important to remember that MATLAB allows you to switch back to interactive mode at any time by just typing commands in the workspace window like in the previous section.

MATLAB provides an editor for creating these files on the PC platform. This editor uses color to identify MATLAB statements and elements. It provides the current values of the variables (after they are available in the workspace) when the mouse is over the variable name in the editor. There are two ways to access the editor through the MATLAB Command window on the PC. Start MATLAB. This will open a MATLAB Command or Workspace window. In this window the editor can be started by using the menu or the toolbar. On the **File** menu, click on **New** and choose **M-file**. Alternately, click on the leftmost icon on the toolbar (the tooltip reads **New File**). The icon for the editor can also be placed on the desktop, in which case the editor can be started by double-clicking the icon. In this event, a MATLAB Command window will not be opened. The editor provides its own window for entering the script statements.

At this point we are ready to use the editor. Make sure you read the comments and understand them, as these procedures will be used often. Using the editor implies we will be working with m-files.

The commands are the same as in the interactive session except there is no MATLAB prompt prefixing the expressions. To execute these commands you will have to save them to a file. Let us call the file ***script1.m***. The **.m** extension need not be typed if you are using the MATLAB editor. You can save the file using the **Save** or **Save As** command from most editors. It is important to know the full path to this file. Let us assume the path for the file is C:\Opt_book\Ch1\script1.m. Note that the path here is specified as a PC path description. The reason we need this information is to inform MATLAB where to find the file. We do this in the MATLAB Command window.

This implies we should have the MATLAB Command window open. On PCs we start MATLAB through the icons. In this book we will accomplish most tasks by typing in the Command window or through programming. In many instances there are alternate ways using menu items or the toolbar. This is left to the reader to experiment and discover. In the MATLAB Command window use the **addpath** command to inform MATLAB of the location of the file:

```
>> addpath C:\Opt_book\Ch1\
```

The script that will be created and saved in ***script1.m*** can be run by typing (note that the extension is omitted)

```
>> script1
```

To understand and associate with the programming concepts embedded in the script, particularly for persons with limited programming experience, it is recommended to run the script after a block of statements have been written rather than typing the file

in its entirety. Prior to running the script you will have to save the script each time so that the changes in the file are recorded and the changes are current. Another recommendation is to deliberately *mistype* some statements and attempt to debug the error generated by the MATLAB debugger during execution.

Creating the Script M-file (The following will be typed/saved in a file.)

```
%  example of using script
A1 = [1 2 3];
A2 = [4 5 6];
% the commands not terminated with semi-colon will
% display information on the screen
A = [A1; A2]
B = [A1' A2']
C = A*B

% now re-create the matrix and perform matrix
% multiplication as in other programming languages
% example of for loop
for i = 1 : 3    % variable i ranges from 1 to 3 in
                 % steps of 1 (default)
        a1(1,i) = i;
end              % loops must be closed with end
a1

for i = 6:-1:4  % note loop is decremented
    a2(1,i-3) = i;   % filling vector from rear
end
a2

% creating matrix A and B (called AA and BB here)
for i = 1:3
    AA(1,i) = a1(1,i); % assign a1 to AA.
    AA(2,i) = a2(1,i);
    BB(i,1) = a1(1,i);
    BB(i,2) = a2(1,i);
end
% the same can be accomplished by AA (1,:) = a1
% without the for loop
AA        % print the value of AA in the window
BB
who       % list all the variables in the workspace
% consider code for Matrix multiplication
% which Matlab does so easily
```

```
% multiply two matrices (column of first matrix must
% match row of second matrix)
szAA = size(AA) %      size of AA
szBB = size(BB);
if (szAA(1,2) == szBB(1,1))
% only in column of AA match the rows of BB
  for i = 1:szAA(1,1)
   for j = 1:szBB(1,2)
     CC(i,j) = 0.0; % initialize value to zero
     for k = 1:szAA(1,2)
          CC(i,j) = CC(i,j) + AA(i,k)*BB(k,j);
     end     % k - loop
   end    % j - loop
  end  % i - loop
end % if - loop
CC
% Note the power of MATLAB derives from its ability to
% handle matrices very easily
% this completes the script session
```

Save the above file (***script1.m***). Add the directory to the MATLAB path as indicated before. Run the script file by typing *script1* at the command prompt. The commands should all execute and you should finally see the MATLAB prompt in the Command window.

Note: You can always revert to the interactive mode by directly entering commands in the MATLAB window after the prompt. In the Command window:

```
>> who
>> clear C  % discards the variable C from the workspace
>>   % use with caution. Values cannot be recovered
>> help clear
>> exit
```

This session illustrated:

- Use of the editor
- Creating a script
- Running a script
- Error debugging (recommended activity)
- Programming concepts
- Loop constructs , **if** and **for** loops
- Loop variable and increments
- Array access

- **Clear** statement

1.2.5 Creating a Code Snippet

In this section, we will examine the other type of m-file which is called the ***function m-file***. For those familiar with other programming languages like C, Java, or FORTRAN, these files represent functions or subroutines. They are primarily used to handle some specific types of calculations. They also provide a way for the modular development of code as well as code reuse. These code modules are used by being called or referred in other sections of the code, say through a script file we looked at earlier. The code that calls the function m-file is called the calling program/code. The essential parameters in developing the function m-file are (1) what input is necessary for the calculations, (2) what specific calculations must take place, and (3) what information must be returned to the calling program. MATLAB requires the structure of the function m-file to follow a prescribed format.

We will use the editor to develop a function m-file that will perform a polynomial curvefit. It requires a set of *xy data*, representing a curve that needs to be fit, together with the *order of the polynomial* to be fit. This exercise is called curve fitting. In Chapter 6 such a problem will be identified as a problem in *unconstrained optimization.* For now, the calculations necessary to accomplish the exercise are considered known. It involves solving a linear equation with the *normal matrix* and a right-hand vector obtained using the data points. The *output* from the m-file will be the coefficients representing the polynomial.

Before we start to develop the code, the first line of this file must be *formatted as specified* by MATLAB. In the first line, the first word starting from the first column is the word *function.* It is followed by the set of *return parameters* (**returnval**). Next, the name (**mypolyfit**) of the function with the parameters passed to the function within parentheses (*XY,N*). The file must be saved as name.m (***mypolyfit.m***). The comments between the first line and the first executable statement will appear if you type *help name* (**help mypolyfit**) in the Command window. The reason for the name mypolyfit.m is that MATLAB has a built-in function **polyfit.** Open the editor to create the file containing the following information:

```
function returnval = mypolyfit(XY, N)
%
% These comments will appear when the user types
% help mypolyfit in the Command window
% This space is intended to inform the user how to
% interact with the program, what it does
% what are the input and output parameters
%   Least square error fit of polynomial of order N
%   xy - Data found in XY
%   returns the vector of coefficients starting from
%   the constant term
```

```
%
for i = 1:N+1
  a(i) = 0.0; % initialize the coefficient to zero
end

sz = size(XY);
NDATA = sz(1,1); % number of data points - rows of
% xy matrix
if NDATA == 0
  fprintf('There is no data to fit');
  returnval = a;       % zero value returned
  return;        % return back to calling program
end

if NDATA < 2*N
  fprintf('Too few data points for fit')
  returnval = a;
  return
end

% The processing starts here.
% The coefficients are obtained as solution to the
% Linear Algebra problem [A][c] = [b]
% Matrix [A] is the Normal Matrix which contains
% the sum of the powers of the independent variable
% [A] is symmetric

for i = 1:N+1;
  b(i) = 0.0;
  for ll = 1:NDATA; % loop over all data points
                    % variable is "ll" (el)(el)
   b(i) = b(i) + XY(ll,2)*XY(ll,1)^(i-1);
  end      % loop ll

  for j = 1:N+1;
   if j >= i % calculating upper diagonal terms
     power = (i-1) + (j-1);
     A(i,j) = 0.0      % initialize
     for k = 1:NDATA;  % sum over data points
      A(i,j) = A(i,j) + XY(k,1)^power;
     end  % k loop
   end     % close if statement
  A(j,i) = A(i,j) % exploiting Matrix symmetry
   end     % end j loop
```



```
end       % end i loop
%  if the x-points are distinct then inverse is not a
%  problem

returnval = inv(A)*b';
```

Save the file as ***mypolyfit.m***. To use the function we will need to create some *xy* data and then call the mypolyfit function. Start MATLAB in the directory that ***mypolyfit.m*** resides in, or add the directory to the path. In the MATLAB Command window: The following code is typed in the command window.

```
>> for i = 1: 20;
    x = i/20;
    XY1(i,1) = x;
    XY1(i,2) = 2 + 3.0*x - x*x - 3*x^3;
end

>> coeff = mypolyfit(XY1,3)
>> % a cubic polynomial was deliberately created to
>> % check the results. You should get back
>> % the coefficients you used to generate the curve
>> % this is a good test of the program
>> % Let us create another example

>> XY2(:,1) = XY1(:,1); % same first column
>> XY2(:,2) = 2.0 + exp(XY2(:,1))
>> % note the power of Matlab - a vector fill
>> % a new set of xy-data

>> coeff1 = mypolyfit(XY2, 3) % cubic poly. again
>> help mypolyfit
>> % you should see the statements you set up
>> % In the next exercise we will create a script
>> % file that will run a program for polynomial
>> % curve-fitting we will save XY2 so that it can
>> % be used again

>> save C:\Opt_book\Ch1\XY2.dat -ascii -double
>> % this will save the file as an ascii text file
>> % with double precision values
```

This concludes the exercise where a code snippet was written to calculate the coefficients of the polynomial used to fit a curve to some *xy* data. The type of file is the

function m-file. It needs to be used in a certain way. The code was tested using a cubic polynomial. Nonpolynomial data were also tested. The data were saved for later use.

1.2.6 Creating a Program

In this section a program that will read *xy* data, curvefit the data using a polynomial, and compare the original and fitted data graphically will be developed. The data will be read using a User Interface (UI) window and a dialog box. The code will be developed as a script file. This gives us an opportunity to revisit most of the code we used earlier for reinforcement. We will also use the function m-file created in the previous exercise.

There are several ways for you or the users to interact with the code you develop. The basic method is to prompt users for information at the prompt in the Command window. This is the quickest. This is probably what you will use when developing the code. Once the code has been tested, depending on usefulness it might be relevant to consider using more sophisticated custom elements like input boxes and file selection boxes. This book will continue to use these elements throughout as appropriate. While the input elements used in this code are new commands, the rest of the program will mostly use commands that have been introduced earlier. In sequential order the events in this program are: (1) to read the *xy* data saved earlier using a **file selection** box, (2) to read the order of fit using an **input dialog** box, (3) to use the **mypolyfit** function developed in the last section to obtain the coefficients, (4) to obtain the coordinates of the fitted curve, (5) to graphically compare the original and fitted data, and (6) to report on the fitted accuracy on the figure itself. The new script file will be called ***prog_pfit.m***.

Start the editor to create the file called ***prog_pfit.m***. In it type

```
% Program for fitting a polynomial curve to xy data
% from
%         Applied Optimization using Matlab
%         Dr. P.Venkataraman
%
%         Chapter 1, Section 1.2.6
% The program looks for a file with two column ascii
% data with extension .dat. The order of the curve is
% obtained from user. The original and fitted data
% are compared with relevant information displayed on
% the same figure. The program demonstrates the use
% of the file selection box, an input dialog box,
% creating special text strings and displaying them

[file,path] =uigetfile('*.dat','All Files', 200,200);
% uigetfile opens a file selection box
% checkout help uigetfile
% the string variable file will hold the filename
```



```
% the string variable path will have the path
% information the default directory pointed will
% differ depending on the platform

if isstr(file) % if a file is selected
  loadpathfile = ['load ',path file];
  % loadpathfile is a string variable concatenated
  % with three strings "load ", path and file
  % note the space after load is important

  eval(loadpathfile);
  % evaluates the string enclosed - which includes the
  % Matlab command load. This will import the xy-data
  % the data will be available in the workspace as a
  % variable with the same name as the filename
  % without the extension (this assumes you selected
  % the xy-data using the file selection box)

  newname = strrep(file,'.dat','');
  % newname is a string variable which contains the
  % string file stripped of the .dat extension. This
  % is a string replacement command

  x = eval(newname); % assigns the imported data to x
  % just for convenience. The above step is not
  % necessary

  NDATA = length(x(:,1));  % number of data points
  clear path loadpathfile newname
  % get rid of these variables to recover memory
end

% Note: if a file is not selected, nothing is being
% done
% Use of an input dialog box to get the order
% of polynomial to be fitted
PROMPT = {'Enter the Order of the Curve'};
% PROMPT is a string Array with one element
% note the curly brackets
TITLE = 'Order of the Polynomial to be Fitted';
% a string variable
LINENO = 1;  % a data variable

getval = inputdlg(PROMPT, TITLE, LINENO);
```

```
% the input dialog captures the user input in getval
% getval is a string Array
% check help inputdlg for more information

no = str2num(getval{1,1});
% the string is converted to a number- the order

clear PROMPT TITLE LINENO  % deleting variables

% call function mypolyfit and obtain the coefficients
coeff = mypolyfit(x,no);

% generate the fitted curve and obtain the squared
% error
err2 = 0.0;
for i = 1:NDATA  % for each data point
  for j = 1:no + 1
   a(1,j) = x(i,1)^(j-1);
  end
  y(i) = a*coeff;  % the data for the fitted curve
  err2 = err2 + (x(i,2) -y(i))*(x(i,2)-y(i));
  % the square error
end

% plotting
plot (x(:,1),x(:,2),'ro',x(:,1),y,'b-');
% original data are red o's
% fitted data is blue solid line
xlabel('x');
ylabel('y');
strorder = setstr(num2str(no));
% convert the order of curve to a string
% same as getval if you have not cleared it
% setstr assigns the string to strorder
titlestr = ['Polynomial curvefit or order', ...
strorder,  'of file', file);
% the three dots at the end are continuation marks
% the title will have the order and the file name
title(titlestr)
legend('original data', 'fitted data');

errstr1 = num2str(err2);
errstr2 = ['squared error = ', errstr1];
gtext(errstr2);
```



```
% this places the string errstr2 which is obtained
% by combining the string 'squared error' with
% the string representing the value of the error,
% wherever the mouse is clicked on the plot.
% moving the mouse over the figure you should
% see location cross-hairs
clear strorder titlestr errstr1 errstr2 a y x i j
clear NDATA no coeff XY2 file err2 getval
grid

% This finishes the exercise
```

Run the program by first running MATLAB in the directory where these files are, or adding the path to locate the files. At the command prompt type **prog_pfit**. The program should execute requiring user input through the tile selection box, input dialog, and finally displaying Figure 1.11. The appearance may be slightly different depending on the platform MATLAB is being run.

This finishes the MATLAB section of the chapter. The section has introduced MATLAB in a robust manner. A broad range of programming experience has been initiated in this chapter. All new commands have been identified with a brief explanation in the comments. It is important that you use the opportunity to type in the code yourself. That is the only way the use of MATLAB will become familiar. The practice also will lead to fewer syntax errors. The writing of code will significantly improve your ability to debug and troubleshoot. While this chapter employed a

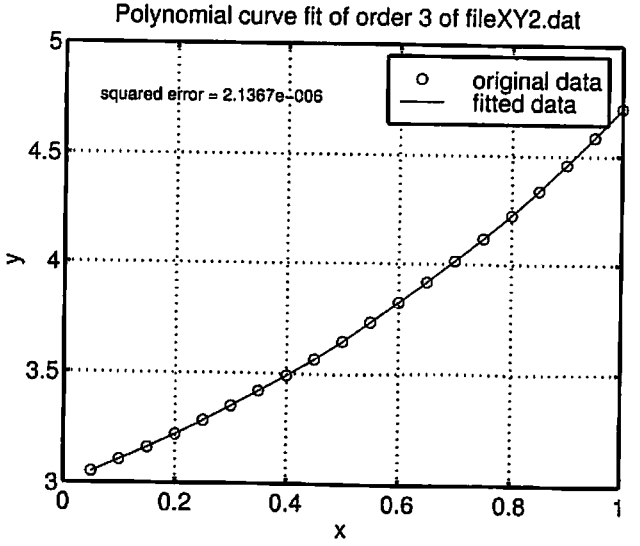

Figure 1.11 Original and fitted data.

separate section on the use of MATLAB out of necessity, subsequent chapters are characterized by a complete integration of the use of MATLAB during the discussion of optimization.

PROBLEMS

1.1 Identify several possible optimization problems related to an automobile. For each problem identify all the disciplines that will help establish the mathematical model.

1.2 Identify several possible optimization problems related to an aircraft. For each problem identify all the disciplines that will help establish the mathematical model.

1.3 Identify several possible optimization problems related to a ship. For each problem identify all the disciplines that will help establish the mathematical model.

1.4 Identify several possible optimization problems related to a microsystem used for control. For each problem identify all the disciplines that will help establish the mathematical model.

1.5 Define a problem with respect to your investment in the stock market. Describe the nature of the mathematical model.

1.6 Define the problem and establish the mathematical model for the I-beam holding up an independent single-family home.

1.7 Define the problem and identify the mathematical model for an optimum overhanging traffic light.

1.8 Define the problem and identify a mathematical model for scheduling and optimization of the daily routine activity.

1.9 Define the problem for a laminar flow in a pipe for maximum heat transfer driven given a specific pump.

1.10 Define a chemical engineering problem to mix various mixtures of limited availability to make specified compounds to meet specified demands.

2

GRAPHICAL OPTIMIZATION

This book includes a large number of examples with two variables. Two-variable problems can be displayed graphically and the solution obtained by inspection. Along with the usefulness of obtaining the solution without applying mathematical conditions, the graphical representation of the problem provides an opportunity to explore the geometry of many of the numerical techniques we examine later. This is necessary for intuitively understanding the algorithm and its progress toward the solution. In practical situations which usually involve over two variables, there is no opportunity for neat graphical correlation because of the limitations of graphical representation. It will be necessary to review the numbers to decide on convergence or the lack thereof. What happens with many variables is an extension of the geometric features that are observed in problems involving two variables. A good imagination is an essential tool for these problems with many variables.

Contour plots provide the best graphical representation of the optimization problem in two variables. The points on any contour (or curve) have the same value of the function. Several software packages are available to create and display these plots. MATLAB, Mathematica, Maple, and Mathcad are among the general-purpose software packages that can draw contour plots. In this book, we will use MATLAB for graphical optimization.

2.1 PROBLEM DEFINITION

The standard format for optimization problems was established in Chapter 1. It is reintroduced here for convenience:

$$\text{Minimize} \quad f(x_1, x_2, \ldots, x_n) \tag{2.1}$$

$$\begin{aligned} \text{Subject to:} \quad & h_1(x_1, x_2, \ldots, x_n) = 0 \\ & h_2(x_1, x_2, \ldots, x_n) = 0 \\ & h_l(x_1, x_2, \ldots, x_n) = 0 \end{aligned} \tag{2.2}$$

$$\begin{aligned} & g_1(x_1, x_2, \ldots, x_n) \le 0 \\ & g_2(x_1, x_2, \ldots, x_n) \le 0 \\ & g_m(x_1, x_2, \ldots, x_n) \le 0 \end{aligned} \tag{2.3}$$

$$x_i^l \le x_i \le x_i^u, \quad i = 1, 2, \ldots, n \tag{2.4}$$

In this chapter, while adhering to the format, the necessity for zero on the right-hand side is relaxed. This is being done for convenience and to aid comprehension. The right-hand side can also have numerical values other than zero. The first example chosen for illustration is a simple one using elementary functions whose graphical nature is well known. This simple example will permit examination of the MATLAB code that will generate the curves and the solution. It will also allow us to define the format for the display of solution to the graphical optimization problem.

2.1.1 Example 2.1

The first example, Example 2.1, will have two equality constraints and two inequality constraints:

$$\text{Minimize} \quad f(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 2)^2 \tag{2.5}$$

$$\text{Subject to:} \quad h_1(x_1, x_2): 2x_1 + x_2 = 8 \tag{2.6a}$$

$$h_2(x_1, x_2): (x_1 - 1)^2 + (x_2 - 4)^2 = 4 \tag{2.6b}$$

$$g_1(x_1, x_2): x_1 + x_2 \le 7 \tag{2.7a}$$

$$g_{2(x_1, x_2)}: x_1 - 0.25x_2^2 \le 0 \tag{2.7b}$$

$$0 \le x_1 \le 10; \quad 0 \le x_2 \le 10 \tag{2.8}$$

In the above definition, we have two straight lines, two circles, and a parabola. Note that two equality constraints and two variables imply that there is no scope for optimization. The problem will be determined by the equality constraints provided the two constraints are linearly independent, which is true in this example. This example was created to help understand the code in MATLAB that will be used to draw graphical solutions in this book.

Figure 2.1 illustrates the graphical solution to this problem. The figure also displays additional information, related to the inequality constraints, placed on it after the principal graphic information was generated and displayed.

2.1.2 Format for the Graphical Display

The graphical solution to Example 2.1, as seen in Figure 2.1, is generated using MATLAB [1, 2] except for the identification of the inequality constraints. All inequality constraints are distinguished by hash marks. The hashed side indicates the infeasible region. In MATLAB 5.2, there is no feature to insert these marks through programming. An add-on program called *Matdraw* was used to create the hash lines in the figure. It can be downloaded from the Mathworks site. Beginning with Version 5.3 there is a figure (*plotedit*) editor that will allow you to insert additional graphic elements to the figure. The student can also pencil in these hash lines after obtaining the printout. It is to be noted that the graphical solution is incomplete if the inequality constraints are not distinguished, or if the feasible region has not been established in some manner.

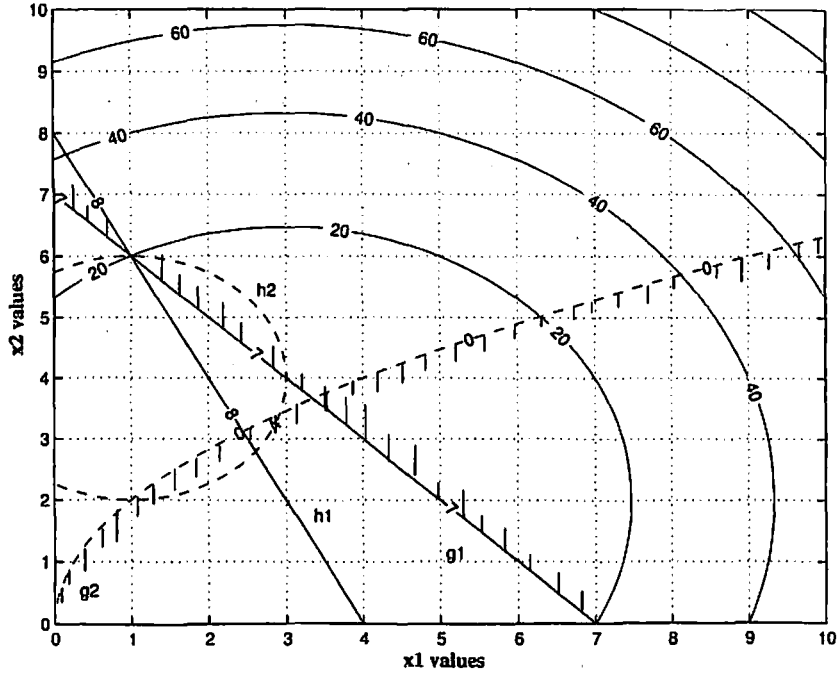

Figure 2.1 Graphical solution for Example 2.1.

In the graphical display of solutions, only the objective function is drawn for several contours so that the direction for the minimum can be identified. Each equality constraint is drawn as a single curve for the numerical value on the right-hand side. They should be identified. Each inequality constraint is similarly drawn for a value on the right-hand side. They need to be identified too. The hash marks are drawn/penciled in on the figure or a printout of the figure. The extent of the design region is established and the solution identified on the figure. The relevant region can be zoomed for better accuracy. The data for the plot are obtained using MATLAB's matrix operations after establishing the plotting mesh. This exploits MATLAB's natural speed for matrix operations.

2.2 GRAPHICAL SOLUTION

MATLAB possess a powerful visualization engine that permits the solution of the two-variable optimization problem by inspection. There are three ways to take advantage of the graphic features of MATLAB. The first is the use of MATLAB's high-level graphing routines for data visualization. This will be the primary way to solve graphical optimization problems in this book. This will also be the way to incorporate graphical exploration of numerical techniques in the book. For more precise control over the display of data, MATLAB allows user interaction through programming using an object-oriented system identified in MATLAB as *Handle Graphics*. The third use of the MATLAB graphics engine is to use the Handle Graphics system to develop a Graphical User Interface (GUI) for the program or m-file. This creates a facility for sophisticated user interaction. Most of the plotting needs can be met by the high-level graphics functions available in MATLAB.

2.2.1 MATLAB High-Level Graphics Functions

There are three useful windows during a typical MATLAB session. The first is the MATLAB Command window through which MATLAB receives instruction and displays alphanumeric information. The second window is the text-editor window where m-files are coded. The third is the Figure window where the graphic elements are displayed. There can be more than one figure window. The figure window is the target of the high-level graphics functions.

The graphics functions in MATLAB allow you to plot in 2D or 3D. They allow contour plots in 2D and 3D, mesh and surface plots, bar, area, pie charts, histograms, animation, and gradient plots. Subplots can also be displayed using these functions. In addition they permit operation with images and 3D modeling. They allow basic control of the appearance of the plot through color, line style, and markers, axis ranges, and aspect ratio of the graph. They permit annotation of the graph in several ways. Some of these functions will be used in the next section when we develop the

m-file for the first example. The following introduces some more useful information about the use of MATLAB's high-level graphics functions.

The two main graphical elements that are typically controlled using the high-level graphics functions are the figure and the axes. Using Handle Graphics you can control most of the other graphical elements, which include elements used in the GUI. These other elements are typically children of the *axes* or *figure*. The *figure* function or command creates a figure window with a number starting at one, or will create a new figure window incrementing the window count by one. Normally all graphics functions are targeted to the current figure window, which is selected by clicking it with the mouse or executing the command figure (number), where number is the number of the figure window that will have the focus. All commands are issued in command window. Graphics functions or commands will automatically create a window if none exists.

Invoke *help commandname* at the MATLAB prompt to know more about the functions or commands. The word *handle* appears on many platforms and in many applications, particularly those that deal with graphics. They are widely prevalent in object-oriented programming practice. MATLAB's visualization system is object oriented. Most graphical elements are considered as objects. The "handle" in MATLAB is a system/software created number that can identify the specific graphic object. If this handle is available, then properties of the object such as line size, marker type, color, and so on can be viewed, set, or reset if necessary. In MATLAB, Handle Graphics is the way to customize the graphical elements. In high-level graphics functions this is used in a limited way. In this chapter, we will use it in a minimal way to change the characteristics of some of the graphical elements on the figure.

To understand the concept of "handles" we will run the following code interactively. It deals with creating a plot, which was introduced in Chapter 1. The boldface comments in the code segment indicate the new features that are being emphasized.

Start MATLAB and *interactively* perform the following:

```
>> x = 0:pi/40:2*pi;    % create x vector
>> y = x.*sin(x);       % create y vector
>> plot(x,y,'b-');      % plot y vs x in blue color
>> grid;
>> h = plot(x,y,'b-')   % h is the handle to the plot
>> % a new plot is overwritten in the same figure window
>> % a numerical value is assigned to h
>> % We can use the variable name h or its value
>> % to refer to the plot again
>>
>> set(h,'LineWidth',2); % this should make your plot
>>                       % thicker blue
>> set(h,'LineWidth',3,'LineStyle',':','Color','r')
```

```
>> % The handle is used to refer to the object
>> % whose property is being changed
>> % Usually Property information occurs in pairs of
>> % property-name/property-value
>> % property-value can be a text string or number
>>
>> get(gca)  % this will list the property of the axes
>> % of the current plot. Note there are a significant
>> % amount of properties you can change to customize
>> % the appearance of the plot
>>
>> set(gca,'ytick',[-5,-2.5,0,2.5,5])
>> % you have reset the ytick marks on the graph
>> set(gca,'FontName','Arial','FontWeight',...
>>   'bold','FontSize',14)
>>
>> % Changes the font used for marking the axes
>> set(gca,'Xcolor','blue')
>> % changes the x-axis to blue
>> % concludes the demonstration of handles
```

From the session above it is evident that to fine-tune the plot you create, you need to first identify the object handle. You use the handle to access the object property. You change a property by setting its value. We will see more graphics function as we obtain the graphical solution to Example 2.1 in the following section.

2.2.2 Example 2.1—Graphical Solution

Figure 2.1 is the graphical representation of Example 2.1. The range for the plots matches the side constraints for the problem. The intersection of the two equality constraints identifies two possible solutions to the problem, approximately (2.6, 2.8) and (1, 6). The inequality constraint g_2 makes the point (2.6, 2.8) unacceptable. Point (1, 6) is acceptable with respect to both constraints. The solution is therefore at (1, 6). While the objective function was not used to determine the solution, contours of the objective function are drawn indicating the direction for the minimum of the objective function. The solution can be identified by inspection of the assembled plots without reference to the terminology or the techniques of optimization. Recall, however, that this is only possible with one or two design variables in the problem.

The code for this example involves six m-files: a script m-file that will include the MATLAB statements to create and display the data, and five m-files—one for each function involved in Example 2.1. We begin with the script m-file—*Ex2_1.m*.

File: **Ex2_1.m**

```
% Chapter 2: Optimization with MATLAB
% Dr. P. Venkataraman
% Example 2_1 (Sec 2.1-2.2)
%
% graphical solution using MATLAB (two design variables)
% the following script should allow the display
% of graphical solution
%
% Minimize f(x1,x2) = (x1-3)**2 + (x2-2)**2
%          h1(x1,x2) : 2x1 + x2 = 8
%          h2(x1,x2) : (x1-1)^2 + (x2-4)^2 = 4
%          g1(x1,x2) : x1 + x2 <=  7
%          g1(x1,x2) : x1 - 0.25x2^2 <= 0.0
%
%      0 <= x1 <= 10 ; 0 <= x2 <= 10
%
% NOTE: The hash marks for the inequality
% constraints must be determined and drawn outside
% of this exercise and on the printout
%-------------------------------------------------------
x1=0:0.1:10; % the semi-colon at the end prevents
% the echo
x2=0:0.1:10; % these are also the side constraints
% x1 and x2 are vectors filled with numbers starting
% at 0 and ending at 10.0 with values at intervals
% of 0.1

[X1 X2] = meshgrid(x1,x2);
% generates matrices X1 and X2 corresponding to
% vectors x1 and x2. It is a mesh of x1 and x2 values
% at which the functions will be evaluated
f1 = obj_ex1(X1,X2);% the objective function is
% evaluated over the entire mesh and stored in f1
% MATLAB will compute the values for the objective
% througha function m-file called obj_ex1.m
ineq1 = inecon1(X1,X2);  % the inequality g1 is
                         % evaluated over the mesh
ineq2 = inecon2(X1,X2);  % the inequality g2 is
                         % evaluated over the mesh
eq1 = eqcon1(X1,X2);     % the equality h1 is evaluated
                         % over the mesh
eq2 = eqcon2(X1,X2);     % the equality h2 is evaluated
                         % over the mesh
```

```
[C1,han1] = contour(x1,x2,ineq1,[7,7],'r-');
% a single contour plot of g1 (or ineq1) is drawn for
% the value of 7 in red color as a continuous line
% duplication of the contour value is necessary
% for drawing a single contour
% han1 is the handle to this plot. This handle
% can be used to change the plot display characteristics
% C1 contains the value of the contour

clabel(C1,han1);
% labels the contour with the values in vector C1
% contour(x1,x2,ineq1,[7,7],'r-') will draw the contour
% without labeling the value
hold on   % allows multiple plots in the same figure
%            window
gtext('g1');
% will place the string 'g1' on the plot at the spot
% selected by a mouse click. This is a text label
% this procedure is repeated for remaining
% constraints

[C2,han2] = contour(x1,x2,ineq2,[0,0],'r--');
clabel(C2,han2);
gtext('g2');

[C3,han3] = contour(x1,x2,eq1,[8,8],'b-');
clabel(C3,han3);
gtext('h1');

[C4,han4] = contour(x1,x2,eq2,[4,4],'b--');
clabel(C4,han4);
gtext('h2');

[C,han] = contour(x1,x2,f1,'g');
% contour of 'f' is drawn in green color. The number
% of contours are decided by the default value
clabel(C,han);
xlabel('x1 values','FontName','times','FontSize',12,...
'FontWeight','bold'); % label for x-axes
ylabel('x2 values','FontName','times','FontSize',12, ...
'FontWeight','bold');
grid
hold off
```

The function m-files are:

inecon1.m

```
function retval = inecon1(X1, X2)
retval = X1 + X2;
% X1, X2 are matrices
% retval is the value being returned after the
% computation
% Since X1 and X2 are matrices retval is also a
% matrix
% By this way the entire information on the mesh
% is generated by a single call to the function
% inecon1
```

inecon2.m

```
function retval = inecon2(X1,X2)
retval = X1 - 0.25*X2.^2;
% Note the use of .^ operator for element by element
% operation. That is each element of the X2 matrix
% is squared. Without the dot the implication is a
% matrix multiplication - between matrices whose
% inner dimensions must agree. Similar operators are
% defined for element by element multiplication
% and division
```

econ1.m

```
function retval = eqcon1(X1,X2)
retval = 2.0*X1 + X2;
```

econ2.m

```
function retval = eqcon2(X1,X2)
retval = (X1 - 1).*(X1 - 1) + (X2 - 4).*(X2 - 4);
```

obj_ex1.m

```
function retval = obj_ex1(X1,X2)
retval = (X1 - 3).*(X1 - 3) +(X2 - 2).*(X2 - 2);
```

2.2.3 Displaying the Graphics

All of the files required for graphical display of the problem have been created. In the Command window, type **addpath** followed by the complete path for the directory that holds these files. At the prompt, type the name of the script file for the example without the **.m** extension.

```
>> Ex2_1
```

The first contour plot will appear. Move the mouse over to the plot and a cross hair appears. Clicking on the plot will place the string "g1" at the cross hair. The second contour plot should appear and there is a pause to place the text label. After the four constraints, the objective function is plotted for several contour values. Finally the plot should appear as in Figure 2.1 without the hash marks (and with your choice of the location for constraint labels). You can insert the hash marks through editing the plot directly (Version 5.3 onwards). The solution for the problem is at (1,6), where four plots intersect. The value of the objective function is 20 at the solution.

If the plot is acceptable, you can print the information on the figure by exporting it (using an appropriate extension) to a file and later incorporating it in another document. You can also send it to the printer from the print command on the File menu. Typing **help print** in the Command window should list a set of commands you can use to save the file.

```
>> print -depsc2 plot_ex_2_1.eps
```

will create the level 2 color postscript file called plot_ex_2_1.eps in the working directory. A complete path name should save the file to the specified directory. You can then execute standard operating system commands to dump the file to the applicable printer. You can also save the figure (extension .fig) and later open it in MATLAB using the file open command from menu bar or tool bar.

2.2.4 Customizing the Figure

In this section the basic figure and plots created above will be customized using the Handle Graphics commands first explored in Section 2.2.1. All of the plots describing the constraints will have a linewidth of 2 points. The objective function will have a linewidth of 1 point. The text labels will be in the "Times" font. It will be in boldface with a font of size 14 and in the color of the constraint. The x and y tick marks will be changed to establish a broader grid. The x and y labels will include a subscript. A two-row text will be inserted at the point selected through the mouse. The new figure is shown in Figure 2.2.

First copy the file in the previous exercise and rename it. The renamed file will be used to make the necessary changes to the code. These changes include both new code as well as replacement of existing lines of code. In the following only the changes to the code are illustrated. The original code is shown in the italic style and identified with three ellipsis points (. . .) at the beginning of the line. The code to be inserted or replacement code is shown in normal style. If no replacement is apparent, then the old code is used to define the locations where new code is appended.

```
... [C1,han1] = contour(x1,x2,ineq1,[7,7],'r-');
set(han1,'LineWidth',2); % sets line width of the g1
% contour to 2 points
```


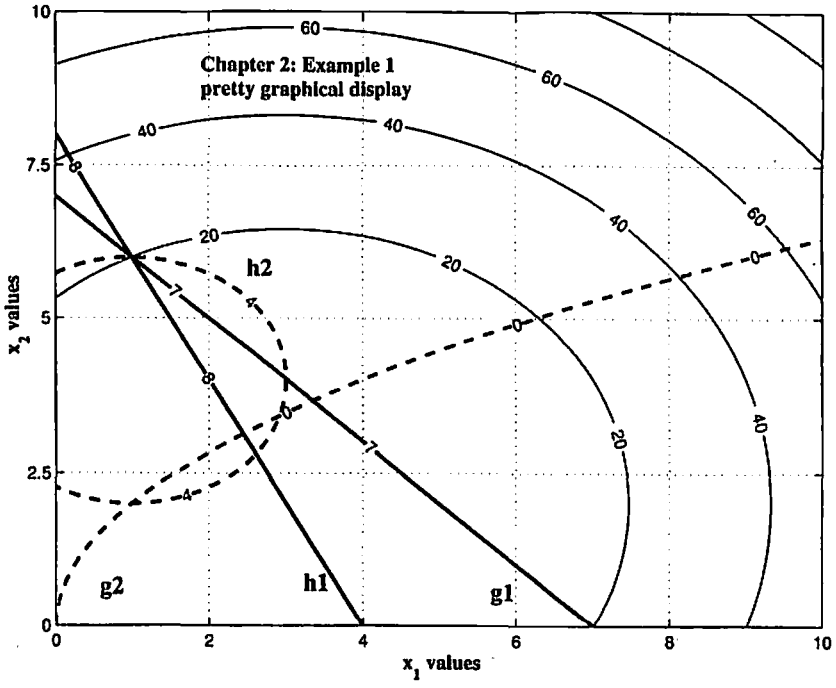

Figure 2.2 Customized figure for Example 2.1.

```
... gtext('g1');
k1 = gtext('g1'); % k1 is the handle to the text element
set(k1,'FontName','Times','FontWeight','bold', ...
 'FontSize',14,'Color','red')

...clabel(C2,han2);
set(han2,'LineWidth',2)
k2 = gtext('g2');
set(k2,'FontName','Times','FontWeight','bold', ...
'FontSize',14,'Color','red')
... clabel(C3,han3);
set(han3,'LineWidth',2)
k3 = gtext('h1');
set(k3,'FontName','Times','FontWeight','bold', ...
'FontSize',14,'Color','blue')

... clabel(C4,han4);
set(han4,'LineWidth',2)
k4 = gtext('h2');
```

```
set(k4,'FontName','Times','FontWeight','bold', ...
'FontSize',14,'Color','blue')
... clabel(C,han);
set(han,'LineWidth',1)
... xlabel('x1 values','FontName','times', ...
xlabel(' x_1 values','FontName','times','FontSize',12,...
'FontWeight','bold');
ylabel(' x_2 values','FontName','times',' ...
FontSize',12, 'FontWeight','bold');
set(gca,'xtick',[0 2 4 6 8 10]) % set xticks
set(gca,'ytick',[0 2.5 5.0 7.5 10]) % set yticks
k5 = gtext({'Chapter 2: Example 1', ...
          'pretty graphical display'})
% the above gtext describes a string array
% string array is defined using curly braces
set(k5,'FontName','Times','FontSize',12, ...
'FontWeight','bold')

... (at the end)
clear C C1 C2 C3 C4 h h1 h2 h3 h4 k1 k2 k3 k4 k5
% get rid of variables from the workspace
```

Run the program to see the figure in Figure 2.2. Once the plot is customized to your satisfaction, you can make it the standard for other plots you will produce in MATLAB. This file could also be a template for general contour plotting. Note that all function information is obtained through function m-files which are coded outside the script file. Setting up new problems or examples only requires adding/changing new function m-files.

2.3 ADDITIONAL EXAMPLES

The following additional examples will serve to illustrate both optimization problems as well as additional graphical features of MATLAB that will be useful in developing graphical solutions to optimization problems. The graphical routines in MATLAB are powerful and easy to use. They can graphically display the problems in several ways with very simple commands. The useful display is, however, determined by the user. The first example in this section, Example 2.2, is a problem in unconstrained optimization. The second example is a structural engineering problem of reasonable complexity. The third example demonstrates optimization in the area of heat transfer design.

2.3.1 Example 2.2

This example illustrates several different ways of graphically displaying a function of two variables. The problem was used to illustrate global optimization in Reference 3. The single objective function is

$$f(x_1, x_2) = ax_1^2 + bx_2^2 - c\cos(px_1) - d\cos(qx_2) + c + d \tag{2.9}$$

with

$$a = 1, b = 2, c = 0.3, d = 0.4, p = 3\pi, q = 4\pi$$

Figures 2.3–2.7 are the graphical display of solutions in this section. The figures in the book being restricted to black and white will not convey the impact of color images you will see on the screen. There are two files associated with the plots: the script m-file (*ex2_2.m*) and the function m-file (*obj_ex2.m*).

ex2_2.m

```
% Chapter 2: Optimization with MATLAB
% Dr. P.Venkataraman
% Example 2.2 Sec.2.3.1
%
```

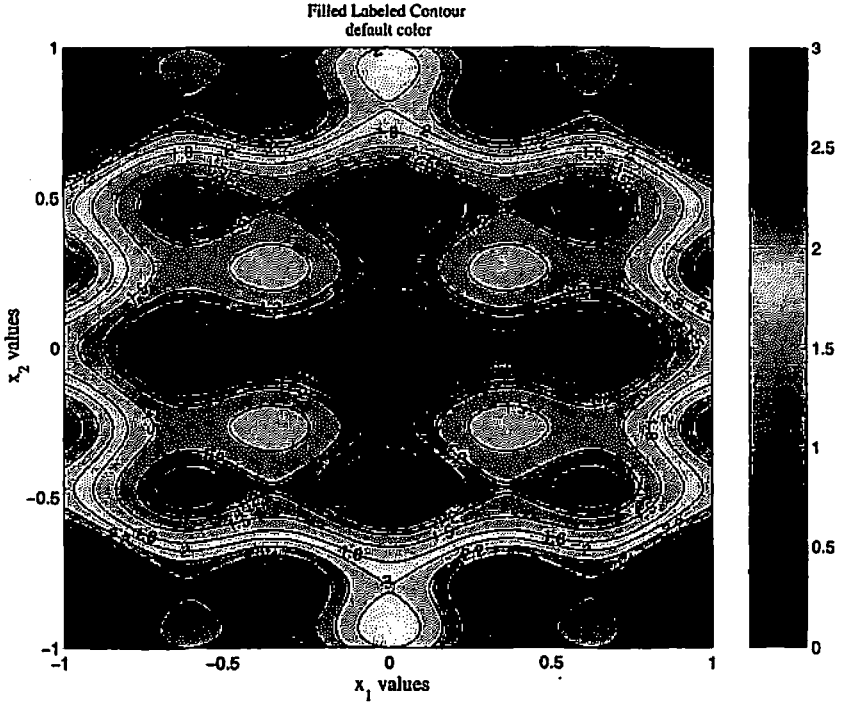

Figure 2.3 Filled contours with colorbar: Example 2.2.

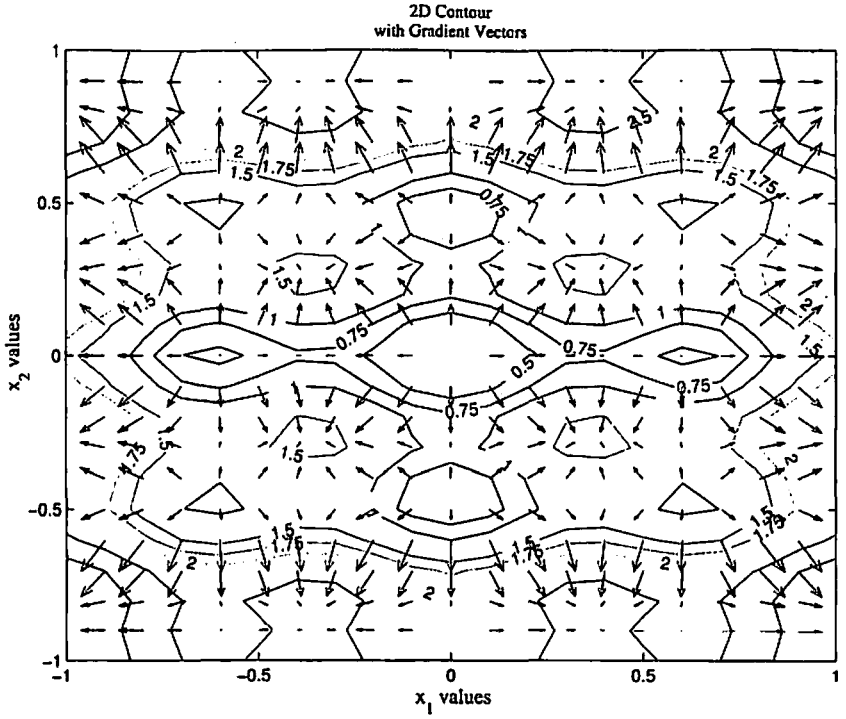

Figure 2.4 Contour with gradient vectors: Example 2.2.

```
% graphical solution using MATLAB (two design variables)
% Unconstrained function illustrating global minimum
% Example will introduce 3D plots, 3D contours, filled
% 2D contours with gradient information
%-------------------------------------------------------
x1=-1:0.01:1;    % the semi-colon at the end prevents
                 % the echo
x2=-1:0.01:1;    % these are also the side constraints
% x1 and x2 are vectors filled with numbers starting
% at -1 and ending at 1.0 with values at intervals of
% 0.01

[X1 X2] = meshgrid(x1,x2);
% generates matrices X1 and X2 corresponding
% vectors x1 and x2
% reminder MATLAB is case sensitive
```


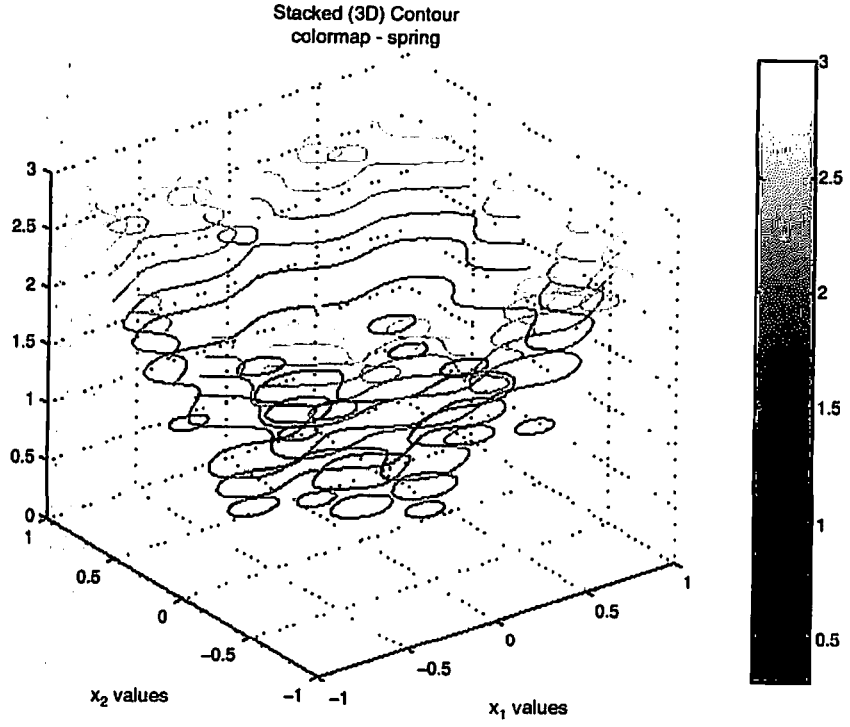

Figure 2.5 3D contour with colorbar: Example 2.2.

```
f1 = obj_ex2(X1,X2);  % the objective function is
                      % evaluated over the entire mesh
% filled contour with default colormap
% help graph3d gives you the choices for colormap
[C1,han1] = contourf(x1,x2,f1, . . .
[0 0.1 0.6 0.8 1.0 1.2 1.5 1.8 2.0 2.4 2.6 2.8 3.0]);

% specific contour levels indicated above
clabel(C1,han1);
colorbar % illustrates the default color scale
set(gca,'xtick',[-1 -0.5 0.0 0.5 1.0]) % custom ticks
set(gca,'ytick',[-1 -0.5 0.0 0.5 1.0]) % custom ticks
grid
xlabel(' x_1 values','FontName','times', ...
 'FontSize',12);
% label for x-axes
ylabel(' x_2 values','FontName','times', ...
```

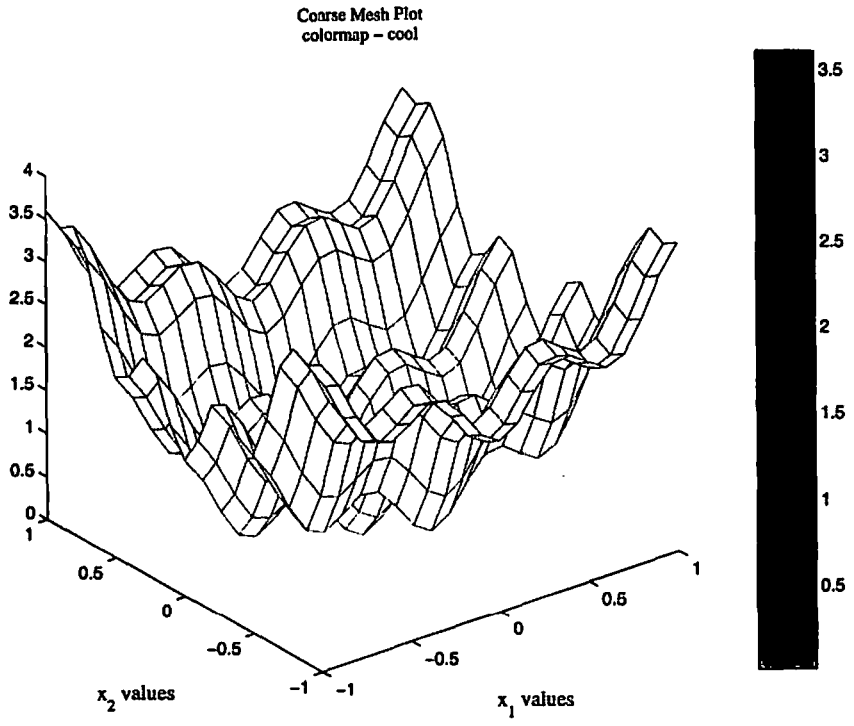

Figure 2.6 3D mesh plot with colorbar: Example 2.2.

```
  'FontSize',12);
title({'Filled Labelled Contour','...
 default color map'},'FontName','times','FontSize',10)

% a new figure is used to draw the
% basic contour plot superimposed with gradient
% information
% also information is generated on a coarser mesh to
% keep the figure tidy. grid is removed for clarity

figure % a new figure window is drawn
y1 = -1:0.1:1.0;
y2 = -1:0.1:1;
[Y1,Y2] = meshgrid(y1,y2);
f2 = obj_ex2(Y1,Y2);
[C2,han2] = contour(y1,y2,f2, ...
          [0 0.5 0.75 1.0 1.5 1.75 2.0 2.5 3.0]);
```


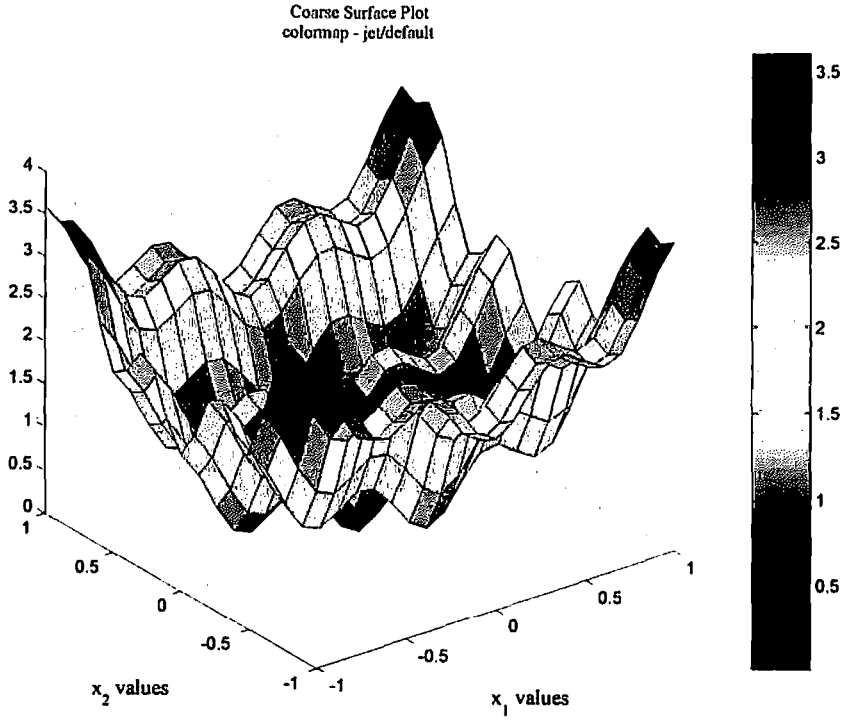

Figure 2.7 3D surface plot with default colorbar: Example 2.2.

```
clabel(C2,han2)
[GX, GY] = gradient(f2,0.2);
% generation of gradient information see help gradient
hold on   % multiple plots on the same figure
quiver(Y1,Y2,GX,GY);
% draws the gradient vectors at grid points
% see help quiver
hold off
set(gca,'xtick',[-1 -0.5 0.0 0.5 1.0])
set(gca,'ytick',[-1 -0.5 0.0 0.5 1.0])
xlabel(' x_1 values','FontName','times', ...
   'FontSize',12);
ylabel(' x_2 values','FontName','times', ...
  'FontSize', 12);
title({'2D Contour','with Gradient Vectors'}, ...
   'FontName','times','FontSize',10)
```

```
% A final example of contour information is a 3D
% contour plot - or a stacked contour plot
figure
colormap(spring)
contour3(x1,x2, f1, ...
      [0 0.3 0.6 0.8 1.0 1.5 1.8 2.0 2.4 2.6 2.8 3.0]);
set(gca,'xtick',[-1 -0.5 0.0 0.5 1.0])
set(gca,'ytick',[-1 -0.5 0.0 0.5 1.0])
% change colormap and set colorbar
% changing color for display is easy by selecting
% among some colormaps provided by MATLAB
colorbar
xlabel(' x_1 values','FontName','times', ...
  'FontSize', 12);
ylabel(' x_2 values','FontName','times',...
  'FontSize', 12);
title({'Stacked (3D) Contour','colormap - spring'}, ...
'FontName','times','FontSize',10)

grid

% the next two figures will display 3D plots
% the first is a mesh plot of the function
% once more coarse data is used for clarity
figure
colormap(cool) % another colormap
mesh(y1,y2,f2) % using information generated earlier
set(gca,'xtick',[-1 -0.5 0.0 0.5 1.0])
set(gca,'ytick',[-1 -0.5 0.0 0.5 1.0])
colorbar
xlabel(' x_1 values','FontName','times', ...
  'FontSize', 12);
ylabel(' x_2 values','FontName','times',...
  'FontSize', 12);
title({'Coarse Mesh Plot','colormap - cool'}, ...
'FontName','times','FontSize',10)

grid

% the final plot in this series
% surface plot with default colormap
figure
colormap(jet)
surf(y1,y2,f2) % using old information
colorbar
xlabel(' x_1 values','FontName','times',...
```



```
  'FontSize', 12);
ylabel(' x_2 values','FontName','times', ...
  'FontSize', 12);
set(gca,'xtick',[-1 -0.5 0.0 0.5 1.0])
set(gca,'ytick',[-1 -0.5 0.0 0.5 1.0])
title({'Coarse Surface Plot','colormap - jet/ ...
default'},'FontName','times','FontSize',10)

grid
```

Obj_ex2.m

```
function retval = obj_ex1(X1,X2)
% Optimization with MATLAB
% Dr. P.Venkataraman
% Chapter 2. Example 2.2
%
% f(x1,x2) = a*x1^2 + b*x2^2 -c*cos(aa*x1)-
% d*cos(bb*x2)...
% + c + d
a = 1; b = 2; c = 0.3; d = 0.4; aa = ...
3.0*pi; bb = 4.0*pi;

% note matrix operations need a dot operator
retval = a*X1.*X1 + b*X2.*X2 -c*cos(aa*X1) - ...
   d*cos(bb*X2) + c + d;
```

The brief comments in the code should provide an explanation of what you see on the figure. Figures can be further customized as seen in the previous section. From an optimization perspective, Figure 2.4 provides the best information about the nature of the problem. The 2D contour curves identify the neighborhood of the local minimum. The gradient vectors indicate the direction of the functions steepest rise at the point, so peaks and valleys can be distinguished. The contours themselves can be colored without being filled.

The quiver plot shown in Figure 2.4 also provides a mechanism to indicate the feasible region when dealing with inequality constraints since they indicate the direction in which the constraint function will increase. If several functions are being drawn, then the clutter produced by the arrows may diffuse the clarity. The user is encouraged to use the powerful graphical features of MATLAB to his benefit at all times without losing sight of the objective of his effort. MATLAB graphics has many more features than will be covered in this chapter. The exposure in this chapter should be sufficient for the reader to confidently explore many other useful graphics commands.

The 3D mesh and surface plots have limited usefulness. These plots can be used to reinforce some of the features found in Figure 2.4. The information in these plots may be improved by choosing a camera angle that emphasizes some aspect of the graphical description. This exploration is left to the reader as an exercise. Using *help view* in the MATLAB Command window should get you started in this direction.

2.3.2 Example 2.3

The next example is a complex one from structural engineering design that is relevant in civil/mechanical/aerospace engineering applications. It appeared as a problem in Reference 4. It is developed in detail here. The problem is to redesign the basic tall flagpole in view of the phenomenal increase in wind speeds during extreme weather conditions. In recent catastrophic events, the wind speeds in tornadoes have been measured at over 350 miles per hour. These high speeds appear to be the norm rather than an unusual event.

Design Problem: Minimize the mass of a standard 10-m tubular flagpole to withstand wind gusts of 350 miles per hour. The flagpole will be made of structural steel. Use a factor of safety of 2.5 for the structural design. The deflection of the top of the flagpole should not exceed 5 cm. The problem is described in Figure 2.8.

Mathematical Model: The mathematical model is developed in detail for completeness and to provide a review of useful structural [5] and aerodynamic relations [6]. The relations are expressed in original symbols rather than in standard format of optimization problems to provide an insight into problem formulation.

Design Parameters: The structural steel [5] has the following material constants:

E (modulus of elasticity): 200 E+09 Pa
σ_{all} (allowable normal stress): 250 E+06 Pa
τ_{all} (allowable shear stress): 145 E+06 Pa

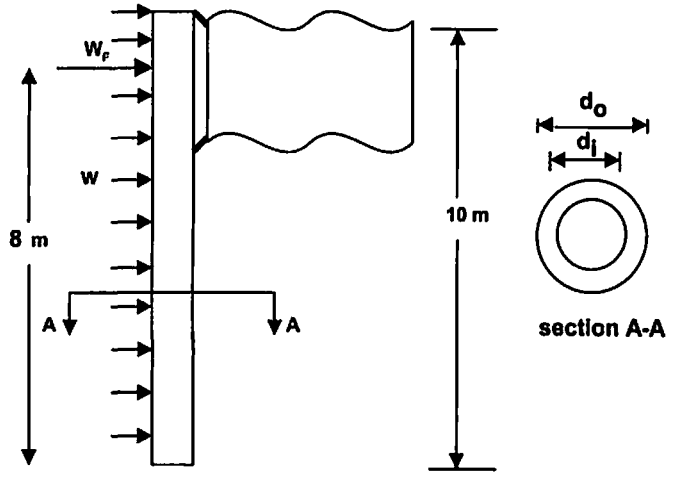

Figure 2.8 Flagpole design: Example 2.3.

γ (material density): 7860 kg/m^3
FS (factor of safety): 2.5
g (gravitational acceleration) = 9.81 m/s^2

For the aerodynamic calculations the following are considered:

ρ (standard air density): 1.225 kg/m^3
C_d (drag coefficient of cylinder): 1.0
W_F (flag wind load at 8 m): 5000 N
V_W (wind speed): 350 mph (156.46 m/s)

The geometric parameters are

L_p: the location of flag wind load (8 m)
L: length of the pole (10 m)
δ_{all}: permitted deflection (5 cm)

Design Variables: The design variables shown in Fig 2.8 are

d_o: outside diameter (x_1) [Note: x's are not used in the model]
d_i: inside diameter (x_2)

Geometric Relations: The following relations will be useful in later calculations:
A: area of cross-section = $0.25 * \pi * (d_o^2 - d_i^2)$
I: diametrical moment of inertia = $\pi * (d_o^4 - d_i^4)/64$
Q/t: first moment of area above the neutral axis divided by thickness
$= (d_o^2 + d_o d_i + d_i^2)/6$

Objective Function: The objective function is the weight of the 10-m uniform flagpole:

$$\text{Weight: } f(x_1, x_2): L * A * \gamma * g \tag{2.10}$$

Constraint Functions: The wind load per unit length (F_D) on the flagpole is calculated as

$$F_D = 0.5 * \rho * V_W^2 * C_d * d_o$$

The bending moment at the base of the pole due to this uniform wind load on the entire pole is

$$M_W = 0.5 * F_D * L * L$$

The bending moment due to the wind load on the flag is

$$M_F = W_P * L_p$$

Bending (normal) stress at the base of the pole is

$$\sigma_{bend} = 0.5 * (M_W + M_F) * d_o/I$$

Normal stress due to the weight is

$$\sigma_{weight} = \gamma * g * L$$

Total normal stresses to be resisted for design is the sum of the normal stresses computed above. Incorporating the factor of safety and the allowable stress from material values, the first inequality constraint can be set up as

$$g_1(x_1, x_2): \quad \sigma_{bend} + \sigma_{weight} \le \sigma_{all}/FS \tag{2.11}$$

The maximum shear load in the cross section is

$$S = W_F + F_D * L$$

The maximum shear stress in the pole is

$$\tau = S * Q/(I * t)$$

The second inequality constraint based on handling the shear stresses in the flagpole is

$$g_2(x_1, x_2): \quad \tau \le \tau_{all}/FS \tag{2.12}$$

The third practical constraint is based on the deflection of the top of the pole. This deflection due to a uniform wind load on the pole is

$$\delta_W = F_D * L^4/(8 * E * I)$$

The deflection at the top due to the flag wind load at L_p is

$$\delta_F = (2 * W_F * L^3 - W_F * L * L * L_p)/(E * I)$$

The third constraint translates to

$$g_3(x_1, x_2) \quad \delta_W + \delta_F \le \delta_{all} \tag{2.13}$$

To discourage solutions where $d_o < d_i$, we will include a geometric constraint:

$$g_4(x_1, x_2) \quad d_o - d_i \ge 0.001 \tag{2.14}$$

Side Constraints: This defines the design region for the search.

$$2 \text{ cm} \le d_o \le 100 \text{ cm}; \quad 2 \text{ cm} \le d_i \le 100 \text{ cm} \tag{2.15}$$

MATLAB Code: The m-files for this example are given below. An important observation in this problem, and structural engineering problems in particular, is the order of magnitude of the quantities in the constraining equations. The stress constraints are of the order of 10E+06, while the displacement terms are of the order of 10E-02. Most numerical techniques struggle to handle this range. Typically, such problems need to be normalized before being solved. It is essential in applying numerical techniques used in optimization.

This example is plotted in two parts. In the first part, each inequality constraint is investigated alone. Two curves are shown for each constraint. This avoids clutter. The side for the location of the hash mark on the constraint is determined by drawing these curves in color. The blue color indicates the feasible direction. Alternately, quiver plots can be used. The second part is the consolidated curve shown. The consolidated curve has to be drawn by removing the comments on some of the code and commenting the code that is not needed. Since the optimal solution cannot be clearly established, the zoom feature of MATLAB is used to narrow down the solution.

ex2_3.m (the main script file)

```
% Chapter 2: Optimization with MATLAB
% Dr. P.Venkataraman
% Example 2.3 Sec.2.3
%
% graphical solution using MATLAB (two design variables)
% Optimal design of a Flag Pole for high winds
% Ref. 2.4
%-------------------------------------------------------
% global statement is used to share same information
% between various m-files
global ELAS SIGALL TAUALL GAM FS GRAV
global RHO CD FLAGW SPEED LP L DELT
%-------------------------------------------------------
% Initialize values
ELAS = 200e+09;          % Modulus of elasticity -Pa
SIGALL = 250E+06;        % allowable normal stress -Pa
TAUALL = 145e+06;        % allowable shear stress - Pa
```

```
GAM= 7860;              % density of material - kg/m3
FS = 2.5;               % factor of safety
GRAV = 9.81;            % gravitational acceleration
%-------------------
RHO = 1.225;            % density of air - kg/m3
CD = 1.0;               % drag coefficient
FLAGW = 5000;           % concentrated load on flag - N
SPEED = 156.46;         % m/s
%-------------------
LP = 8;                 % location of drag load on flag - m
L = 10;                 % length of pole
DELT = 0.05;            % allowable deflection - m
%-------------------
g1val = SIGALL/FS       % right hand side values
g2val = TAUALL/FS       % for the constraints
g3val = DELT
g4val = 0.001
%-----------------------------------------------------
x1=0.02:0.01:1; % the semi-colon at the end prevents
% the echo
x2=0.025:0.01:1; % these are also the side constraints
% x1 and x2 are vectors filled with numbers
% note a way to avoid x1 = x2
[X1 X2] = meshgrid(x1,x2);
% generates matrices X1 and X2 corresponding to
% vectors x1 and x2

f1 = obj_ex3(X1,X2);
% the objective function is evaluated over the entire
% mesh

% Constraints are evaluated
ineq1 = ineq1_ex3(X1,X2);
ineq2 = ineq2_ex3(X1,X2);
ineq3 = ineq3_ex3(X1,X2);
ineq4 = ineq4_ex3(X1,X2);

[C1,han1] = contour(x1,x2,f1,[0,10000,50000, ...
100000, 150000, 200000, 250000, 300000],'g-');
clabel(C1,han1);
set(gca,'xtick',[0 0.2 0.4 0.6 0.8 1.0])
set(gca,'ytick',[0 0.2 0.4 0.6 0.8 1.0])
xlabel('outside diameter','FontName','times', ...
'FontSize',12);
```



```
ylabel('inside diameter','FontName','times', ...
'FontSize',12)
grid
% hold on

figure % a new figure window
contour(x1,x2,ineq1,[g1val,g1val],'r-');
hold on
% draw another contour at 10% the constraint boundary
contour(x1,x2,ineq1,[0.1*g1val,0.1*g1val],'b-');
set(gca,'xtick',[0 0.2 0.4 0.6 0.8 1.0])
set(gca,'ytick',[0 0.2 0.4 0.6 0.8 1.0])
xlabel('outside diameter','FontName','times','...
FontSize',12);
ylabel('inside diameter','FontName','times','...
FontSize',12);
hold off
grid

% the following code may be useful for the consolidated
% figure. Not used here __ uncomment below
% [C2,han2] = contour(x1,x2,ineq1,[g1val, g1val],'r-');
% clabel(C2,han2);
% set(h2,'LineWidth',1)
% k2 = gtext('g1');
% set(k2,'FontName','Times','FontWeight','bold',...
% FontSize',14%,'Color','red')

figure
contour(x1,x2,ineq2,[g2val,g2val],'r-');
hold on
contour(x1,x2,ineq2,[0.1*g2val,0.1*g2val],'b-');
set(gca,'xtick',[0 0.2 0.4 0.6 0.8 1.0])
set(gca,'ytick',[0 0.2 0.4 0.6 0.8 1.0])
xlabel('outside diameter','FontName','times', ...
'FontSize',12);
ylabel('inside diameter','FontName', ...
'times', 'FontSize',12);
hold off
grid

%[C3,han3] = contour(x1,x2,ineq2,[g2val,g2val],'r--');
%clabel(C3,han3);
%set(h3,'LineWidth',1)
```

```
%k3 = gtext('g2');
%set(k3,'FontName','Times','FontWeight','bold', ...
% 'FontSize',14,'Color','red')

figure
contour(x1,x2,ineq3,[g3val,g3val],'r-');
hold on
contour(x1,x2,ineq3,[0.1*g3val,0.1*g3val],'b-');
set(gca,'xtick',[0 0.2 0.4 0.6 0.8 1.0])
set(gca,'ytick',[0 0.2 0.4 0.6 0.8 1.0])
xlabel('outside diameter','FontName','times', ...
'FontSize',12);
ylabel('inside diameter','FontName','times', ...
'FontSize',12);
hold off
grid

% [C4,han4] = contour(x1,x2,ineq3,[g3val,g3val],'b-');
% clabel(C4,han4);
% set(h4,'LineWidth',1)
% k4 = gtext('g3');
% set(k4,'FontName','Times','FontWeight','bold', . . .
% 'FontSize',14,'Color','blue')

figure
contour(x1,x2,ineq4,[g4val,g4val],'r-');
hold on
contour(x1,x2,ineq4,[0.001*g4val,0.001*g4val],'b-');
set(gca,'xtick',[0 0.2 0.4 0.6 0.8 1.0])
set(gca,'ytick',[0 0.2 0.4 0.6 0.8 1.0])
xlabel('outside diameter','FontName','times', ...
'FontSize',12);
ylabel('inside diameter','FontName','times', ...
'FontSize',12);
hold off
grid

% [C5,han5] = contour(x1,x2,ineq4,[g4val,g4val],
%'b--');
% clabel(C5,han5);
% set(h5,'LineWidth',1)
% k5 = gtext('g4');
% set(k5,'FontName','Times','FontWeight','bold', ...
% 'FontSize',14,'Color','blue')
```



```
% the equality and inequality constraints are
% not written with 0 on the right hand side.
% If you do write them that way you would have to
% include [0,0] in the contour commands
```

ineq1_ex3.m (the first constraint)

```
function retval = ineq1_ex3(X1,X2)
% global statement is used to share same information
% between various m-files
global ELAS SIGALL TAUALL GAM FS GRAV
global RHO CD FLAGW SPEED LP L DELT

AREA = 0.25* pi*(X1.^2 - X2.^2); % matrix
INERTIA = pi*(X1.^4 - X2.^4)/64; % matrix
FD = 0.5*RHO*SPEED*SPEED*CD*X1;
MW = 0.25*FD*L*L.*X1./INERTIA;
MF = 0.5*FLAGW * LP*X1./INERTIA;

SIGW = GAM*GRAV*L;

retval = MW + MF + SIGW;% SIGW is added to all matrix
% elements
```

ineq2_ex3.m (the second constraint)

```
function retval = ineq2_ex3(X1,X2)
% global statement is used to share same information
% between various m-files
global ELAS SIGALL TAUALL GAM FS GRAV
global RHO CD FLAGW SPEED LP L DELT

AREA = 0.25* pi*(X1.^2 - X2.^2);
INERTIA = pi*(X1.^4 - X2.^4)/64;
FD = 0.5*RHO*SPEED*SPEED*CD*X1;
S = FLAGW + (FD * L);
Q = (X1.*X1 + X1.*X2 + X2.*X2)/6.0;
retval = S.*Q./INERTIA;
```

ineq3_ex3.m (the third constraint)

```
function retval = ineq3_ex3(X1,X2)
% global statement is used to share same information
% between various m-files
global ELAS SIGALL TAUALL GAM FS GRAV
global RHO CD FLAGW SPEED LP L DELT
```

```
AREA = 0.25* pi*(X1.^2 - X2.^2);
INERTIA = pi*(X1.^4 - X2.^4)/64;
FD = 0.5*RHO*SPEED*SPEED*CD*X1;
dw = FD*L^4./(8*ELAS*INERTIA);
df = (2.0*FLAGW*L^3 - FLAGW*L*L*LP)./(ELAS*INERTIA);

retval = dw + df;
```

ineq4_ex3.m (the fourth constraint)

```
function retval = ineq4_ex3(X1,X2)
retval = X1 - X2;
```

Figure 2.9 displays the graphical or consolidated solution to the problem. The code is available in the m-file above (by commenting the codes that create the new figure windows and removing the comments on the code that is currently commented). The optimal solution is not very clear. Figure 2.10 is obtained by zooming in near the neighborhood of 0.6. See *help zoom* for instruction on its use. It can be achieved by

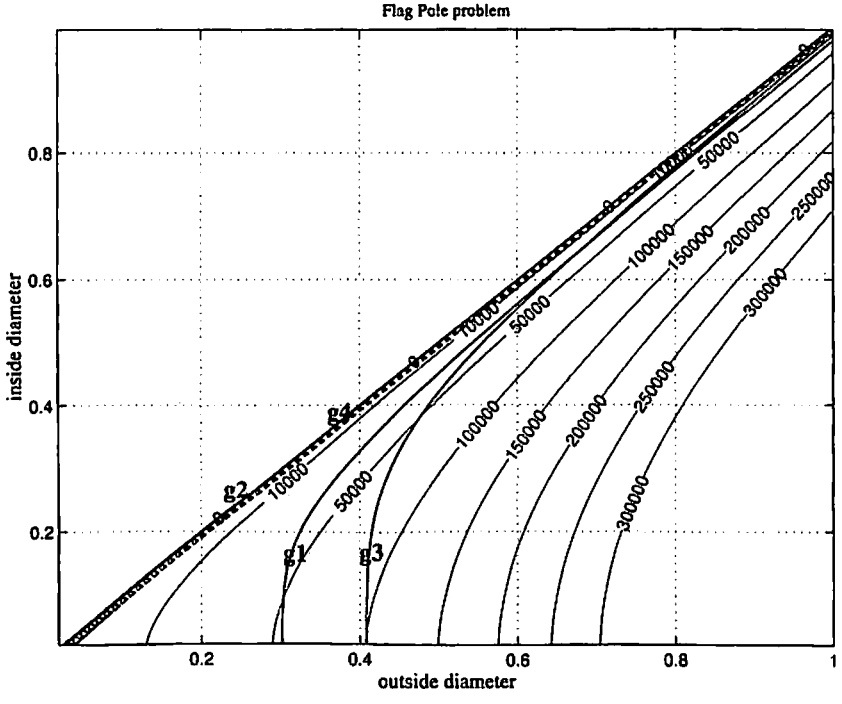

Figure 2.9 Graphical solution: Example 2.3.

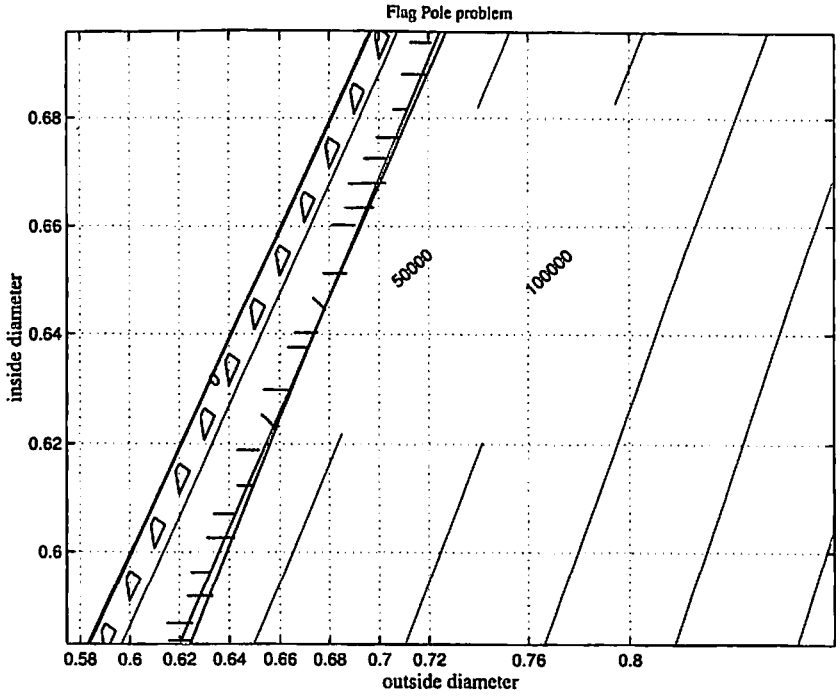

Figure 2.10 Graphical solution (zoomed): Example 2.3.

typing **zoom** at the workspace prompt and using the mouse to drag a rectangle around the region that needs to be enlarged. From Figure 2.10 the solution is around the outside diameter of 0.68 m and the inside diameter of 0.65 m. Typing **zoom** in the workspace again will toggle the figure back to normal state. In Figure 2.10, the tick marks are placed on the figure through the command window for better interpretation of the solution.

The graphics in the above example were created using the same statements encountered previously. Color contours were used to establish the feasible region. The zoom feature was employed to obtain a better estimate of the solution.

2.3.3 Example 2.4

This example is from the area of heat transfer. The problem is to design a triangular fin of the smallest volume that will at least deliver specified fin efficiencies. The graphical feature of this code is very similar to Example 2.3. In this example, the inequality constraints are computed and returned from a single function m-file rather than separate files considered in the previous example. Another new feature in this example is to invoke special mathematical functions, the Bessel functions that are

available in MATLAB. This example also illustrates a problem where the optimization problem can be adequately defined but the solution is easily determined from the equality and side constraints. In other words, the problem can easily accommodate additional demanding constraints. In large complex mathematical models with many design variables it is not easy to ensure that at least one of the inequality constraints is active.

Design Problem: Minimize the amount of material used in the design of a series of identical triangular fins that cover a given area and operate at or above the specified efficiencies.

Mathematical Model: This type of problem should be able to accommodate several design variables. In this section, the problem is set up to have two design variables. The fin material is aluminum.

Design Parameters: For aluminum

$h = 50$ W/m^2 [convection coefficient]
$k = 177$ W/m-K [thermal conductivity]
$N = 20$ [number of fins]
$W = 0.1$ m [width of the fins]

Fin gap is the same as the base length of the triangular fin.

Design Variables

b: base of the triangular fin
L: height of the triangular fin

Figure 2.11 illustrates the geometry of the fins.

Geometric Relations: The following are some of the area calculations that are used later in the development of the constraints:

$A_f = (2N - 1)*b*W$	footprint of the fin and gap
$A_c = 2*W*[L^2 + (b/2)^2]^{1/2}$	fin area for heat transfer
$A_b = (N - 1)*b*W$	gap area
$A_t = N*A_c + A_b$	total area for heat transfer

Objective Function: The total volume of material for the fin is

$$f(b, L) = 0.5*N*W*b*L \tag{2.16}$$

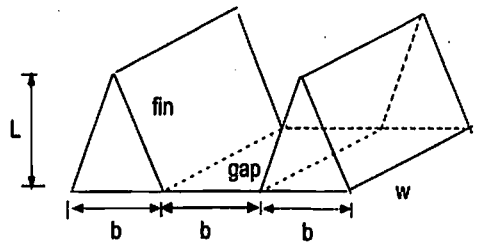

Figure 2.11 Fin design for heat transfer: Example 2.4.

Constraint Functions: The heat transfer equations for the fin are available in Reference 7. Fins are typically mounted on a specified area. The first constraint is one on area:

$$h(b, L): A_f = 0.015 \text{ m}^2 \tag{2.17}$$

Note: this constraint essentially fixes the value of b. In the graph this constraint will be a straight line parallel to the L axis. If this were to happen in a problem with several design variables, it would be prudent to eliminate this variable from the mathematical model by identifying it as a design parameter.

The efficiency of a single fin can be established as

$$\eta_f = (1/mL)\, I_1(2mL)/I_0(2mL)$$

where

$$m = (2h/kb)^{1/2}$$

and the I's are Bessel equations of the *first kind*. The first inequality constraint is

$$g_1(b, L): \quad \eta_f \geq 0.95 \tag{2.18}$$

The overall efficiency for the heat transfer is

$$\eta_o = 1 - N*(A_f/A_t)(1 - \eta_f)$$

The final inequality constraint is

$$g_2(b, L): \eta_o \geq 0.94 \tag{2.19}$$

Side Constraints: The side constraints are

$$0.001 \leq b \leq 0.005 \tag{2.20}$$
$$0.01 \leq L \leq 0.03$$

MATLAB Code: Similar to the previous example, the hash marks on the inequality constraint can be established by plotting two values of the constraint: the limit value and a value higher. Once they have been determined, then the consolidated figure can be established. The code shown here will draw all the functions in separate windows. For the consolidated plot this code needs to be edited. It is left to the student as an exercise.

ex2_4.m The main script file

```
%Chapter 2: Optimization with MATLAB
% Dr. P.Venkataraman
% Example 2.4 Sec.2.3
%
% graphical solution using MATLAB (two design variables)
% Minimum fin volume for efficient heat transfer
% material Aluminum
%-----------------------------------------------------
% global statement is used to share same information
% between various m-files
global N H K W AREA
%-----------------------------------------------------
% Initialize values
N = 20          % number of fins
W = 0.1         % width of fins
H = 50.0  % convection coefficient W/m*m
K = 177.0 % thermal conductivity W/m-K
AREA = 0.015 % available fin foot print area
%-----------------------------------------------------
% right hand limits for the functions
h1val = AREA;
g1val = 0.95;
g2val = 0.94;
%-----------------------------------------------------
x1=0.001:0.0001:0.005;
x2=0.01:0.001:0.03; % x1 and x2 are vectors filled with
%                     numbers
[X1 X2] = meshgrid(x1,x2);
% generates matrices X1 and X2 corresponding to
% vectors x1 and x2

f1 = obj_ex4(X1,X2);
% the objective function is evaluated over the entire
% mesh

% Constraints are evaluated
```



```
eq1 = eq1_ex4(X1,X2); % the equality constraint

[ineq1, ineq2] = ineq_ex4(X1,X2);
% in the above note that the two inequality constraints
% are obtained simultaneously. Keep in mind
% that each constraint is a matrix

% ready for the plots
[C1,han1] = contour(x1,x2,f1,[0.00001, 0.00002,...
     0.00004, 0.00006, 0.00008, 0.0001],'g-');
clabel(C1,han1);
set(gca,'xtick',[0.001 0.0015 0.002 0.0025 0.003 ...
0.0035 0.004 0.0045 0.005])
set(gca,'ytick',[0.010 0.015 0.02 0.025 0.03 0.125]);
xlabel('fin length','FontName','times','FontSize',12);
ylabel('fin width','FontName','times','FontSize',12)
grid
% hold on

figure % open a new figure window
% plotting the equality constraint
contour(x1,x2,eq1,[h1val,h1val],'r-');
set(gca,'xtick',[0.001 0.0015 0.002 0.0025 0.003 ...
      0.0035 0.004 0.0045 0.005])
set(gca,'ytick',[0.010 0.015 0.02 0.025 0.03 0.125]);

xlabel('fin length','FontName','times','FontSize',12);
ylabel('fin width','FontName','times','FontSize',12)
grid

% plotting the first inequality constraint
figure
contour(x1,x2,ineq1,[0.968,0.968],'r-');
hold on
contour(x1,x2,ineq1,[0.969,0.969],'b-');
set(gca,'xtick',[0.001 0.0015 0.002 0.0025 0.003 ...
      0.0035 0.004 0.0045 0.005])

set(gca,'ytick',[0.010 0.015 0.02 0.025 0.03 0.125]);
xlabel('fin length','FontName','times','FontSize',12);
ylabel('fin width','FontName','times','FontSize',12)
hold off
grid
% plotting the second inequality constraint
```

```
figure
contour(x1,x2,ineq2,[g2val,g2val],'r-');
hold on
contour(x1,x2,ineq2,[1.01*g2val,1.01*g2val],'b-');
set(gca,'xtick',[0.001 0.0015 0.002 0.0025 0.003 ...
    0.0035 0.004 0.0045 0.005])
set(gca,'ytick',[0.010 0.015 0.02 0.025 0.03 0.125]);
xlabel('fin length','FontName','times','FontSize',12);
ylabel('fin width','FontName','times','FontSize',12)
hold off
grid
```

obj_ex4.m The objective function

```
function retval = obj_ex4(X1,X2)
% volume of the fin
global N H K W AREA
retval = 0.5*N*W*X1.*X2
```

eq1_ex4.m The equality constraint

```
function retval = eq1_ex4(X1,X2)
% the equality constraint on area
global N H K W AREA
retval = (2.0*N - 1)*W*X1;
```

ineq_ex4.m The inequality constraints

```
function [ret1, ret2] = ineq_ex4(X1,X2)
% returns both the inequality constraints
global N H K W AREA
c = 2*sqrt(2.0)*sqrt(H/K)*X2./sqrt(X1);
ret1 = (besseli(1,c)./((0.5*c).*besseli(0,c)));
Ac = 2.0*W*sqrt((X2.*X2 + .25*X1.*X1));
Ab = (N-1)*W*X1;
At = N*Ac + Ab;
Ar = Af./At;
ret2 = (1.0 - N *Ar.*(1 - ret1));
```

The above script files should display each function in a separate figure window. The inequality constraint figures should contain two contours with the blue contour indicating the feasible region. Figure 2.12 displays a composite plot. The solution is determined by the equality constraint and the lower limit on the fin length as discussed previously.

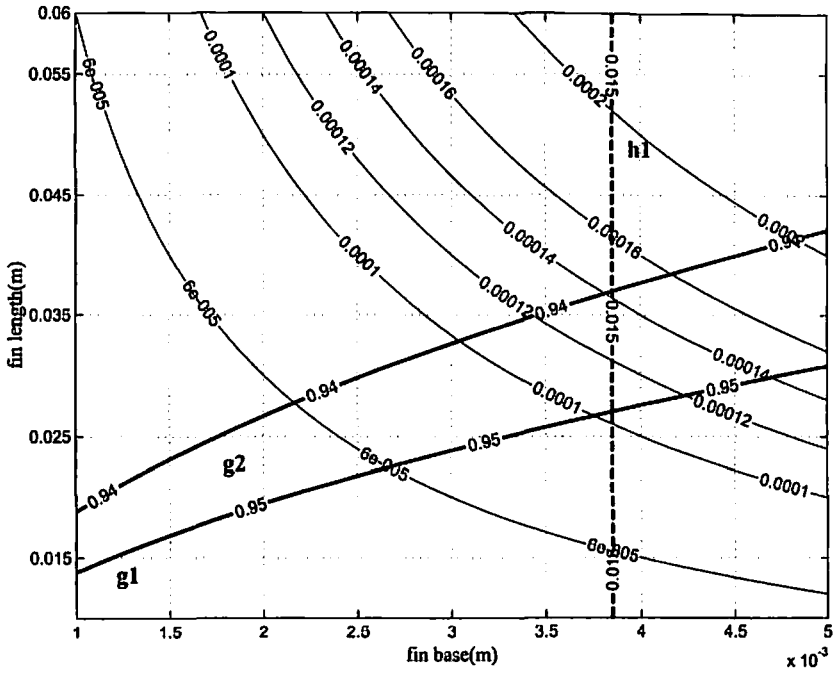

Figure 2.12 Graphical solution: Example 2.4.

2.4 ADDITIONAL MATLAB GRAPHICS

This section provides a brief exposure to additional features of Handle Graphics, including development of a GUI in MATLAB. The coverage is very modest and is included here for the sake of completeness as we have accomplished our goal of graphical optimization in the previous sections. This section can be avoided, postponed, or emphasized for the projects in the course, as it has no additional information for graphical optimization. It does contain reasonable information in understanding MATLAB graphics and programming. The reader is encouraged to take some time getting familiar with the version of MATLAB installed on his machine. There are usually significant enhancements in newer versions of MATLAB. For example, MATLAB Version 5.3 (this book is based on Version 5.2) allows you to customize your plot and add annotations to it through additional editing features of the figure window. You do not have to customize your figure through code alone. In this section, we will also be developing the GUI using MATLAB Handle Graphics so that we have more experience with MATLAB Handle Graphics. MATLAB does provide a GUI creation tool, *guide*, that the user is recommended to explore.

In the previous section there were only two design variables. An equally spaced vector was obtained for each variable based on the side constraint. A meshgrid was

then generated. All functions were evaluated on this grid. Contour plots were drawn for the functions representing the objective and constraints. Except for the objective function which displayed several contours, the constraints were only drawn for a single value. This procedure suggests the possibility of automation. User input will be necessary for selecting the range for the variables as well as the functions that will be plotted.

In this section, we will create plotting facility to include a GUI that will

- Obtain the number of plots
- The range and increment of the design variables
- Use a file selection box/utility that the user will use to identify the function to be plotted
- Prompt the user for the number of contours for each plot
- Allow selection of contour levels
- Create the plot

The exercise will require two m-files: one to set up the GUI and the other to assist in the plotting. The user will be able to tweak the plot in the Command window after it is drawn. The following subsections present a brief introduction to Handle Graphics and GUI controls in Version 5.3.

2.4.1 Handle Graphics

MATLAB Handle Graphics refer to a collection of low-level graphic routines that actually generate or carry out graphical changes you see in the MATLAB figure window. For the most part these routines are transparent and typically users need not be aware of them. In fact, to understand optimization and to program the techniques in MATLAB, it is not necessary to know about Handle Graphics since using the higher-level plot commands such as plot and contour is sufficient to get the job done. These commands actually kick in several of the Handle Graphics routines to display the plot on the figure. A noteworthy feature of MATLAB graphics is that it is implemented in an object-oriented manner. What this means is that most of the entities you see on the figure, like axes, labels, text, and lines, are all objects. This means that most of the graphical items on the figure have properties that can be changed through program code. In the previous sections we used this to manipulate color, font size, linewidth, and the like.

In order to change the properties of objects, it is necessary to identify or refer to them. Objects in MATLAB are identified by a unique number referred to as a handle. Handles can be assigned to all objects when they are created. Some objects are part of other objects. Objects that contain other objects are referred to as container or *parent* objects. There is a definite hierarchical structure among objects. The root of all MATLAB graphic objects is the figure object. The figure object for example can hold several GUI controls, which are known as uicontrol objects. It also contains the axes

object. The axes object in turn has the line, text, and image object. All objects in MATLAB can be associated with a set of properties that is usually based on its function or usefulness. Different object types have different sets of properties. Each of these properties is described through a pair of related information (*name*, *value*). The first element of this pair is the name for the property and the second is the value corresponding to the property. When the object is created, it inherits all of the default properties. For example:

```
plot(t, sin(2*t),'-go', ...
                 'LineWidth', 2, ...
                 'MarkerEdgeColor','k', ...
                 'MarkerFaceColor',[0.49 1 0.63], ...
                 'MarkerSize',12,)
```

The last four lines above represent four properties of the plotted line that is being changed. The property names on the left can be easily understood. The format size units are in points. The color value is a row vector of three color values between zero and one representing the red, green, and blue values, respectively. There are several other properties of the plotted line which will be set at their default values. The MATLAB online reference should list all objects and their properties.

In general, object properties can be changed by using the *set* function to change their values. Similarly, object properties can be read by the use of the *get* function. To inquire about the property of an object:

```
get(handle,'Property Name')
get(h_1,'Color')
```

In the above, h_1 is the handle of the object whose color is desired. Similarly,

```
set(handle, 'Property Name', 'Property Value')
set(h_1,'Color','r') or set(h_1,'Color',[1 0 0])
```

in the above the value of Color for the object represented by the handle h_1 is set to red in two ways. The second specifies the color red through the vector value. It is expected that the object identified through h_1 should have a Color property. MATLAB will inform you of the error in case you are assigning a property/value that is not valid for the object.

2.4.2 Graphical User Interface

The most significant feature of software design and development today is the object-oriented design paradigm that was briefly illustrated in the previous section. The combination of the above with event-driven programming is both a natural and a required feature in current software programs. Instead of the programs controlled by

a predetermined processing sequence (batch processing), today's processing is governed by the user through some directed mouse clicks and buttons. These user selections cause certain events that instruct the software to carry out some action. The objects that fire the events are usually referred to as user interface elements. It is important to understand that the same events can be triggered by obtaining user response to screen prompts at the command line (unattractive). Using buttons to achieve the same result is more glamorous although it uses more computational resources. In fact, for commercial success it is necessary to include this mechanism of user interaction in the software product. Also, users expect these elements to behave in a standard way. There are many elements that the user can use to interact with programs. Many applications across several disciplines have now defined a minimum standard collection of these elements. MATLAB also provides them. The organization of these elements and their presentation to the user in a graphical manner is identified as the graphical user interface. A number of GUI-based tools, functions, are available at the MATLAB site.

This section provides only a brief introduction to the elements used in the example. These elements are primarily used through MATLAB Handle Graphics programming. Consider the case of a button that the user can push to cause some action, referred to as a push button in the subsequent discussion. At the outset, there are two principal requirements that are expected of the push button. First, the user must be able to see it. Second, when the user clicks on it (single versus double clicks can be distinguished in many software development codes) the button must respond with some expected action. In MATLAB the push button is a graphical object. It has many properties that can be used to describe its appearance on the screen. In MATLAB, all graphical objects can only appear in a figure window. Hence, a figure window is essential to display the push button. This figure window acts as a container and can contain many user interface elements. Considering the presentation of the button on the screen, it is necessary to decide where the button should appear on the figure window, how big it should be, what label it should have (typically indicates the kind of action it is likely to cause), and so on. Since the push button is a graphical object, much of these properties can be set if there is a handle available for the push-button object. Once the graphical appearance is taken care of, the response of the push button to user interaction must be prescribed.

This response of the push button is programmed through the CallBack property of the push-button object. The CallBack property is a string. This string is passed to the MATLAB *eval* function (try *help eval* in the Command window). The eval function will execute the string passed to it as an expression or a statement. A very important point to remember is that the eval function is evaluated in the Command window workspace. Generally user interfaces are set up in separate m-files. The results of CallBack evaluation are not available in the m-file workspace that defines the user interface. It is time to introduce the code used to define the push button in the example in this section.

```
Hpush = uicontrol(Hfig_1,'Style','push', . . .
          'Units','inches', . . .
```



```
          'Position',[2.5,0.1,0.7,.2], . . .
          'BackgroundColor','red', . . .
          'String','EXIT', . . .
          'Callback','close');
```

In the above lines of code, Hpush is the handle to the push-button object. It is created by using the function uicontrol. This is the function that will create the user interface elements. The same function is used to place several different user interface elements through the selection of the style property. In the example below, these include label boxes and user editable text boxes. Several parameters identify the type of element that is created and the properties that need to be set. Each property is set through a propertyname/propertyvalue pair. Hfig_1 in the lines of code above refers to the handle of the figure window in which this element will be placed. The type of element created is a push-button element established through the Style property string name "push." The Units used to size and position the push button is inches. The Position property is a row vector of four values giving the Left, Bottom, Width, Height of the element in the Units selected earlier. The default location of the origin is the bottom left corner of the figure window. The BackgroundColor for the button is set to red. The label that will be on the face of the push button is EXIT. This is the Text property of the push-button object. The CallBack property of the push button is the string "close." This string will be the parameter in the eval function that is called when the push/click event is triggered. MATLAB command close will be executed in the Command workspace when the push button is pressed. This will cause the figure window displaying the push button to close.

Only a few elements are used in this section. The CallBack strings are also quite straightforward. Before developing the code, it is useful to observe the hierarchical order in which the graphical objects in MATLAB are organized (reproduced from the MATLAB documentation) in Figure 2.13. The root implies the starting resource for all graphic displays in MATLAB. Its handle usually has a value of 0. The root also serves as a container for all of the figure windows. Another expression for this relationship is that a figure window is the child of the root. From the hierarchy it is essential to note that the user interface elements, `uicontrol` or `uimenu`, can only exist within a

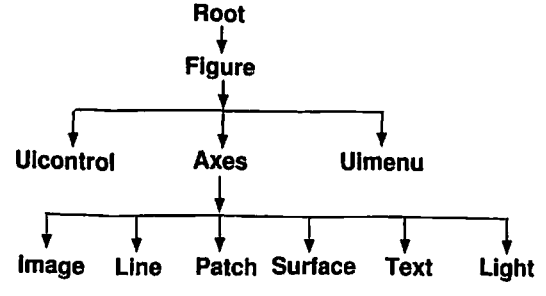

Figure 2.13 Hierarchical structure for MATLAB graphical objects.

figure window. So also the axes object. In the following code each interface element is first associated with a handle so that the property of the element can be referred to if necessary. This is especially true of the text entry boxes to invoke the CallBack strings.

2.4.3 GUI Code

The following code will generate the GUI. The uiwait requires that this window be closed before subsequent processing can take place (or `uiresume` be used in the code).

GUI2_4.m

```
% GUI for Plotting facility
% Dr. P.Venkataraman Applied Optimization Using MATLAB
%
% Ch.2 Sec. 2.4
%
% GUI is created within a figure window
%-----------------------------------------------------
Hfig_1 = figure;       % Hfig_1 is the handle to the
                       % figure window

set(Hfig_1, 'Color',[0.3,0.6,0.5], ... % set color
   'NumberTitle','off', ... % no window title
   'Name', 'Set Range for Design variables', . . .
   'Units','inches', ... % inches used to layout
   % other controls
   'Menubar','none', ...
   'Position',[4,4,4.4,1.5]);

% Position property is [left, bottom, width, height] in
% Units selected

Ht_1 = uicontrol(Hfig_1, ...
   'Style','text', ...
   'Units','inches', ...
   'Position',[0.1,1.1,2.4,.2], . . .
   'String','Number of Functions to Plot');

Ht_2 = uicontrol(Hfig_1, ...
   'Style','edit', ...
   'Units','inches', ...
   'Position',[2.6,1.1,0.5,.2], ...
```



```
   'BackgroundColor','white', ...
   'String','', ...
   'Callback','Np = str2num(get(Ht_2,''String''));');

HL_X1L = uicontrol(Hfig_1, ...
   'Style','text', ...
   'Units','inches', ...
   'Position',[0.1,0.8,0.5,.2], ...
   'String','x1(min)');

HT_X1L = uicontrol(Hfig_1, ...
   'Style','edit', ...
   'Units','inches', ...
   'Position',[0.7,0.8,0.7,.2], ...
   'BackgroundColor','white', ...
   'String','', ...
   'Callback','x1min = str2num(get(HT_X1L,' ...
      'String''));');

HL_X1M = uicontrol(Hfig_1, ...
   'Style','text', ...
   'Units','inches', ...
   'Position',[1.5,0.8,0.6,.2], ...
   'String','increment');

HT_X1M = uicontrol(Hfig_1, ...
   'Style','edit', ...
   'Units','inches', . . .
   'Position',[2.2,0.8,0.7,.2], ...
   'BackgroundColor','white', ...
   'String','', ...
   'Callback','x1inc = str2num(get(HT_X1M,' ...
      'String''));');

HL_X1U = uicontrol(Hfig_1, ...
   'Style','text', ...
   'Units','inches', ...
   'Position',[3.0,0.8,0.5,.2], ...
   'String','x1(max)');

HT_X1U = uicontrol(Hfig_1, ...
   'Style','edit', ...
   'Units','inches', ...
   'Position',[3.6,0.8,0.7,.2], ...
```

```
    'BackgroundColor','white', ...
    'String','', ...
    'Callback','x1max = str2num(get(HT_X1U,' ...
       'String''));');

HL_X2L = uicontrol(Hfig_1, ...
    'Style','text', ...
    'Units','inches', ...
    'Position',[0.1,0.5,0.5,.2], ...
    'String','x2(min)');

HT_X2L = uicontrol(Hfig_1, ...
    'Style','edit', ...
    'Units','inches', ...
    'Position',[0.7,0.5,0.7,.2], ...
    'BackgroundColor','white', ...
    'String','', ...
    'Callback','x2min = str2num(get(HT_X2L,' ...
       'String''));');

HL_X2M = uicontrol(Hfig_1, ...
    'Style','text', ...
    'Units','inches', ...
    'Position',[1.5,0.5,0.6,.2], ...
    'String','increment');

HT_X2M = uicontrol(Hfig_1, ...
    'Style','edit', ...
    'Units','inches', ...
    'Position',[2.2,0.5,0.7,.2], ...
    'BackgroundColor','white', ...
    'String','', ...
    'Callback','x2inc = str2num(get(HT_X2M,' ...
       'String''));');

HL_X2U = uicontrol(Hfig_1, ...
    'Style','text', ...
    'Units','inches', ...
    'Position',[3.0,0.5,0.5,.2], ...
    'String','x2(max)');

HT_X2U = uicontrol(Hfig_1, ...
    'Style','edit', ...
    'Units','inches', ...
```



```
    'Position',[3.6,0.5,0.7,.2], ...
    'BackgroundColor','white', ...
    'String','', ...
    'Callback','x2max = str2num(get(HT_X2U,' ...
       'String''));');

Hlabel = uicontrol(Hfig_1, ...
    'Style','text', ...
    'Units','inches', ...
    'Position',[0.1,0.1,2.2,.2], ...
    'BackgroundColor','y', ...
    'String','After entering values please press ...
     EXIT');

Hpush = uicontrol(Hfig_1,'Style','push', ...
    'Units','inches', ...
    'Position',[2.5,0.1,0.7,.2], ...
    'BackgroundColor','red', ...
    'String','EXIT', ...
    'Callback','close');
uiwait
```

In the above code the statements should be easy to interpret. Much of the code can be edited after copy and paste operation. The last command `uiwait` instructs MATLAB to hold off execution until the user closes the GUI window. Figure 2.14 shows the image of the GUI as it appears in MATLAB. The text boxes are initialized with no values forcing the user to set values thereby causing all of the CallBack strings to be evaluated. In addition, the data entered by the user is a string. It has to be converted to a number. The code here is basic. It does not verify that the user did enter a number. This verification can be done by extending the CallBack string over multiple statements. This is left as an exercise in programming for the user.

The script m-file used to select and create the plots is as follows.

Plot2D.m

```
% Script file to go with Contour plotting
% Dr. P.Venkataraman, Applied Optimization Using MATLAB
```

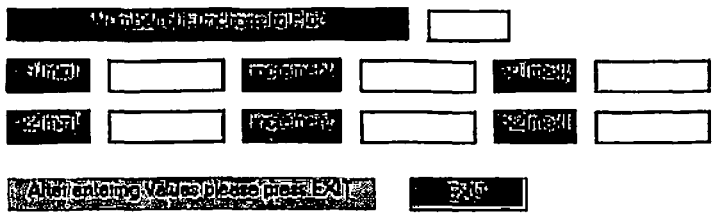

Figure 2.14 Image of GUI.

```
% Ch. 2, Sec 2.4
%
% script M file for 2D Contour Plotting of functions
% involved in optimization
% All functions are expected to be available as
% MATLAB function m-files
% Functions will be selected through the input selection
% box
% Function input parameters will be meshed matrices
% based on the range of parameters available from GUI
%
% The Number of plots and range for the plot
% are obtained from running the
% user interface script GUI2_4 prior
% to the call to Plot2D.m
%

GUI2_4 % call the GUI

% XInfo, YInfo are introduced for introducing default
% values
Xinfo = [x1min x1max x1inc];
% the values x1min, x1max, x1inc are available in
% the Command workspace due to CallBack

if isempty(Xinfo)
   Xinfo = [-4 4 0.05] % default
end
fprintf('\n')

Yinfo = [x2min x2max x2inc];
if isempty(Yinfo)
   Yinfo = [-4 4 0.05]
end
fprintf('\n')
xvar = Xinfo(1):Xinfo(3):Xinfo(2); % x1 vector
yvar = Yinfo(1):Yinfo(3):Yinfo(2); % x2 vector
[X1, X2]= meshgrid(xvar,yvar);     % matrix mesh

% set default number of plots to 1
if isempty(Np)
   Np = 1;
end
clf;    % clear figure
```



```
   for Mplot = 1:Np
   text1 = ['The function which is being plotted must ...
   '\nbe a MATLAB' . . .
   '\nfunction M - File. Given a meshed matrix input' ...
   '\nit must return a Matrix' . . .
   '\nPlease select function name in the dialog box ...
   and hit return : \n '];
   % this prints the text to the screen in the command
   % window and serves as a prompt
   % the \n is a new line command

   fprintf(text1)

   % using the uigetfile dialog box
   [file, path] = uigetfile('c:\*.m','Files of type ...
   MATLAB m-file',300,300);

   % check if file is string
   % strip the .m extension from the file so it can be
   % called by the program
     if isstr(file)
      functname = strrep(file,'.m','');
     else
      fprintf('\n\n');
   text2 = [' You have chosen CANCEL or the file was ...
   '\nnot acceptable' ...
   '\nThe program needs a File to Continue' . . .
   '\nPlease call Plot2D again and choose a file OR ' ...
   '\npress the up-arrow button to scroll through ...
   '\nprevious commands \n\n' . . .
   'Bye !'];
   error(text2);
   end

clear text1 text2; % clears the variables text1 ...
% and text2 for reuse
clear Fun maxval minval strcon convalue onevalue ...
  labcont labcontU
  Fun = feval(functname,X1,X2);
  maxval = max(max(Fun));
  minval = min(min(Fun));
  fprintf('The contour ranges from MIN: :%12.3f MAX. : ...
%12.3f ',minval,maxval);
  fprintf('\n');
  strcon = input('Do you want to set contour values ?...
```

```
[ no]:','s');
 strconU = upper(strcon);
 if strcmp(strconU,'YES') | strcmp(strconU,'Y')
  fprintf(' Input a vector of contour levels ');
  fprintf('\n')
  fprintf('between %10.2f and %10.2f ',minval,maxval);
   fprintf('\n')
   convalue = input(' Input contour level as a Vector :');
    labcont = input('Do you want labelled contours ? ...
[ no]:','s');
    labcontU = upper(labcont);
     if strcmp(labcontU,'YES')| strcmp(labcontU,'Y')
      [C,h] = contour(xvar,yvar,Fun,convalue);
      clabel(C,h);
     else
      contour(xvar,yvar,Fun,convalue);
     end

else
  ncon = input('Input number of contours [20] :');
  if isempty(ncon)
   ncon = 20;
   labcont = input('Do you want labelled contours ? ...
[ no]:','s');
   labcontU = upper(labcont);
   if strcmp(labcontU,'YES')| strcmp(labcontU,'Y')
     [C,h] = contour(xvar,yvar,Fun,ncon);
     clabel(C,h);
   else
     contour(xvar,yvar,Fun,ncon);
   end
elseif ncon == 1
   onevalue = input('Input the single contour level : ');
   labcont = input('Do you want labelled contours ?  ...
[ no]:','s');
   labcontU = upper(labcont);
     if strcmp(labcontU,'YES')| strcmp(labcontU,'Y')
   [C,h] = contour(xvar,yvar,Fun,[onevalue,onevalue]);
       clabel(C,h);
     else
       contour(xvar,yvar,Fun,[onevalue,onevalue]);
   end
else
```



```
    labcont = input('Do you want labelled contours ? [...
no]:','s');
     labcontU = upper(labcont);
     if strcmp(labcontU,'YES')| strcmp(labcontU,'Y')
       [C,h] = contour(xvar,yvar,Fun,ncon);
       clabel(C,h);
  else
        contour(xvar,yvar,Fun,ncon);
      end

   end
  end
  if Np > 1
     hold on;
   end
     Hf = gcf;
  end

  figure(Hf);
  grid
  hold off
```

In the above the plotting commands have been used before. The new MATLAB commands are `isempty`, `clf`, `isstr`, `strrep`, `max`, `min`, `upper`, `strcmp`, and `fprint`. Use the help command to learn more about these. Run the script file and understand the sequence of actions as well as the prompts regarding the contour. In this example the functions that create the plot do not need global values for calculation transferred from the command workspace. In other words the functions that are to be plotted can be calculated independently.

REFERENCES

1. *MATLAB The Language of Technical Computing: Using MATLAB*, Version 5, MathWorks Inc., 1998.
2. *MATLAB The Language of Technical Computing: Using MATLAB Graphics*, Version 5, MathWorks Inc., 1996.
3. Bohachevsky, I. O., Johnson, M. E., and Stein, M. L., Generalized Simulated Annealing for Function Optimization, *Technometrics*, Vol. 28, No. 3, 1986.
4. Arora, J. S., *Introduction to Optimum Design*, McGraw–Hill, New York, 1989.
5. Beer, F. P., and Johnston, E. R., Jr., *Mechanics of Materials*, 2nd ed., McGraw–Hill, New York, 1992.
6. Fox, R. W., and McDonald, A. T., *Introduction to Fluid Mechanics*, 4th ed., Wiley, New York, 1992.
7. Arpaci, V. S., *Conduction Heat Transfer*, Addison–Wesley, Reading, MA, 1966.

8. Hanselman, D., and Littlefield, B., *Mastering Matlab 5, A Comprehensive Tutorial and Reference*, The MATLAB Curriculum Series, Prentice–Hall, Englewood Cliffs, NY, 1996.
9. Building GUIs with MATLAB, online reference in pdf format, MathWorks Inc., 1998.

PROBLEMS

(Many of the graphical enhancements are now possible through plotedit features in later releases of MATLAB. The following problems are just suggestions. The suggested problems use additional graphical elements, and also require deleting handles. Please consult MATLAB documentation.)

2.1 Plot the ballistic trajectory from simple two-dimensional mechanics.

2.2 Produce an animated display of the trajectory.

2.3 Create a two-dimensional program that will display a random firing location and a random target location. Allow the user to choose initial velocity magnitude and angle. Plot his trajectory and calculate his error.

2.4 Create a three-dimensional program that will display a random firing location and a random target location. Allow the user to choose initial velocity magnitude and direction angles. Plot his trajectory and calculate his error.

2.5 Draw the boundary layer profile of laminar flow over a flat plate. Draw lines indicating velocity profile at ten points in the boundary layer.

2.6 Create a program that will plot the velocity profile for user-specific inputs. Allow the user to express his inputs through a dialog box.

3

LINEAR PROGRAMMING

The major part of this book deals with mathematical models that are characterized by nonlinear equations, like most of the examples used for illustration thus far. The presence of a single nonlinear equation in the model is sufficient to identify the problem from the class of nonlinear programming (NLP) problems. Mathematical programming is another term that is used to describe such models and their solution techniques. Most of engineering design falls in this category. Having represented such problems graphically in the previous chapter, it appears that the gradient and the curvature of the functions in the mathematical model had a significant impact on identifying the solution, even though no effort was made to point out this feature specifically. In fact, graphical solution was obtained by inspection rather than determined by mathematical relations or entities.

There is an equally important class of problems whose mathematical model is made up exclusively of functions that are only linear, such as Example 1.3, as modeled in Equations 1.21–1.24. The same example was modified in Equations 1.25–1.28 so that a graphical solution could be discussed. It is apparent from the graphical description in Figure 1.6 that there is no curvature evident in the figure. These problems are termed *linear programming* (LP) problems. They are natural in the subject of operations research, which covers a vast variety of models used for several kinds of decision making. Example 1.3 does represent a decision-making problem. Typically, LP can be a course by itself (or several courses for that matter). A significant portion of such a course would be to develop mathematical models from different areas of applications, as there is usually one numerical technique that is commonly used to solve LP problems. It is called the simplex method and is based on the algorithm by Dantzig [1]. The method involves mostly elementary row operations typically encountered in Gauss-elimination type methods that are part of the numerical

techniques used in linear algebra. In this chapter, only a limited, but useful discussion is presented. First, it is important to understand LP problems, the modeling issues, and their solution, as they are different from most of the other problems in this book. Second, many of the current numerical techniques for NLP problems obtain their solution by linearizing the solution at the current design point. These linearized equations can be solved by the methods of this chapter. The reader is directed to any of the books on LP for a more detailed description of the models and the techniques as the presentation in this book is simple and brief. References 1–3 are useful for this purpose.

3.1 PROBLEM DEFINITION

Example 1.3 will be used to define the terminology associated with LP problem. The modified version of the problem is used to develop the format of the LP problem so that we can follow through with the graphical solution. The mathematical model has two design variables x_1, x_2 that represent the number of Component Placement Machines of type A and B, respectively. The objective is to maximize the number of boards to be manufactured. Constraint g_1 represents the acquisition dollars available. Constraint g_2 represents the floor space constraint. Constraint g_3 represents the number of operators available.

$$\text{Maximize} \quad f(\mathbf{x}): 990\,x_1 + 900\,x_2 + 5250 \tag{3.1}$$

$$\text{Subject to:} \quad g_1(\mathbf{x}): 0.4\,x_1 + 0.6\,x_2 \le 8.5 \tag{3.2}$$

$$g_2(\mathbf{x}): 3\,x_1 - x_2 \le 25 \tag{3.3}$$

$$g_3(\mathbf{x}): 3\,x_1 + 6\,x_2 \le 70 \tag{3.4}$$

$$x_1 \ge 0; \quad x_2 \ge 0$$

The problem defined in Equations (3.1)–(3.4) is a natural representation of the mathematical model in engineering design or any other discipline. By allowing equality (=) constraints and greater than or equal (≥) constraints the problem could accommodate any mathematical model that is characterized by linear functions. This representation can be considered the inequality form of the mathematical model although it is not expressed in the *standard format* for the LP problem.

3.1.1 Standard Format

The standard format of the LP problem includes only equality constraints. It is set up as a minimization problem (some authors prefer a maximization problem). In addition, all of the variables in the model are expected to be semipositive (≥ 0) or nonnegative [4]. Finally, the constraint limits, the quantity on the right-hand side of

the constraint, are required to be positive (> 0). In the problem defined in Equations (3.1)–(3.4), the changes that must be carried out include converting the objective function to the opposite type, and transforming the constraints to the equality type. The variables x_1 and x_2, representing the number of Component Placement Machines of model A and B, respectively, can be expected to be semipositive (≥ 0), that is, either some machines will be ordered or no machines will be ordered. Note that the discrete nature of the problem is being ignored, as 3.75 machines of Type A is not something that would be ordered. The solution will be rounded to a suitable integer. There are extensive discussions in discrete and integer programming literature to suggest that this may not be the optimum value. The justification for rounding off is that it is convenient to do so at this juncture.

In the objective function $f(\mathbf{x})$, the coefficients 990 and 900 are called the cost coefficients. Each coefficient that is associated with a design variable represents the increase (decrease) in cost per unit change in the related variable. The simplest way to transform the objective function to the required format is to multiply all of the terms by −1. Therefore, the new objective function is

$$\text{Minimize} \quad f(\mathbf{x}): -990\,x_1 - 900\,x_2 - 5250 \tag{3.5}$$

In Equation (3.5), the constant term on the right (−5250) can be absorbed into the left-hand side by defining a new objective function $f(\mathbf{x})$ without affecting the optimal values for the design variables. It can be observed that increasing the values of the variables will make the objective more negative, which is good as we are trying to make $f(\mathbf{x})$ as low as possible. If a particular cost coefficient had a positive sign, then increasing the amount of the corresponding variable would lead to an increase of the objective function, which is not desirable if the function needs to be minimized. This idea is exploited in the Simplex method presented later.

The inequality constraints have to be transformed to an equality constraint. The simplest way to achieve this change is to introduce an additional semipositive variable for each inequality constraint. That will create the corresponding equality constraint in a straightforward way. Consider the first constraint:

$$g_1(\mathbf{x}): 0.4\,x_1 + 0.6\,x_2 \le 8.5 \tag{3.2}$$

This can be transformed to

$$g_1(\mathbf{x}): 0.4\,x_1 + 0.6\,x_2 + x_3 = 8.5 \tag{3.6}$$

The new variable x_3 is referred to as the *slack variable*. The definition is quite appropriate because it takes up the slack of the original constraint terms when it is less than 8.5. The slack will be positive as the first two terms will be less than or equal to 8.5 (for constraint satisfaction) from the original definition in Equation (3.2). In other words, it is the difference between the constraint limit (8.5) and the first two terms of the constraint. If the values of x_1 and x_2 cause the constraint to be at its limit value, then x_3 will be zero. This is the reason it is defined to be semipositive. By definition,

therefore, x_3 is similar to the original variables in the LP problem. Transforming the remaining constraints in a similar manner, the complete equations in standard format are

$$\text{Minimize} \quad f(\mathbf{x}): -990\,x_1 - 900\,x_2 - 5250 \tag{3.5}$$

$$\text{Subject to:} \quad g_1(\mathbf{x}): 0.4\,x_1 + 0.6\,x_2 + x_3 = 8.5 \tag{3.6}$$

$$g_2(\mathbf{x}): 3\,x_1 - x_2 + x_4 = 25 \tag{3.7}$$

$$g_3(\mathbf{x}): 3\,x_1 + 6\,x_2 + x_5 = 70 \tag{3.8}$$

$$x_1 \geq 0; \quad x_2 \geq 0; \quad x_3 \geq 0; \quad x_4 \geq 0; \quad x_5 \geq 0 \tag{3.9}$$

Equations (3.5)–(3.9) express the LP problem in *standard format*. An inconsistency can be observed with respect to the format of the problems previously defined in the book. Namely, $g(\mathbf{x})$ was reserved to define inequality constraints while $h(\mathbf{x})$ defined equality constraints. Here, $g(\mathbf{x})$ is also being used for equality constraints. This change is accommodated in this chapter only. In subsequent chapters the prior meaning for the symbols is restored. Actually, symbols are not required as the constraint under discussion can also be referred to by equation number. The formulation in Equations (3.5)–(3.9) can be expressed succinctly by resorting to a matrix notation. This conversion is quite straightforward. Before embarking on this transformation, the representation of the objective function needs to be changed again. The constant term is brought to the right-hand side so that the objective function can be expressed with terms only involving the design variables on the right. This is shown as $f\_(\mathbf{x})$ below. Once again for convenience the same symbol $f(\mathbf{x})$ is used in place of this new function in subsequent discussions. These extra symbols have been thrown in to keep the derivation distinct. This is a small price for the ability to develop the mathematical model in a natural way and reconstruct it to conform to the standard representation in most references. Therefore:

$$\text{Minimize} \quad f\_(\mathbf{x}): f(\mathbf{x}) + 5250 = -990\,x_1 - 900\,x_2 \tag{3.5}$$

The following definitions are used to rewrite the problem using matrices. Let the cost coefficients be represented by $c_1, c_2, \ldots, c_n$. The coefficients in the constraints are represented by the symbol a_{ij} (using double subscripts). The first subscript identifies the constraint number while the second constraint is the same as the variable it is multiplying. It is also useful to identify the first subscript with the row number and the second subscript with the column number to correspond with operations that use linear algebra. For example, the constraint $g_1(\mathbf{x})$ will be represented as

$$g_1(\mathbf{x}): a_{11}\,x_1 + a_{12}\,x_2 + a_{13}\,x_3 = 8.5$$

where $a_{11} = 0.4$, $a_{12} = 0.6$, $a_{13} = 1$. Because there are five design variables in the problem definition, additional terms with the following coefficients, a_{14} (= 0) and a_{15}(= 0), are included in the expression for $g_1(\mathbf{x})$ without changing its meaning:

$$g_1(\mathbf{x}): a_{11}\,x_1 + a_{12}\,x_2 + a_{13}\,x_3 + a_{14}\,x_4 + a_{15}\,x_5 = 8.5$$

The constraint $g_2(\mathbf{x})$ can be similarly expressed as

$$g_2(\mathbf{x}): a_{21}\,x_1 + a_{22}\,x_2 + a_{23}\,x_3 + a_{24}\,x_4 + a_{25}\,x_5 = 25$$

The coefficients in the above equation can be easily established by comparison with Equation (3.7). Letting the right-hand side values of the constraints be represented by $b_1, b_2, \ldots, b_m$, the standard format in Equations (3.5)–(3.9) can be translated to

$$\text{Minimize} \quad f(\mathbf{x}): \mathbf{c}^T\mathbf{x} \tag{3.10}$$

$$\text{Subject to:} \quad g(\mathbf{x}): \mathbf{A}\,\mathbf{x} = \mathbf{b} \tag{3.11}$$

$$\text{Side constraints:} \quad \mathbf{x} \geq 0 \tag{3.12}$$

x represents the column vector of design variables, including the slack variables, $[x_1, x_2, \ldots, x_n]^T$ (T represents the transposition symbol). c represents the column vector of cost coefficients, $[c_1, c_2, \ldots, c_n]^T$. b represents the column vector of the constraint limits, $[b_1, b_2, \ldots, b_m]^T$. A represents the $m \times n$ matrix of constraint coefficients. In this book all vectors are column vectors unless otherwise identified.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdot & a_{1n} \\ a_{21} & a_{22} & \cdot & a_{2n} \\ \cdot & \cdot & \cdot & \cdot \\ a_{m1} & a_{m2} & \cdot & a_{mn} \end{bmatrix}; \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ b_m \end{bmatrix}; \quad \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \cdot \\ c_n \end{bmatrix}; \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ x_n \end{bmatrix}$$

Comparing Equations (3.10) and (3.11) with Equations (3.5)–(3.8) the vectors and matrix are

$$\mathbf{c} = [-990 \quad -900 \quad 0 \quad 0 \quad 0]^T$$

$$\mathbf{b} = [8.5 \quad 25 \quad 70]^T$$

$$\mathbf{A} = \begin{bmatrix} 0.4 & 0.6 & 1 & 0 & 0 \\ 3 & -1 & 0 & 1 & 0 \\ 3 & 6 & 0 & 0 & 1 \end{bmatrix}$$

The following additional definitions have been used in the standard format. n is the number of design variables and m is the number of constraints. In the example, $n = 5$ and $m = 3$. Note that the slack variables are included in the count for n.

Negative Values of Design Variables: The LP standard mathematical model allows only for nonnegative design variables. This does not limit the application of LP in any way. There is a standard procedure to handle negative values for variables. If a design variable, say x_1, is expected to be unrestricted in sign, then it is defined as the difference of two nonnegative variables

$$x_1 = x_1^h - x_1^l \tag{3.13}$$

where x_1^h and x_1^l are acceptable LP variables. If the latter is higher than the former, x_1 can have negative values. Equation (3.13) is used to replace x_1 in the mathematical model. Note that this procedure also increases the number of design variables in the model.

Type of Constraints: The less than or equal to ($\leq$) constraint was handled naturally in developing the LP standard mathematical model. The standard numerical technique for obtaining the solution requires that the slack variables be positive to start and b also be positive throughout. If there is a constraint that is required to be maintained above a specified value, for example

$$g_1(\mathbf{x}): a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \geq b_1 \tag{3.14}$$

then it appears natural to introduce a negative slack variable $-x_4$ to change Equation (3.14) to an equality constraint:

$$g_1(\mathbf{x}): a_{11}x_1 + a_{12}x_2 + a_{13}x_3 - x_4 \geq b_1$$

While it is possible to change the sign on x_4 by multiplying with -1 throughout, this causes the right-hand constraint limit to be negative, which should be avoided. In such cases, another variable, x_5, is included in the expression. In this case, two additional variables are now part of the model:

$$g_1(\mathbf{x}): a_{11}x_1 + a_{12}x_2 + a_{13}x_3 - x_4 + x_5 = b_1$$

The variable x_5 is called an *artificial variable*. Artificial variables are also introduced for each *equality* constraint that is naturally present in the model. The use of these variables will be illustrated later through an example.

3.1.2 Modeling Issues

The LP program is characterized by the mathematical model defined in Equations (3.10)–(3.12). Due to the linear nature of all of the functions involved, the discussion of the solution need only consider the coefficients of the functions—the variables

themselves can be ignored. These coefficients, which are present as vectors or a matrix, can be manipulated to yield solutions using concepts from linear algebra.

A look at the graphical solution to the LP problem suggests that the solution to the problem, if it exists, must lie on the constraint boundary because the curvature of the objective function is zero everywhere (it is linear). Furthermore, the solution to the LP problem is primarily influenced by Equation (3.11). Being a linear equation, topics from linear algebra are necessary to understand some of the implications of the mathematical model.

If $n = m$, that is, the number of unknowns is the same as the number of variables, there can be no scope for optimization as Equation (3.11) will determine the solution, provided one exists. In the following, many of the concepts are explained through simple graphical illustration, or by consideration of simple examples. They mostly deal with examples of two variables (x_1, x_2). For example, consider two constraints

$$g_1: \quad x_1 + x_2 = 2$$
$$g_2: \quad -x_1 + x_2 = 1$$

Figure 3.1 illustrates the graphical solution to the problem. The solution is at $x_1 = 0.5$, and $x_2 = 1.5$. This is independent of the objective function (whatever it may be). In matrix form, **A** and **b** are

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

The existence of a solution depends on the rows of **A** (or columns of **A**—we will deal with rows only for convenience). If the rows of **A** are *linearly independent*, then there is a *unique solution* to the system of equations.

Define another problem as follows (only changing g_2 from the above example). The new problem is

$$g_1: \quad x_1 + x_2 = 2$$
$$g_2: \quad 2x_1 + 2x_2 = 4$$

The new function g_2 represents twice the value of g_1. This means g_2 can be obtained from g_1. Therefore, g_2 is said to depend on g_1. Technically, the two equations are *linearly dependent*—the actual dependence of the functions is not a concern as much as the fact that they are not linearly independent. Both g_1 and g_2 establish the same line graphically. What is the solution to the problem? There are *infinite* solutions, as long as the pair of (x_1, x_2) values satisfies g_1. In Figure 3.1, any point on the line representing g_1 is a solution. Not only (0.5, 1.5), but (2,0), (0,2) are also solutions. From an optimization perspective, this is quite good for unlike the previous case of unique solution, the objective function can be used to find the best solution. It would be better, however, if the choices were finite rather than infinite.

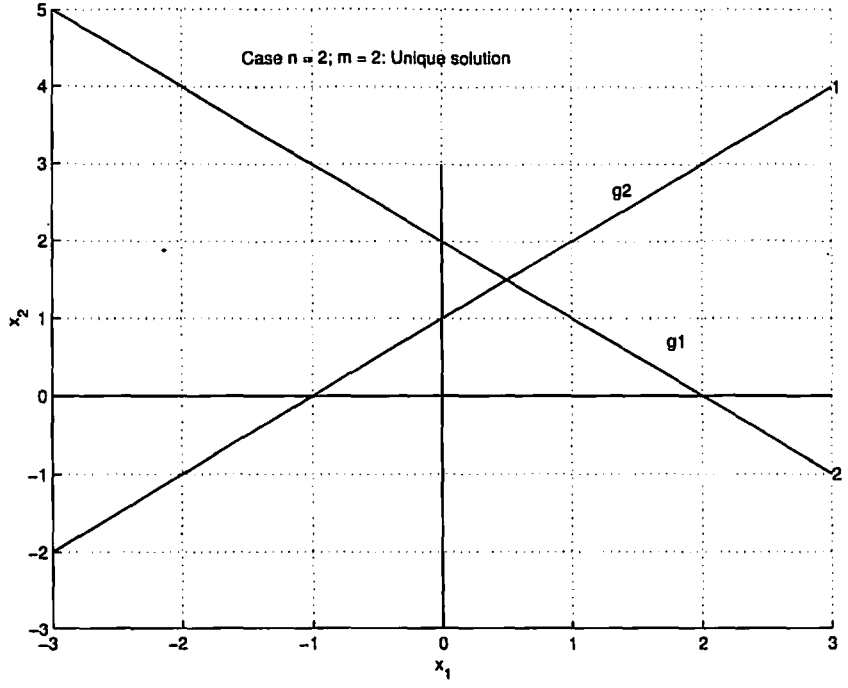

Figure 3.1 Unique solution—linear independence.

While several examples can be constructed to illustrate linear dependence, it appears reasonable to assume that *for a useful exercise in optimization, the number of constraints cannot equal the number of unknowns* ($m \neq n$). In the previous illustration, $n = 2$ and $m = 1$ (there was only one effective constraint). It is useful to recognize that this corresponds to the case $n > m$.

If $m > n$, there are more equations than the number of variables. This implies that the system of equations represented by Equation (3.11) is an inconsistent set or has a redundant set of equations. Consider the following illustration for $m = 3$ and $n = 2$ which uses the same g_1 and g_2 as in the first illustration while adding a new g_3:

$$g_1: \quad x_1 + x_2 = 2$$

$$g_2: \quad -x_1 + x_2 = 1$$

$$g_3: \quad x_1 + 2x_2 = 1$$

Figure 3.2 illustrates that the set of equations is *inconsistent* since a solution does not exist. If one were to exist, then the three lines must pass through the solution. Since they are all straight lines, there can be only one unique intersecting point. In Figure 3.2 only two of the three lines intersect at different points.

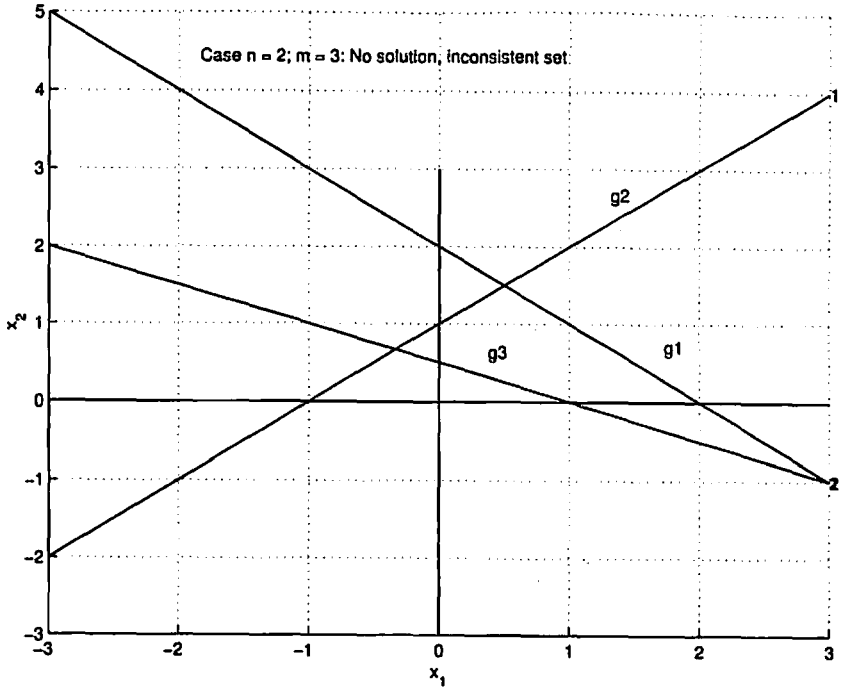

Figure 3.2 No solution, inconsistent set of equations.

Redefine g_3 as

$$g_3: x_1 + 2x_2 = 3.5$$

This new g_3 is the dashed line in Figure 3.3. Now a unique solution to the problem at (0.5, 1.5) is established. This is also the solution established by consideration of g_1 and g_2 alone. This implies that g_3 is *redundant*. If g_1 is multiplied by 1.5, g_2 is multiplied by 0.5 and both added, the result is g_3 defined above. That is,

$$g_3 = 1.5\, g_1 + 0.5\, g_2$$

g_3 can be obtained by *linearly* combining g_1 and g_2, that is, adding constant multiples of the functions. This is another example of linear dependence. This *linear dependence* and *redundancy* is also illustrated by the set

$$g_1: \quad x_1 + x_2 = 2$$

$$g_2: \quad -x_1 + x_2 = 1$$

$$g_3: \quad 2x_1 + 2x_2 = 4 \text{ (same as } g_1)$$

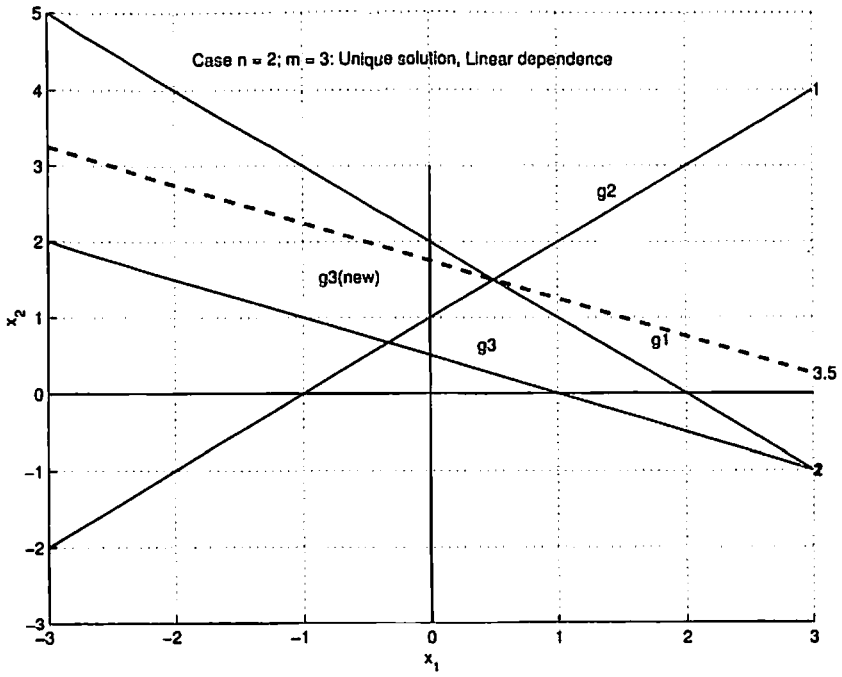

Figure 3.3 Unique solution, linear dependence.

This suggests that the concept of redundancy can be associated with linear dependence. The discussion of linear dependence and independence was established using equations above. The same discussion can take place by reasoning on the coefficients themselves. In this case the coefficient matrix **A** should lead us to the same conclusion regarding the linear independence of a set of linear equations. The concept of a *determinant* is necessary to develop the criteria for linear dependence.

Determinant: The determinant is associated with a square matrix. For a general 2×2 matrix **A**, where

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

the determinant is expressed and evaluated as

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = \det(\mathbf{A}) = a_{11} \times a_{22} - a_{12} \times a_{21} = a_{11}a_{22} - a_{12}a_{21} \tag{3.15}$$

For $n = m$, and with g_1: $x_1 + x_2 = 2$, and g_2: $-x_1 + x_2 = 1$, the matrix **A** is

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

and its determinant is

$$\det(\mathbf{A}) = |\mathbf{A}| = (1)*(1) - (1)*(-1) = 2$$

From theorems in linear algebra [5], if det(**A**) is *not zero*, in which case the matrix A termed *nonsingular*, a unique solution exists to the set of equations. This was true for the example illustrated above, as the only solution was located at (0.5, 1.5), shown graphically in Figure 3.1.

Consider the example with g_1: $x_1 + x_2 = 2$, and g_2: $2x_1 + 2x_2 = 4$. In this case,

$$\det(\mathbf{A}) = |\mathbf{A}| = (1)*(2) - (1)*(2) = 0$$

If det(**A**) is *zero*, that is, matrix A is *singular*, there are either *no solutions* or *infinite solutions*. For this example, there were infinite solutions. Determinants of higher-order square matrices are evaluated by setting up lower-order determinants until they are reduced to a 2×2 determinant, which is evaluated as above. Any textbook on engineering mathematics or linear algebra should illustrate this technique. It is not reproduced here.

If $n \neq m$, discussion of the existence of solutions requires additional concepts like *rank* of a matrix and *augmented* matrix. Since the case $n > m$ is of interest in optimization, only that case is employed in subsequent illustration. A useful discussion will need at least three variables and two equations. Three variables will deny the use of graphics to develop the following concepts. Using the set

$$\begin{aligned} g_1&: \quad x_1 + x_2 + x_3 = 3 \\ g_2&: \quad -x_1 + x_2 + 0.5\,x_3 = 1.5 \end{aligned}$$

the matrices are

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & 0.5 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 3 \\ 1.5 \end{bmatrix} \quad \mathbf{A}^* = \begin{bmatrix} 1 & 1 & 1 & 3 \\ -1 & 1 & 0.5 & 1.5 \end{bmatrix} \tag{3.16}$$

The new matrix $\mathbf{A}^*$ is called the *augmented* matrix—the columns of **b** are added to **A**. According to theorems of linear algebra (presented here without proof):

- If the augmented matrix ($\mathbf{A}^*$) and the matrix of coefficients (**A**) have the same rank $r < n$, then there are many solutions.

- If the augmented matrix ($\mathbf{A}^*$) and the matrix of coefficients ($\mathbf{A}$) do not have the same rank, a solution does not exist.
- If the augmented matrix ($\mathbf{A}^*$) and the matrix of coefficients ($\mathbf{A}$) have the same rank $r = n$, where $m = n$, then there is a unique solution.

Rank of a Matrix: While there are formal definitions for defining the rank of the matrix, a useful way to determine the rank is to look at the determinant. The rank of a matrix $\mathbf{A}$ is the order of the largest nonsingular square submatrix of $\mathbf{A}$, that is, the largest submatrix with a determinant other than zero.

In the example above, the largest square submatrix is a 2×2 matrix (since $m = 2$ and $m < n$). Taking the submatrix which includes the first two columns of $\mathbf{A}$, the determinant was previous established to have a value of 2, therefore nonsingular. Thus, the rank of $\mathbf{A}$ is 2 ($r = 2$). The same columns appear in $\mathbf{A}^*$ making its rank also 2. Therefore, infinitely many solutions exist ($r < n$). One way to determine the solutions is to assign $(n - r)$ variables arbitrary values and use them to determine values for the remaining r variables. The value $n - r$ is also often identified as the *degree of freedom* (DOF) for the system of equations. In the particular example above, the DOF can be identified with a value of 1 (i.e., 3 – 2). For instance, x_3 can be assigned a value of 1 in which case $x_1 = 0.5$ and $x_2 = 1.5$. On the other hand, if $x_3 = 2$, then $x_1 = 0.25$ and $x_2 = 0.75$. The above cases illustrate that for a DOF of 1, once x_3 is selected, x_1 and x_2 can be obtained through some further processing.

Another example for illustrating the rank is the inconsistent system of equations introduced earlier where $n = 2$ and $m = 3$, reproduced for convenience as

$$g_1: \quad x_1 + x_2 = 2$$
$$g_2: \quad -x_1 + x_2 = 1$$
$$g_3: \quad x_1 + 2x_2 = 1$$

The matrices for this system are

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \\ 1 & 2 \end{bmatrix}; \quad \mathbf{b} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}; \quad \mathbf{A}^* = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix}$$

The rank of $\mathbf{A}$ cannot be greater than 2, since $n = 2$ and is less than $m = 3$. From prior calculations the determinant of the first two rows of $\mathbf{A}$ is 2. This is greater than zero and therefore the rank of $\mathbf{A}$ is 2. For $\mathbf{A}^*$ the determinant of the 3×3 matrix has to be examined. The determinant of a general $n \times n$ square matrix $\mathbf{A}$ can be calculated in terms of the *cofactors* of the matrix. The cofactor involves the determinant of a *minor* matrix—which is an $(n-1) \times (n-1)$ matrix obtained by deleting an appropriate single row and a single column from the original $n \times n$ matrix $\mathbf{A}$, multiplied by 1 or –1, depending on the total of the row and column values. References 5 and 6 can provide more information. Here, the determinant for the 3×3 matrix is defined for illustration

in the book. The (+1) and (–1) below are obtained by using $(-1)^{i+j}$, where i and j represent the row and column values of the coefficient multiplying the value

$$\mathbf{A} = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = (+1)a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} + (-1)a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + (+1)a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} \tag{3.17}$$

Therefore,

$$|\mathbf{A}^*| = \begin{vmatrix} 1 & 1 & 1 \\ -1 & 1 & 1 \\ 1 & 2 & 1 \end{vmatrix} = 1 \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} - 1 \begin{vmatrix} -1 & 1 \\ 1 & 1 \end{vmatrix} + 1 \begin{vmatrix} -1 & 1 \\ 1 & 2 \end{vmatrix}$$

$$= (1)(0) + (-1)(-2) + (1)(-3) = -1$$

$\mathbf{A}^*$ is nonsingular and its rank is 3 while the rank of $\mathbf{A}$ is 2. The system of equations will have no solutions from the above theorem and can be seen graphically in Figure 3.2. For the sake of completeness, using the previously defined dependent set of equations:

$$g_1: \quad x_1 + x_2 = 2$$
$$g_2: \quad -x_1 + x_2 = 1$$
$$g_3: \quad 2x_1 + 2x_2 = 4$$

which yields the following matrices:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \\ 2 & 2 \end{bmatrix}; \quad \mathbf{b} = \begin{bmatrix} 2 \\ 1 \\ 4 \end{bmatrix}; \quad \mathbf{A}^* = \begin{bmatrix} 1 & 1 & 2 \\ -1 & 1 & 1 \\ 2 & 2 & 4 \end{bmatrix}$$

The rank of $\mathbf{A}$ is 2. Note that using the first and last row of $\mathbf{A}$ for the calculation of the determinant yields a *singular* 2×2 submatrix. The determinant of $\mathbf{A}^*$ is

$$|\mathbf{A}^*| = \begin{vmatrix} 1 & 1 & 2 \\ -1 & 1 & 1 \\ 1 & 2 & 4 \end{vmatrix} = (1) \begin{vmatrix} 1 & 1 \\ 2 & 4 \end{vmatrix} + (-1) \begin{vmatrix} -1 & 1 \\ 2 & 4 \end{vmatrix} + (2) \begin{vmatrix} -1 & 1 \\ 2 & 2 \end{vmatrix} = 2 + 6 - 8 = 0$$

which makes $\mathbf{A}^*$ singular. However, the rank of $\mathbf{A}^*$ is at least 2 due to the presence of the same submatrix which determined the rank of $\mathbf{A}$. From the above theorem, the rank of $\mathbf{A}$ is equal to the rank of $\mathbf{A}^*$, which is 2 and this is less than $m = 3$. It can be concluded that there are *infinite* solutions for the system of equations.

A useful observation of matrix $\mathbf{A}^*$ in the preceding example is that two of the rows are a constant multiple of each other. This makes the matrices singular. The same holds for two columns that are multiples of each other.

Unit Vectors: Real-world optimization problems frequently involve a large number of design variables (*n variables*). In real LP problems, the number of variables can be staggering. For a successful solution to the optimization problem, it is implicitly assumed that the set of design variables is linearly independent. That is, there is no direct dependence among any two variables of the set. In this case, the *abstract space* spawned by the design variables is called the *n-dimensional Euclidean space*. The word *abstract* is used here to signify that it is a construct in the imagination, as only three variables can be accommodated in the familiar 3D physical space. The design vector is a point in this space. The Euclidean space is a direct extrapolation of the physical, geometrical 3D space that surrounds us. This space is termed the *Cartesian* space. For an illustration, consider the point (2,3,2) in Cartesian space or in the *rectangular* coordinate system. In Figure 3.4, the point is marked P. The triad of numbers within parentheses denotes the values along the x_1, x_2, and x_3 axes, respectively.

The Cartesian space is identified by a set of mutually perpendicular lines or *axes* marked x_1, x_2, and x_3. The point where they intersect is the origin (O) of the coordinate system. The point P can also be associated with the vector **OP**, drawn from the origin O to the point P. Very often a *point* and a *vector*—two very different entities—are used synonymously. From elementary vector addition principles, the vector **OP** can be constructed by consecutively joining three vectors along the three coordinate axes. In the figure this is shown by laying out a vector along the x_1 axis, adding a vector along the x_2 axis to it, and finally adding the third vector along the x_3 axis. The additional small arrows in the figure are the *unit* vectors. They have a magnitude of *one* and a direction along each of the axes. These unit vectors are labeled e_1, e_2, and e_3 in the figure. Mathematically, the point P (2,3,2) and the vector addition can be represented as (column vectors)

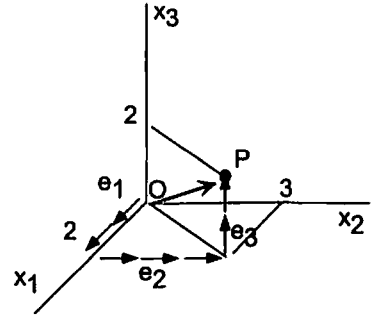

Figure 3.4 Rectangular coordinate system.

$$\begin{bmatrix}2\\3\\2\end{bmatrix} = 2\begin{bmatrix}1\\0\\0\end{bmatrix} + 3\begin{bmatrix}0\\1\\0\end{bmatrix} + 2\begin{bmatrix}0\\0\\1\end{bmatrix} = 2e_1 + 3e_2 + 2e_3 \tag{3.18}$$

The advantage of unit vectors is that they can establish any point in the Cartesian space through an appropriate triad of numbers signifying coordinate values or location. Actually, any point in the Cartesian space can be determined by *any set* of three linearly independent vectors. These vectors are termed the *basis* vectors. In this case, the simple connection between the point and the elementary addition, illustrated above, will not be available. Furthermore, through the methods of linear algebra, these three vectors can be reduced to the unit vectors through elementary row/column operations—also called *Gauss-Jordan elimination* or *reduction*. In n-dimensional Euclidean space spanned by the n design variables, the point $P_n(p_1, p_2, \ldots, p_n)$, will be represented by a collection of n values enclosed in parentheses. The corresponding **n** unit vectors are defined as

$$\mathbf{e}_1 = \begin{bmatrix}1\\0\\\cdot\\\cdot\\0\end{bmatrix}; \quad \mathbf{e}_2 = \begin{bmatrix}0\\1\\\cdot\\\cdot\\0\end{bmatrix}; \quad \mathbf{e}_j = \begin{bmatrix}0\\\cdot\\1\\\cdot\\0\end{bmatrix}; \quad \mathbf{e}_n = \begin{bmatrix}0\\0\\\cdot\\\cdot\\1\end{bmatrix} \tag{3.19}$$

The vector from the origin (O) to P_n is represented as

$$\mathbf{OP} = p_1\mathbf{e}_1 + p_2\mathbf{e}_2 + \ldots + p_n\mathbf{e}_n$$

These new definitions will be sufficient to understand the techniques used to solve LP problems.

3.2 GRAPHICAL SOLUTION

The graphical discussion is again limited to two design variables in the original problem description. It is used here to develop certain geometric ideas related to LP. In LP, the introduction of slack variables will easily exceed this limit of two variables for graphical illustration, so the graphical solution is illustrated with respect to the original variables in the design problem. MATLAB is once again used to draw the graphical solution. The problem is not reduced to the standard format for the graphical solution. It is important to do so for a numerical solution. Since the primary graphical element is the *straight line*, a function for drawing lines is developed through the function m-file drawLine.m below. This function is also used to provide more exposure to MATLAB programming.

drawLine.m: This is a function m-file for drawing straight lines. The line to be drawn is represented as

$$ax + by = c$$

Since x, y are used as symbols for variables, x_1 represents the lower range on x, while x_2 represents the upper range. Limiting lines/constraints can be drawn by setting either a or b to zero. Color (red) is used to draw a parallel line at 10% increase/decrease in the original value of c (depending on the direction of the original inequality) instead of hash marks. Limit constraints are in magenta and the objective function is in blue dashed line. The new MATLAB command here is the line command. You are encouraged to seek help on the command. The code:

```
drawLine.m
% Drawing linear constraints for LP programming problems
% Dr. P.Venkataraman
% Optimization Using Matlab
% Chapter 3 - linear Programming
%
% Lines are represented as: ax + by = c ( c >= 0)
% x1, x2 indicate the range of x for the line
% typ indicates type of line being drawn l (<=)
%                                        g (>=)
%                                        n (none)
%
% The function will draw line(s) in the figure window
% the green solid line represents the actual value
% of the constraint
% the red dashed line is 10 % larger or smaller
% (in lieu of hash marks)
% the limit constraints are identified in magenta color
% the objective function is in blue dashed lines
%
function drawLine(x1,x2,a,b,c,typ)
% recognize the types and set color
if (typ == 'n')
   str1 = 'b';
   str2 = 'b';
   cmult = 1;
else
   str1 = 'g';
   str2 = 'r';
end

% values for drawing hash marks
% depending on the direction of inequality
if (typ ~= 'n')
  if (typ == 'l')
   cmult = +1;
  else cmult = -1;
  end
end

% set up a factor for drawing the hash constraint
if (abs(c) >= 10)
  cfac = 0.025;
elseif (abs(c) > 5) & (abs(c) < 10)
  cfac = 0.05;
else
  cfac = 0.1;
end

if (c == 0)
  cdum = cmult*0.1;
else
  cdum = (1 + cmult* cfac)*c;
end

% if b = 0 then determine end points of line x line
if (b ~= 0)
 y1 = (c - a*x1)/b;
 y1n = (cdum - a* x1)/b;
 y2 = (c - a* x2)/b;
 y2n = (cdum - a*x2)/b;
else
 % identify limit constraints by magenta color
 str1 = 'm';
 str2 = 'm';
 y1 = x1; % set y1 same length as input x1
 y2 = x2; % set y2 same length as input x2
 x1 = c/a; % adjust x1 to actual value
 x2 = c/a; % adjust x2 to actual value
 y1n = 0; % set y = 0;
 y2n = 0; % set y = 0
end

if (a == 0)
 str1 = 'm'; % set color for limit line
 str2 = 'm'; % set color for limit line
end;
% draw axis with solid black color

hh = line([x1,x2],[0,0]);
```

```
set(hh,'LineWidth',1,'Color','k');

hv = line([0,0],[x1,x2]);
set(hv,'LineWidth',1,'Color','k');

% start drawing the lines

h1 = line([x1 x2], [y1,y2]);

if (typ == 'n')
  set(h1,'LineWidth',2,'LineStyle','--','Color',str1);
else
  set(h1,'LineWidth',1,'LineStyle','-','Color',str1);
end

if (b ~= 0)&(a ~= 0)
  text(x2,y2,num2str(c));
end
if( b == 0)|(a == 0)| (typ == 'n')
  grid
  return, end
grid;
h2 = line([x1 x2], [y1n,y2n]);
set(h2,'LineWidth',0.5,'LineStyle',':','Color',str2);
grid
hold on
```

3.2.1 Example 3.1

The problem from page 94 is reproduced once more for convenience.

$$\text{Maximize} \quad f(\mathbf{X}): 990\,x_1 + 900\,x_2 + 5250 \tag{3.1}$$

$$\text{Subject to:} \quad g_1(\mathbf{X}): 0.4x_1 + 0.6x_2 \le 8.5 \tag{3.2}$$

$$g_2(\mathbf{X}): 3x_1 - x_2 \le 25 \tag{3.3}$$

$$g_3(\mathbf{X}): 3x_1 + 6x_2 \le 70 \tag{3.4}$$

$$x_1 \ge 0; \quad x_2 \ge 0$$

The graphical solution is shown in Figure 3.5. The solution, read from the figure (you can zoom the area of solution), is $x_1^* = 10$, and $x_2^* = 7$. It is the intersection of the active constraints (3.3) and (3.4). The actual values are $x_1^* = 10.48$ and $x_2^* = 6.42$. Since an integral number of machines have to be ordered, the solution is adjusted to a neighboring integer value that satisfies the contraints.

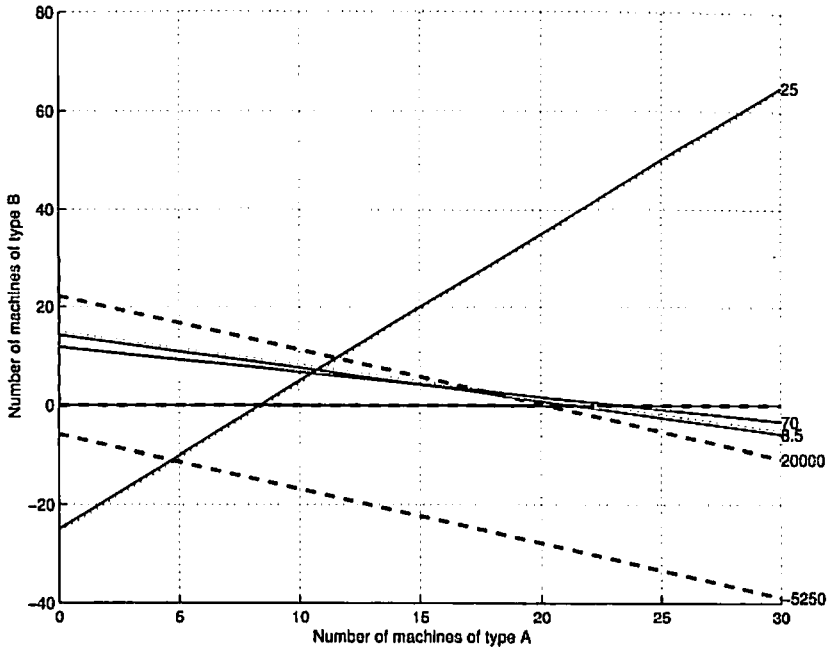

Figure 3.5 Graphical solution, Example 3.1.

3.2.2 Characteristics of the Solution

The geometry evident in the graphical solution of Example 3.1 is used to explain some of the concepts associated with LP and its numerical technique. Figure 3.6 is the graphical representation of the constraints involved in Example 3.1 (same as Figure 3.5 without the objective function). In order to relate the geometry to LP concepts, the standard format of LP is necessary. The standard format established before is

$$\text{Minimize} \quad f(\mathbf{X}): -990x_1 - 900x_2 - 5250 \tag{3.5}$$

$$\text{Subject to:} \quad g_1(\mathbf{X}): 0.4x_1 + 0.6x_2 + x_3 = 8.5 \tag{3.6}$$

$$g_2(\mathbf{X}): 3x_1 - x_2 + x_4 = 25 \tag{3.7}$$

$$g_3(\mathbf{X}): 3x_1 + 6x_2 + x_5 = 70 \tag{3.8}$$

$$x_1 \ge 0; \quad x_2 \ge 0; \quad x_3 \ge 0; \quad x_4 \ge 0; \quad x_5 \ge 0 \tag{3.9}$$

In Figure 3.6 the constraints $x_1 \ge 0$ and $x_2 \ge 0$ are added to the three functional constraint lines (3.6)–(3.8). The hashed area is the feasible region, that is, the design

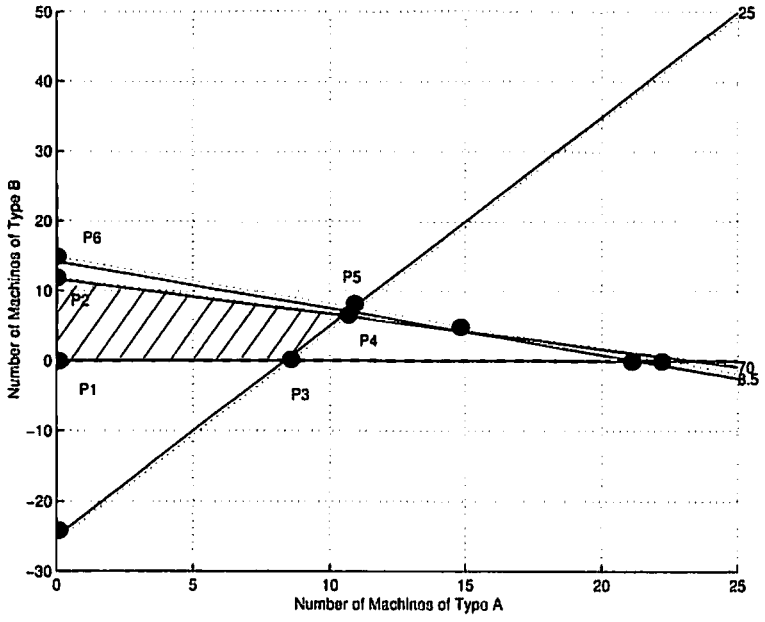

Figure 3.6 Feasible region, Example 3.1.

space in which all of the constraints are satisfied. The circles exaggerate the points of intersection of five constraints taken two at a time. Many of them are numbered P1, P2, . . . , P6. All of these points of intersection can be associated with a certain property concerning the values of the variables. Note there are five design variables ($n = 5$) and three functional constraints ($m = 3$). For example:

$$\text{P1: } (x_1 = 0, x_2 = 0, x_3 = 8.5, x_4 = 25, x_5 = 70)$$

$$\text{P2: } (x_1 = 0, x_2 = 11.67, x_3 = 1.5, x_4 = 36.67, x_5 = 0)$$

$$\text{P5: } (x_1 = 10.7, x_2 = 7.05, x_3 = 0, x_4 = 0, x_5 = -4.4)$$

$$\text{P6: } (x_1 = 0, x_2 = 14.17, x_3 = 0, x_4 = 39.17, x_5 = -15.42)$$

The values of the variables are obtained as the intersection of the constraints taken two at a time. In the above list, for each point, exactly two of the variables are zero. The number 2 corresponds to the value of $n - m$. n represents the number of variables and m the number of constraints. Points P5 and P6 are infeasible because one of the variables has a negative value.

Basic Solution: A basic solution is one obtained by setting exactly $n - m$ variables to zero. In Figure 3.6, all of the points identified by the circles represent basic solutions. The points chosen above are all basic variables. In general, for n design variables and m constraints, the number of basic solutions is given by the combination

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

This would yield 10 points for our example, which is shown in Figure 3.6.

Basic Variables: The set of variables in the basic solution that have nonzero values are called *basic variables*. Correspondingly, the set of variables in the basic solution that have the value of zero are called *nonbasic variables*. For the point P1, x_1 and x_2 are nonbasic variables while x_3, x_4, and x_5 are basic variables.

Basic Feasible Solution: This is a basic solution that is also feasible. These are the points P1, P2, P3, and P4 in Figure 3.6. In LP, the solution to the problem, if it is unique, must be a basic feasible solution. The basic solution can also be considered geometrically as a *corner* point or an *extreme* point of the feasible region.

Convex Polyhedron: This is a bounded region of the feasible design space—the region defined by the quadrilateral comprising the points P1, P2, P3, and P4 in Figure 3.6. The term *convex set* represents a collection of points or vectors having the following property: For any two points in the set (or within a region), if all of the points on the line connecting the two points also lie in the same region, the region is a convex set. Imagine any line drawn in the region defined by the quadrilateral whose corner points are P1, P2, P3, and P4. From Figure 3.6 it is clear the line will still be within the region established by the quadrilateral— making it a convex set.

Optimum Solution: This is a feasible solution that minimizes the objective function—point P4—in Figure 3.6. In LP, the optimum solution must be a basic feasible solution.

Basis: The basis represents the columns of the coefficient matrix **A** that correspond to the basic variables. They form the basis of the m-dimensional space. They are termed the *basis vectors*.

Canonical Form: The basis vectors reduced to unit vectors through row/column operations (or Gauss-Jordan elimination). The basic feature of the numerical technique for LP (Simplex method) is a repetitive procedure starting from an initial basic feasible solution, and determining the best neighboring basic feasible solution that improves the objective. The procedure is carried on until the optimum solution is reached, or if it is determined that no solution is possible. The canonical form is used for rapidly identifying the solution. Each iteration in the procedure can be described mathematically as follows.

The starting constraints are organized as

$$[\mathbf{A}]_{m\times n}[\mathbf{X}]_{n\times 1} = [\mathbf{b}]_m \tag{3.20}$$

After the Gauss-Jordan elimination, Equation (3.20) is assembled as

$$[\mathbf{I}]_{m\times m}[\mathbf{X}]_m + [\mathbf{R}]_{m\times(n-m)}[\mathbf{X}]_{(n-m)} = [\mathbf{b}]$$

The set of m design variables in the first term are the basic variables. The set of $(n - m)$ design variables in the second term are the nonbasic values. In the Simplex method, the nonbasic variables are summarily set to zero.

3.2.3 Different Solution Types

There are at most four different results that can be expected for the solution of the LP problem: (1) a unique solution, (2) infinitely many solutions, (3) unbounded solution, and (4) the possibility that there is no solution.

(1) *Unique solution:* The example used for discussion has a unique solution (Figure 3.5). The condition necessary for this to occur is that the objective function and the constraints have dissimilar slopes, and the feasible region is bounded/closed. Geometrically, this can be visually explained as the movement of the line representing the objective function, parallel to itself, in a favorable direction, until it remains just in contact with one of the feasible corners (basic feasible solution) of the feasible region. The feasible region is identified distinctly in Figure 3.6. A simple exercise of imagining different objective function lines should convince the viewer that several different objectives can be defined to locate the unique solution at P1, P2, P3, or P4. This 2D geometry and construction can be used to understand the extension to n dimensions. Here the lines will be represented by *hyperplanes*.

(2) *Infinite solution:* In order for this to occur, the objective function must be parallel to one of the constraints. For example, in Figure 3.6, let the original problem be redefined so that the objective function is parallel to the constraint g_3. Any point on the constraint, and lying between the line segment defined by the points P2 and P4, is an optimal solution to the problem. They will yield the same value of the objective.

(3) *Unbounded solution:* In this case, the feasible region is not bounded. In Figure 3.6, if the constraints g_1 and g_2 were not part of the problem formulation, then the feasible region is not bound on the top. Referring to Figure 3.5, the objective function can be shifted to unlimited higher values. In practice, there will be an upper bound on the range of the design variables (not part of standard format) that will be used to close this region, in which case solutions of type (1) or (2) can be recovered. The presence of an unbounded solution also suggests that the formulation of the problem may be lacking. Additional meaningful constraint(s) can be accommodated to define the solution.

(4) *No solution:* Figure 3.6 is used to explain this possibility. Consider that the direction of inequality in g_1 is changed to the opposite type ($\geq$). The feasible region with respect to the constraints g_1 and g_3 is to the right of $x_1 = 15.0$. This point is not feasible with respect to the constraint g_2. Therefore, there is no point that is feasible. There is no solution to the problem.

In the above discussion, the two-variable situations provided an obvious classification of the solutions. In practice, LP models are large with over hundreds of variables. Modeling and transcription errors may give rise to many of the above situations. The generation of solution is based on numerical techniques of linear algebra, which is often sensitive to the quality of the matrix of coefficients. Filtering out errant data is usually a significant exercise in the search of optimal solutions.

3.3 NUMERICAL SOLUTION—THE SIMPLEX METHOD

The standard numerical procedure is based on the algorithm due to Dantzig as mentioned earlier. It is referred to as the *Simplex method*. The procedure is related to the solution of a system of linear equations. The actual application of the procedure can be associated with the Gauss–Jordan method from linear algebra, where the coefficient rows are transformed through elementary multiplication and addition. Most mainframe computer installations usually carry software that will help solve LP problems. MATLAB also provides procedures to solve LP problems in its Optimization Toolbox. In this section, the Simplex method is applied to simple problems primarily to understand the programming and geometric features. This will be used when we discuss direct techniques for nonlinear problems. In the next section, the Simplex method is introduced in detail with explanations. In subsequent sections, MATLAB, or spreadsheet programs like Excel can also be used to implement the Simplex method.

3.3.1 Features of the Simplex Method

In this section, the machine selection example introduced earlier is set up for applying the Simplex method. It is instructive to note the method is iterative. Given a starting point, it will march forward through improving designs until it has found the solution or cannot proceed further. For completeness the original problem is rewritten here:

$$\text{Maximize} \quad f(\mathbf{X}): 990x_1 + 900x_2 + 5250 \tag{3.1}$$

$$\text{Subject to:} \quad g_1(\mathbf{X}): 0.4x_1 + 0.6x_2 \leq 8.5 \tag{3.2}$$

$$g_2(\mathbf{X}): 3x_1 - x_2 \leq 25 \tag{3.3}$$

$$g_3(\mathbf{X}): 3x_1 + 6x_2 \leq 70 \tag{3.4}$$

$$x_1 \geq 0; \quad x_2 \geq 0$$

The problem was also transformed to the standard format as follows.

Example 3.1

$$\text{Minimize} \quad f(\mathbf{X}): -990x_1 - 900x_2 - 5250 \tag{3.5}$$

$$\text{Subject to:} \quad g_1(\mathbf{X}): 0.4x_1 + 0.6x_2 + x_3 = 8.5 \tag{3.6}$$

$$g_2(\mathbf{X}): 3x_1 - x_2 + x_4 = 25 \tag{3.7}$$

$$g_3(\mathbf{X}): 3x_1 + 6x_2 + x_5 = 70 \tag{3.8}$$

$$x_1 \geq 0; \quad x_2 \geq 0; \quad x_3 \geq 0; \quad x_4 \geq 0; \quad x_5 \geq 0 \tag{3.9}$$

x_3, x_4, and x_5 are the slack variables.

The Simplex method is used on the problem being expressed in the standard format. The following information is useful in organizing the calculation as well as recognizing the motivation for subsequent iterations. These refer to a problem for which a unique solution exists. While many of the items below were introduced earlier, they are referenced here for completeness.

- The number of variables in the problem is n. This includes the slack and surplus variables.
- The number of constraints is $m (m < n)$.
- The problem is always to minimize the objective function f.
- The number of basic variables is m (same as the number of constraints).
- The number of nonbasic variables is $n - m$.
- The set of unit vectors expected is $e_1, e_2, \ldots, e_m$.
- The column of the right-hand side (**b**) is positive and greater than or equal to zero.
- The calculations are organized in a table, which is called a tableau, or Simplex tableau. It will be referred to as the table in this book.
- Only the values of the coefficients are necessary for the calculations. The table therefore contains only coefficient values, the matrix [**A**] in previous discussions. These are the coefficients in the constraint equations.
- The objective function is the *last row* in the table. The constraint coefficients are written first.
- Row operations consist of adding (subtracting) a definite multiple of the *pivot* row to other rows of the table. The pivot row identifies the row in which the unit vector (in a certain column) will have the value of 1.
- Row operations are needed to obtain the canonical form. Practically this implies that you should be able to spot the *unit vectors* in the table. If this happens, then the current iteration is complete and considerations for the next one must begin.

With this in mind, the Simplex method is applied to the machine selection problem. The example will use and clarify the items in the above list.

3.3.2 Application of Simplex Method

Simplex Table 3.1: In Table 3.1 (Simplex Table 1), the first row indicates the variable names. The last row is the objective function. Spreadsheets are an efficient way to process the information. If it is being employed, then the symbol f can be removed from the last column and last row to allow numerical calculations (as shown). The last column is the right-hand side values. The rest of the entries are the coefficients of the constraint equations.

The current iteration is over if the table displays the *canonical* form. In practice the canonical form comprises spotting the m unit vectors in the table, as well as making sure the entries under the b column, except for the rows representing the objective function(s), are nonnegative (≥ 0). A glance at Simplex Table 3.1 indicates that the canonical form is present.

The unit vectors in the table also identify those m variables that will belong to the basis. Those variables will have a nonzero value for this iteration. The remaining, $n-m$, nonbasic variables are set to zero. This solution is directly interpreted from Table 3.1. The solution therefore is

$$x_1 = 0.0, x_2 = 0.0, x_3 = 8.5, x_4 = 25, x_5 = 70, \text{ and } f = -5250$$

Has the Optimal Solution Been Obtained? The optimal solution requires that f be reduced as much as possible. For the current solution the value of x_1 is 0. From Equation (3.5), if x_1 were increased from zero, then the objective function f decreases further in value. Similar reasoning can be used for x_2. *Therefore, the solution can be further improved.* Solution has not converged. The second/next iteration will require assembling another table, Table 3.2.

The only way for x_1 and/or x_2 to have a positive value is if they became a basic variable. Since the number of basic variables is prescribed for the problem, some of the current basic variables must become nonbasic. For each iteration, the Simplex method allows *only one pair* of variables to conduct this *basis–nonbasis* exchange. Each iteration takes place through a new table. To affect the exchange it is necessary

Table 3.1 Simplex Table 3.1

x_1	x_2	x_3	x_4	x_5	b
0.4	0.6	1	0	0	8.5
3	−1	0	1	0	25
3	6	0	0	1	70
−990	−900	0	0	0	5280

to determine which is the variable that will enter the basis (x_1 or x_2), Entering Basic Variable (EBV). Also to be established is which of the variables (x_3, x_4, or x_5) will leave the basis, Leaving Basic Variable (LBV). In terms of the numerical calculations, in the new table, the unit vector under the LBV column will be transferred to the EBV column through elementary row/column operations.

Entering Basic Variable: The EBV is chosen by its ability to provide for the largest decrease in the objective function in the current iteration/table. This is determined by the largest negative coefficient in the row associated with the objective function. From Table 3.1, this value is –990 and corresponds to the variable x_1. EBV is x_1. A tie can be broken arbitrarily.

Leaving Basic Variable: The LBV is also chosen by its ability to improve the objective. The LBV is only determined *after the EBV has been chosen.* This is important because only the column under the EBV is examined to establish the LBV. The minimum ratio of the values of the right-hand side (b column values) to the corresponding coefficients in the EBV column, provided the coefficient value is positive, decides the LBV. This minimum value identifies the row which is also the *pivot row.* In this row, the column, which contains the unity value of the some unit vector (or a current basic variable), is the column of the leaving basic variable. This is the LBV.

In Table 3.1 the EBV has already been identified as x_1. The ratios under this column are 21.25, 8.3333, and 23.3333. The least value is 8.3333. This corresponds to the second row of Table 3.1. The unit vector under the x_4 column is in this row (row 2). Therefore, x_4 is the LBV and the second row is the pivot row for Table 3.2. This implies that the second row will be utilized to change the current x_1 column (EBV column) into a unit vector that is currently under x_4. This mechanical procedure is based on the following reasoning. The value of EBV must be such that it will not cause constraint violation with the current values of the other basic feasible variables. The application of the Gauss–Jordan process should shift the column under the LBV to the EBV leaving the column of the other unit vectors (that are part of the basis) unaltered. Table 3.2 is then constructed using the pivot row 2.

Table 3.2 Simplex Table 3.2

x_1	x_2	x_3	x_4	x_5	b
0	0.7333	1	–0.1333	0	5.1667
1	–0.3333	0	0.3333	0	8.3333
0	7	0	–1	1	45
0	–1230	0	330	0	13530

Simplex Table 3.2 (Table 3.2): The pivot row is obtained from original row 2 of Table 3.1. The following is the original row

3 –1 0 1 0 25

The value of the first element in the new row will be 1. The row is modified by dividing through by 3. The new second row in Table 3.2 will be

1 –0.3333 0 0.3333 0 8.3333

In Table 3.2, the first element of the first row (Table 3.1) must be reduced to 0. This is always achieved by *adding/subtracting from the row being modified an appropriate multiple of the pivot row.* The modification to the first row is

row 1 – {0.4 * (pivot row 2)} to yield
0 0.73331 –0.1333 0 5.1667

The third row will be obtained by

row 3 – {3 * (pivot row 3)}

The fourth row will be obtained by

row 4 + {(990 * (pivot row 2)}

Table 3.2 is obtained after these calculations.

The negative value of the coefficient in Table 3.2 in row 4 indicates that we need at least one more iteration/table. The EBV is x_2. By examining the positive ratios of the values in the b column to the coefficient in the x_2 column, the LBV is identified as x_5. The pivot row is row 3. Table 3.3 will be constructed by transforming the rows of Table 3.2.

Simplex Table 3.3 (Table 3.3)

pivot row 3 = (row 3)/7
new row 1 = row 1 – {0.73333 * (pivot row 3)}
new row 2 = row 2 + {0.3333 * (pivot row 3)}
new row 4 = row 4 + {1230 * (pivot row 3)}

Table 3.3 is obtained based on the above operations.

Table 3.3 Simplex Table 3.3

x_1	x_2	x_3	x_4	x_5	b
0	0	1	–0.0285	–0.1047	0.4524
1	0	0	0.2857	0.0476	10.4762
0	1	0	–0.1428	0.1428	6.4285
0	0	0	154.28	175.71	21437.14

Table 3.3 does not have any negative value in the objective function row. A solution has been achieved. The basic variables have the unit vector in their column. The nonbasic variables are set to zero. From Table 3.3 the optimum values can be observed as:

$$x_1^* = 10.4762, x_2^* = 6.4285, x_3^* = 0.4524, f^* = -21437.14$$

Note, the value of $f^* + 21437.14 = 0$ from which the solution for the objective function (f^*) can be obtained.

3.3.3 Solution Using MATLAB

The tables in the previous computation can be generated quite effortlessly using a spreadsheet. This would be accomplished by setting up the mathematical manipulation in one cell and copying and pasting to the other cells. Since this book also assists in MATLAB programming, the illustration here will allow an opportunity to practice the use of MATLAB for matrix manipulations. The code presented here is **captured** using MATLAB's **diary** command. The command requires the name of a text file which records all communication with MATLAB as well as its response during an interactive session (see *help diary*). The text file is commented liberally (for your understanding) and using the semicolon at the end of the strings prevents their echo. In this file the comments are quoted string. The code represents the details of an actual session. These statements are typed after the MATLAB prompt in the Command window. This is therefore an interactive session. Values to the right of ans = is the echo of result of the execution of the MATLAB command. Note: comments need not be typed.

Example 3_1.txt

```
format compact
format rational
'Set up the coefficient matrix and b as matrix A';

A=[0.4 0.6 1 0 0 8.5;3 -1 0 1 0 25;3 6 0 0 1 70;-990
-900 0 0 0 5280]
A =
    2/5   3/5    1   0   0   17/2
    3     -1     0   1   0   25
    3      6     0   0   1   70
 -990   -900     0   0   0   5280
'This is Table 1 - note the canonical form';
'EBV is x1 - first column';

'To find LBV divide last column by coeff. in EBV';
'column';
'Take the minimum of the positive values';
```



```
'The row identifies the pivot row to create the next
table';

  A(:,6)/A(:,1)
  ans =
       0     0     0    -17/1980
       0     0     0     -5/198
       0     0     0     -7/99
       0     0     0    -16/3
'something not right - The above division should have';
'been an element by element one';

  A(:,6)./A(:,1)
  ans =
     85/4
     25/3
     70/3
     -16/3
  format short

  A(:,6)./A(:,1)
  ans =
     21.2500
      8.3333
     23.3333
     -5.3333
  'Second row is the pivot row and x4 is LBV';
  'Constructing Table 2';
  'note the scaling factor for the matrix';
  A
  A =
   1.0e+003 *
    0.0004   0.0006  0.0010         0         0  0.0085
    0.0030  -0.0010       0    0.0010         0  0.0250
    0.0030   0.0060       0         0    0.0010  0.0700
   -0.9900  -0.9000       0         0         0  5.2800
  'The element at A(2,1) must be 1';
  A(2,:) = A(2,:)/A(2,1)
  A =
   1.0e+003 *
    0.0004   0.0006  0.0010         0         0  0.0085
    0.0010  -0.0003       0    0.0003         0  0.0083
    0.0030   0.0060       0         0    0.0010  0.0700
   -0.9900  -0.9000       0         0         0  5.2800
```

```
'The element at A(1,1) must be a 0 ';
A(1,:) = A(1,:) - 0.4*A(2,:)
A =
 1.0e+003 *
       0   0.0007   0.0010   -0.0001        0   0.0052
  0.0010   -0.0003        0    0.0003        0   0.0083
  0.0030    0.0060        0         0   0.0010   0.0700
 -0.9900   -0.9000        0         0        0   5.2800
'Element at A(3,1) must be a zero';
A(3,:) = A(3,:) - A(3,1)*A(2,:)
A =
 1.0e+003 *
       0   0.0007   0.0010   -0.0001        0   0.0052
  0.0010   -0.0003        0    0.0003        0   0.0083
       0   0.0070        0   -0.0010   0.0010   0.0450
 -0.9900   -0.9000        0         0        0   5.2800
'Element at A(4,1) must be 0';
A(4,:) = A(4,:) - A(4,1)*A(2,:)
A =
 1.0e+004 *
       0   0.0001   0.0001   -0.0000        0   0.0005
  0.0001   -0.0000        0    0.0000        0   0.0008
       0   0.0007        0   -0.0001   0.0001   0.0045
       0   -0.1230       0    0.0330        0   1.3530
'Table 2 complete - and canonical form is present';
'Solution is not converged because of A(4,2)';
'EBV is x2';
'Calculation of LBV';
A(:,6)./A(:,2)
ans =
    7.0455
  -25.0000
    6.4286
  -11.0000
'Pivot row is third row and LBV is x5'
ans =
Pivot row is third row and LBV is x5
'Oops forgot the semicolon';
'Calculation of LBV';
'Construction of Table 3.3 - no echo of calculations';
'A(3,2) must have value 1';
A(3,:) = A(3,:)/A(3,2);
'A(1,2) must have a value of 0';
A(1,:) = A(1,:) - A(1,2)*A(3,:);
```



```
'A(2,2) must have a value of 0';
A(2,:) = A(2,:) - A(2,2)*A(3,:);
'A(4,2) must have a value of 0';
A(4,:) = A(4,:) - A(4,2)*A(3,:);
A
A =
 1.0e+004 *
       0        0    0.0001   -0.0000   -0.0000   0.0000
  0.0001        0         0    0.0000    0.0000   0.0010
       0   0.0001         0   -0.0000    0.0000   0.0006
       0        0         0    0.0154    0.0176   2.1437
format rational
A
A =
  0  0  1  -1/35  -11/105  19/42
  1  0  0   2/7     1/21  220/21
  0  1  0  -1/7     1/7    45/7
  0  0  0  1080/7  1230/7  150060/7
'No further iterations necessary';
diary off
```

The last few commands suggest the possibility of developing a for loop for the row manipulations. It is suggested as a problem at the end of the chapter.

3.3.4 Solution Using MATLAB's Optimization Toolbox

This section is useful for those that have access to the Optimization Toolbox from MATLAB (Ver 5.2).

Start MATLAB. In the Command window type help lp. This will provide information on the use of the linear programming routine to solve the problem. Here, the standard use is employed. It involves specifying values for the cost coefficients (f-vector), the coefficient/constraint matrix (A-matrix), and the right-hand side vector (b-vector). The following sequence of steps illustrates the use of the program.

Using the Linear Programming Routine

```
>> f = [-990;-900];
>> A = [0.4 0.6 ; 3 -1; 3 6]
>> b = [8.5 25 70]'
>> x = lp(f,A,b)
```

The solution as posted in the Command window:

```
x =
   10.4762
    6.4286
>> sol = f'*x
```

```
sol =
 -1.6157e+004
```

To this solution must be added the constant −5280 which was omitted in problem definition for MATLAB. There are many different ways to use the linear programming function in MATLAB.

3.4 ADDITIONAL EXAMPLES

In this section additional examples are presented. These examples illustrate the extension/modification of the Simplex method to handle greater than or equal constraints, negative values for the design variables, equality constraints, and so on. In all the cases, the problem is transformed appropriately and the same Simplex method is then applied.

3.4.1 Example 3.2—Transportation Problem

The Fresh Milk cooperative supplies milk in gallon jugs from its two warehouses located in Buffalo (New York) and Williamsport (Pennsylvania). It has a capacity of 2000 gallons per day at Buffalo and 1600 gallons per day at Williamsport. It delivers 800 gallons/day to Rochester (New York). Syracuse (New York) requires 1440 gallons/day, and the remainder (1360 gallons) are trucked to New York City. The cost to ship the milk to each of the destinations is different and is given in Table 3.4. Establish the shipping strategy for minimum cost.

Problem Formulation: Let x be the number of gallons shipped from Buffalo to Rochester. Let y be the number of gallons shipped from Buffalo to Syracuse.

The warehousing constraint at Buffalo is 2000 gallons/day. Therefore,

$$x + y \le 2000 \tag{3.21}$$

Amount shipped from Williamsport to Rochester = $800 - x$
Amount shipped from Williamsport to Syracuse = $1440 - y$
Amount shipped from Williamsport to New York City is

$$1600 - (800 - x) - (1440 - y) \ge 0$$

Table 3.4 Shipping Cost (Cents per Gallon)

	Rochester	Syracuse	New York City
Buffalo	4.2	4.5	6.0
Williamsport	4.7	4.6	5.1

$$x + y \ge 640 \tag{3.22}$$

The side constraints on x and y are the respective warehouse limits. The shipping cost is

$$\text{Cost} = 4.2 * x + 4.5 * y + 6.0 * (2000 - x - y) +$$
$$4.7 * (800 - x) + 4.6 * (1440 - y) + 5.1 * (x + y - 640)$$
$$\text{Cost} = -1.4 * x - y + 19120 \tag{3.23}$$

Assembling the problem (Example 3.2)

$$\text{Minimize} \quad f(x, y): -1.4x - y + 19120 \tag{3.24}$$

$$\text{Subject to:} \quad g_1(x, y): x + y \le 2000 \tag{3.25}$$

$$g_2(x, y): x + y \ge 640 \tag{3.26}$$

$$0 \le x \le 800;\ 0 \le y \le 1440 \tag{3.27}$$

Figure 3.7 illustrates the graphical solution to Example 3.2. The line representing the constant objective function values term does not include the constant term (19120). The solution is at the intersection of constraint g_1 and the upper limit on the value of x as the objective is to decrease f as much as possible. From the figure the solution is

$$x = 800;\ y = 1200, f = 16800 \text{ or } \$168.00 \tag{3.28}$$

Two-Phase Simplex Method: The main difference between Examples 3.1 and 3.2 is the ≥ constraint in the latter. Since the standard LP problem only requires the design variables be semipositive (≥ 0), the right-hand constraints on x and y have to be accommodated through additional inequality constraints. Applying the regular Simplex method without additional processing would create a problem in recognizing the canonical form. Example 3.2 expressed in the standard format of linear programming is

$$\text{Minimize} \quad f(x,y): -1.4x - y + 19120 \tag{3.24}$$

$$\text{Subject to:} \quad g_1(x, y): x + y + s_1 = 2000 \tag{3.29}$$

$$g_2(x, y): x + y - s_2 = 640 \tag{3.30}$$

$$g_3(x, y): x + s_3 = 800 \tag{3.31}$$

$$g_4(x, y): y + s_4 = 1440 \tag{3.32}$$

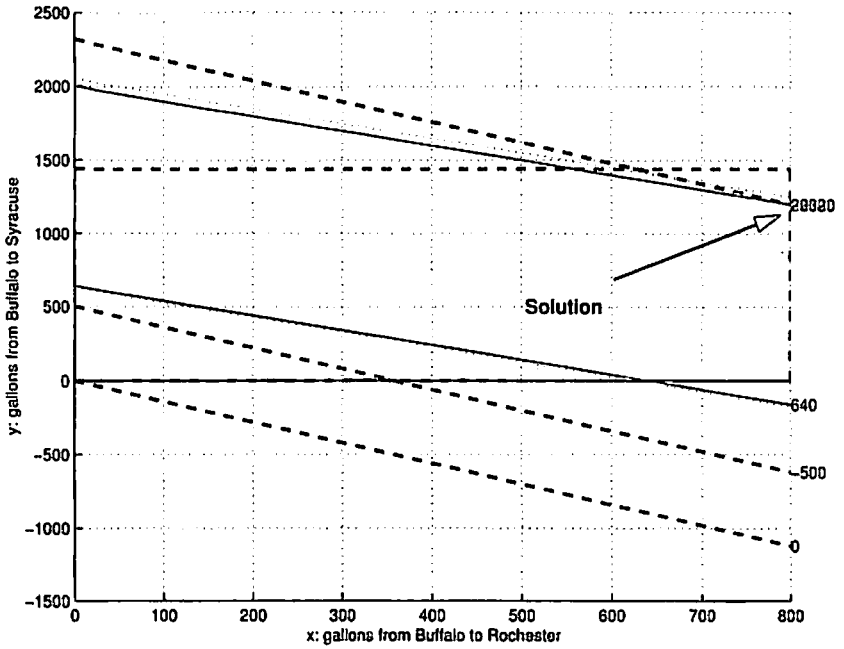

Figure 3.7 Graphical solution, Example 3.2.

$$x \geq 0; y \geq 0; s_1, s_2, s_3, s_4 \geq 0 \tag{3.33}$$

Here s_1, s_3, s_4 are the *slack* variables and s_2 is the *surplus* (similar to a slack—used for ≥ constraints) variable. In Example 3.1, when the first Simplex table was set up (Simplex Table 3.1), the slack variables were helpful in identifying the canonical form. Here in Example 3.2, it does not work out that way because the coefficient of s_2 is −1. Multiplying Equation (3.30) by −1 migrates the negative sign to the right-hand side, which is also disallowed as far as the recognition of the canonical form is concerned. This means additional problem-specific preprocessing must take place to identify the initial canonical form. Since the Simplex method is used to handle large problems with several variables, it is more convenient to apply the Simplex procedure in a consistent way. This is accomplished by applying the same Simplex technique in a two-part sequence.

The first part is recognized as *Phase I*. Here a new kind of variable, an *artificial variable*, is defined for each *surplus* variable in the same equation. Also, a new objective function called an *artificial cost function* or an *artificial objective function* is introduced. The artificial objective is defined as a sum of all the artificial variables in the problem. In Phase I, the artificial objective is reduced to zero using the standard Simplex procedure. When this is accomplished, Phase I is complete. If the artificial

objective depends only on the artificial variables, and if its value is zero, this implies that the artificial variables are *nonbasic* variables. This also suggests that these variables were basic variables at the start of the procedure. When Phase I is completed, then both the artificial objective function and the artificial variables are discarded from the table and *Phase II* begins.

Phase II is the standard Simplex technique applied to the table from the end of Phase I neglecting all the artificial elements. The table should be in canonical form. Additional tables are obtained as necessary until the solution is reached.

In Example 3.2 set up earlier, there will be one artificial variable a_1 and an artificial cost function A_f. In the example only Equation (3.30) will be affected as

$$g_2(x, y): x + y - s_2 + a_1 = 640 \tag{3.34}$$

The artificial cost function will be

$$A_f = a_1 \tag{3.35}$$

Table 3.5 represents the first table in Phase I. The first four rows represent the constraints. The fifth row represents the original objective function. The last row is the artificial objective function. This is the row used to drive the iterations in Phase I. Table 3.5 is not in canonical form, but replacing the sixth row by the result of subtracting the second row from the sixth row will provide a canonical form. Table 3.6 illustrates the result of such a row manipulation.

Simplex Method, Phase I: In Table 3.6 there are two choices (x, y) available for the EBV (entering basic variable) as both of them have a coefficient of −1 in the last row. While the choice can be arbitrary, x is a good choice because it has a larger negative coefficient in the original objective function. The LBV (leaving basic variable) is identified through the minimum positive value of the ratio of the values in the b column to the values under the x column. These ratios are 2000/1, 640/1, and 800/1. The selection identifies the second row as the *pivot* row and a_1 as the LBV. Using the second row as the pivot row, the unit vector $[0\ 1\ 0\ 0\ 0\ 0]^T$ has to be

Table 3.5 Example 3.2: Initial Table, Phase I

x	y	s_1	s_2	a_1	s_3	s_4	b
1	1	1	0	0	0	0	2000
1	1	0	−1	1	0	0	640
1	0	0	0	0	1	0	800
0	1	0	0	0	0	1	1440
−1.4	−1	0	0	0	0	0	f − 19120
0	0	0	0	1	0	0	A_f

Table 3.6 Example 3.2: Simplex Table 1, Phase I (Canonical Form)

x	y	s_1	s_2	a_1	s_3	s_4	b
1	1	1	0	0	0	0	2000
1	1	0	−1	1	0	0	640
1	0	0	0	0	1	0	800
0	1	0	0	0	0	1	1440
−1.4	−1	0	0	0	0	0	f − 19120
−1	−1	0	1	0	0	0	A_f − 640

transferred from the a_1 column to the x column. Construction of Table 3.7 starts with the second row in Table 3.6, which is also the second row of Table 3.7 because of the coefficient of 1 in the second row under the x column (no scaling is necessary). This is the pivot row. To obtain a 0 in the first row in the x column, the pivot row is subtracted from the first row in Table 3.6. The new first row is

$$0 \quad 0 \quad 1 \quad 1 \quad -1 \quad 0 \quad 0 \quad 1360$$

To obtain a 0 in the x column in the third row, the pivot row is subtracted from the third row of Table 3.6. The fourth row has a 0 in place and hence is copied from Table 3.6. The fifth row is obtained by adding a 1.4 multiple of the pivot row to the fifth row in Table 3.6. The last row is the addition of the pivot row to the last row from Table 3.6. Table 3.7 is compiled as follows:

Table 3.7 Example 3.2: Simplex Table 2, Phase I (Canonical Form)

x	y	s_1	s_2	a_1	s_3	s_4	b
0	0	1	1	−1	0	0	1360
1	1	0	−1	1	0	0	640
0	−1	0	1	−1	1	0	160
0	1	0	0	0	0	1	1440
0	0.4	0	−1.4	1.4	0	0	f − 18224
0	0	0	1	0	0	0	A_f

From Table 3.7, the value of A_f is 0, and a_1 is not a basic variable. This identifies the end of Phase I. The a_1 column and the last row are discarded and Phase II is started with Table 3.8.

Table 3.8 Example 3.2: Simplex Table 1, Phase II (Canonical Form)

x	y	s_1	s_2	s_3	s_4	b
0	0	1	1	0	0	1360
1	1	0	−1	0	0	640
0	−1	0	1	1	0	160
0	1	0	0	0	1	1440
0	0.4	0	−1.4	0	0	f − 18224

Simplex Method, Phase II: The column with the greatest negative coefficient (−1.4) is s_2. This is the EBV. The LBV is s_3. The third row is therefore the pivot row. The unit vector $[0\ 0\ 1\ 0\ 0]^T$ must be transferred from the s_3 column to the s_2 column through row manipulations using the pivot row. This exercise results in Table 3.9.

The negative coefficient under the y column suggests we are not finished yet. For the next iteration, EBV is y. LBV is s_1. The first row is the pivot row. The unit vector $[1\ 0\ 0\ 0\ 0]^T$ must be constructed under the y column using row manipulation using the pivot row. This results in Table 3.10.

There are no negative coefficients in the last row. The solution has been reached. From inspection of Table 3.10, the solution is

$$x = 800,\ y = 1200,\ s_2 = 1360,\ s_4 = 240,\ f = 16800 \tag{3.36}$$

In this example, the same Simplex method was repeatedly applied although the problem was solved using *two phases*. The first phase was a preprocessing phase to move the surplus variables away from the initial set of basic variables. This was achieved by introducing additional variables for each surplus variable and an additional cost function that drove the iterations of the first phase.

To summarize, for ≥ and = constraints, artificial variables and an artificial cost function are introduced into the problem for convenience. The Simplex method is applied in two phases. The first phase is terminated when the artificial variables can be eliminated from the problem.

Table 3.9 Example 3.2: Simplex Table 2, Phase II (Canonical Form)

x	y	s_1	s_2	s_3	s_4	b
0	1	1	0	−1	0	1200
1	0	0	0	1	0	800
0	−1	0	1	1	0	160
0	1	0	0	0	1	1440
0	−1	0	0	1.4	0	f − 18000

Table 3.10 Example 3.2: Simplex Table 3 (Final), Phase II

x	y	s_1	s_2	s_3	s_4	b
0	1	1	0	−1	0	1200
1	0	0	0	1	0	800
0	0	1	1	0	0	1360
0	0	−1	0	1	1	240
0	0	1	0	0.4	0	f − 16800

3.4.2 Example 3.3—Equality Constraints and Unrestricted Variables

Example 3.3 will include an equality constraint and illustrates the technique for handling variables that are unrestricted in sign (can have negative values). Variables like profit, temperature, and net income can be negative quite often. The equality constraint is handled by introducing an artificial variable and applying the two-phase Simplex technique illustrated in Example 3.2. When this corresponding artificial variable is zero (end of Phase I), the constraint has been met. The negative value for a variable is simulated through the difference between two positive variables. Once again, these changes preserve the Simplex method in its original form. The following is an imaginary problem to illustrate additional features so some liberty is taken in the formulation.

The Problem: Today, a full-time student on campus is always driven to maximize grades. A strong influence for a favorable outcome is the amount of investment made in hitting the books versus the time spent playing pinball. In a typical day, at least one hour is definitely extended to the pursuit of learning or pressing the sides of the coin-operated machine. Not more than five hours is available for such dispersion. Over the years, a fatigue threshold of four units, based on a combination of the two activities, has been established as a norm for acceptable performance. This combination is the sum of the hours spent at the pinball machine and twice the hours spent on homework, an acknowledgment that hitting the books is stressful. A negative hour of pinball playing will go to increase the time spent studying, which will contribute positively to the grades. The overall grade is determined as a linear combination of twice the time spent academically and subtracting the time spent playing. The problem therefore is to distribute the time to maximize the grades obtained.

Problem Formulation: There are two original design variables in the problem. x_1 is the number of hours spent studying, and x_2 is the time spent enjoying the game of pinball when the same time can be advantageously spent maintaining high grades. The objective function can be expressed as

$$\text{Maximize} \quad f(x_1, x_2): 2x_1 - x_2$$

The constraints can be recognized from the statement of the problem directly as

$$g_1: \quad x_1 + x_2 \leq 5$$

$$g_2: \quad 2x_1 + x_2 = 4$$

$$g_3: \quad x_1 + x_2 \geq 1$$

$$x_1 \geq 0, \qquad x_2 \text{ is unrestricted in sign}$$

Figure 3.8, using the **drawLine.m** m-file, is a graphical description of the problem and the solution. The solution is at $x_1 = 3$ and $x_2 = -2$ in the figure, where conveniently the objective function is also passing through.

Standard Format: As the standard format expects only nonnegative variables, the variable x_2 is replaced by a pair of variables in the formulation:

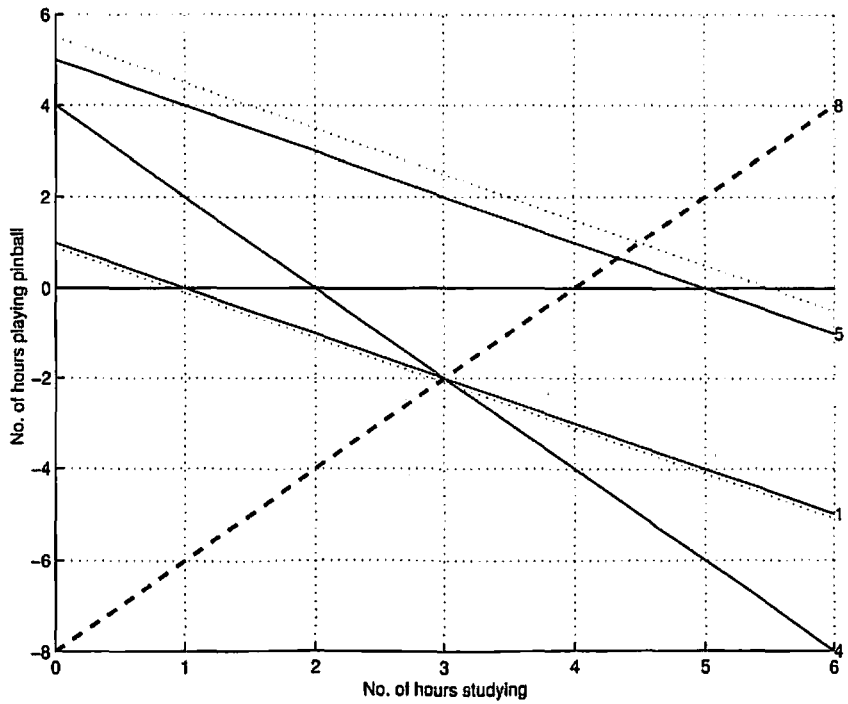

Figure 3.8 Graphical solution, Example 3.3.

$$x_2 = x_{21} - x_{22} \tag{3.37}$$

The standard format after converting to a minimization problem, introducing slack, surplus, and artificial variables, and including Equation (3.37) results in

$$\text{Minimize} \quad f: -2x_1 + x_{21} - x_{22} \tag{3.38}$$

$$\text{Subject to:} \quad g_1: x_1 + x_{21} - x_{22} + s_1 = 5$$

$$g_2: 2x_1 + x_{21} - x_{22} + a_1 = 4$$

$$g_3: x_1 + x_{21} - x_{22} - s_2 + a_2 = 1$$

The artificial function to be driven to zero is

$$A_f = a_1 + a_2 \tag{3.39}$$

Simplex Table 0: Table 3.11 captures the standard format in a table. It is called Table 0 to allow for preprocessing with respect to the artificial variables so that a canonical form can be observed. Adding the second and third row and subtracting the result from the last row and replacing the last row with this computation will yield the next table.

Table 3.11 Example 3.3: Table 0, Phase I

x_1	x_{21}	x_{22}	s_1	a_1	s_2	a_2	b
1	1	−1	1	0	0	0	5
2	1	−1	0	1	0	0	4
1	1	−1	0	0	−1	1	1
−2	1	−1	0	0	0	0	f
0	0	0	0	1	0	1	A_f

Phase I, Table 1: Table 3.12 provides the start of the Simplex method. In Phase I the motivation is to drive the artificial function to 0, which has a value of 5. The EBV is x_1. The LBV is a_2. The third row is the pivot row. Row manipulations with the pivot row lead to Table 3.13.

Table 3.12 Example 3.3: Table 1, Phase I

x_1	x_{21}	x_{22}	s_1	a_1	s_2	a_2	b
1	1	−1	1	0	0	0	5
2	1	−1	0	1	0	0	4
1	1	−1	0	0	−1	1	1
−2	1	−1	0	0	0	0	f
−3	−2	2	0	0	1	0	$A_f - 5$

Phase I, Table 2: Table 3.13 is the second table under Phase I. The value of the artificial function is 2, so Phase I is not yet over. The EBV is x_{22}. The LBV is a_1. The second row is the pivot row. This row will be used for row manipulations to yield the next table.

Table 3.13 Example 3.3: Table 2, Phase I

x_1	x_{21}	x_{22}	s_1	a_1	s_2	a_2	b
0	0	0	1	0	1	−1	4
0	−1	1	0	1	2	−2	2
1	1	−1	0	0	−1	1	1
0	3	−3	0	0	−2	2	$f+2$
0	1	−1	0	0	−2	3	$A_f - 2$

Phase I, Table 3: Table 3.14 is the new table after the row operations. The value of the artificial function is 0 as both a_1 and a_2 are nonbasic variables with a value of 0 for this iteration. This signifies the end of Phase I. Phase II will start with the last row (artificial function) and the two columns that represent the artificial variables removed from the current table.

Table 3.14 Example 3.3: Table 3, Phase I (Final)

x_1	x_{21}	x_{22}	s_1	a_1	s_2	a_2	b
0	0	0	1	0	1	−1	4
0	−1	1	0	1	2	−2	2
1	0	0	0	1	1	−1	3
0	0	0	0	3	4	−4	$f+8$
0	0	0	0	1	0	1	A_f

Phase II, Table 1: Table 3.15 represents the table to start the iterations for Phase II. However, there are no negative coefficients in the last row (row corresponding to the objective function). Scanning the table further, the canonical form can be observed. Therefore, the solution has been obtained. The solution is

Table 3.15 Example 3.3: Table 1, Phase II

x_1	x_{21}	x_{22}	s_1	s_2	b
0	0	0	1	1	4
0	−1	1	0	2	2
1	0	0	0	1	3
0	0	0	0	4	$f+8$

$$x_1 = 3; x_{21} = 0; x_{22} = 2; s_1 = 4; s_2 = 0; f = -8$$

The value of x_2 is

$$x_2 = (x_{21} - x_{22}) = (0 - 2) = -2$$

which was identified earlier graphically.

The above example illustrated the Simplex method for handling negative variables and equality constraints. Note the application of the Simplex technique itself did not change. An interesting feature of the problem was that Phase II was immediately obtained after Phase I without any further iteration, which suggests that the search for the optimum is taking place in Phase I even though the focus is on the artificial variables.

3.4.3 Example 3.4—A Four-Variable Problem

Example 3.4 presents a problem with four variables. The procedure is identical to the one in Example 3.1, except that there is no graphical solution to the problem.

The Problem: The RIT student-run microelectronic fabrication facility is taking orders for four indigenously developed ASIC chips that can be used in (1) touch sensors, (2) LCD, (3) pressure sensors, and (4) controllers. There are several constraints on the production based on space, equipment availability, student hours, and the fact that the primary mission of the facility is student training. First, the handling time constraint, outside of processing, for all chips is 600 hours. Touch sensors require 4 hours, LCD 9 hours, pressure sensors 7 hours, and controllers 10 hours. Second, the time available on the lithographic machines is about 420 hours. Touch sensors and LCD require 1 hour, pressure sensors 3 hours, and controllers 8 hours. Packaging considerations place the maximum at 800 volume units. Touch sensors require 30 volume units, LCD 40 volume units, pressure sensors 20 units, and controllers 10 units because of their compact size. All the constraints above are indicated per week of operation of the facility. The net revenue is \$6, \$10, \$9, and \$20 for the touch sensor, LCD, pressure sensor, and controller, respectively. The facility is interested in maximizing revenue per week and would like to determine the right mix of the four devices.

Problem Formulation: The formulation is straightforward based on the statements above. Let x_1 represent the number of touch sensor chips per week, x_2 the number of LCD, x_3 the number of pressure sensors, and x_4 the number of controllers. The objective function is

$$\text{Maximize:} \quad f\text{: } 6x_1 + 10x_2 + 9x_3 + 20x_4$$

The handling time constraint can be expressed as

$$g_1\text{: } 4x_1 + 9x_2 + 7x_3 + 10x_4 \le 600$$

The availability of the lithographic machine can be developed as

$$g_2\text{: } x_1 + x_2 + 3x_3 + 8x_4 \le 420$$

The packaging constraint is

$$g_3\text{: } 30x_1 + 40x_2 + 20x_3 + 10x_4 \le 800$$

All design variables are expected to be greater than zero. As formulated above, the problem suggests a degree of incompleteness. There are *four* design variables and only *three* constraints. The number of variables in the basis can only be three. Hence, at least one of the variables must have a value of zero. Several useful additional constraints can still be included to define a valid optimization problem with a nonzero solution. This is now a modeling issue. For problems with a limited number of design variables, paying attention to the problem development allows anticipation of the solution as well as the opportunity to troubleshoot decisions from practical considerations. It is left to the student to explore this problem further.

Standard Format: The objective function requires a minimum formulation. Hence,

$$\text{Minimize} \quad f\text{: } -6x_1 - 10x_2 - 9x_3 - 20x_4$$

The symbol f has been retained for convenience even though the direction of optimization has changed. The constraints are set up using slack variables s_1, s_2, and s_3.

$$g_1\text{: } 4x_1 + 9x_2 + 7x_3 + 10x_4 + s_1 = 600$$

$$g_2\text{: } x_1 + x_2 + 3x_3 + 8x_4 + s_2 = 420$$

$$g_3\text{: } 30x_1 + 40x_2 + 20x_3 + 10x_4 + s_3 = 800$$

All variables are ≥ 0. The formulation above suggests the standard Simplex method.

Simplex Table 1: Table 3.16 is the initial table for Example 3.4. The canonical form is observable from the table and the initial basic feasible solution can be determined.

Table 3.16 Example 3.4: Simplex Table 1

x_1	x_2	x_3	x_4	s_1	s_2	s_3	b
4	9	7	10	1	0	0	600
1	1	3	8	0	1	0	420
30	40	20	10	0	0	1	800
−6	−10	−9	−20	0	0	0	f

Table 3.17 Example 3.4: Simplex Table 2

x_1	x_2	x_3	x_4	s_1	s_2	s_3	b
2.75	7.75	3.25	0	1	−1.25	0	75
0.125	0.125	0.375	1	0	0.125	0	52.5
28.75	38.75	16.25	0	0	−1.25	1	275
−3.5	−7.5	−1.5	0	0	2.5	0	f+1050

s_1, s_2, and s_3 are the basic variables. The EBV is x_4 and the LBV is s_2 which is the minimum of {60, 52.5, 80}. The pivot row is the second row used for the row manipulations to lead to Table 3.17.

Simplex Table 2: Using the pivot row identified in the last table, the unit vector $[0\ 0\ 1\ 0]^T$ under the s_2 column needs to be transferred to the x_4 column through elementary row operations. Table 3.17 shows the canonical form after completion of all the operations. The basic variables are x_4, s_1, and s_3. The EBV is x_2 and the LBV is s_3. The pivot row is the third row. The objective of the row manipulations is to transfer the unit vector $[0\ 0\ 1\ 0]^T$ from the s_3 column to the x_2 column.

Simplex Table 3: Table 3.18 denotes the reduced table with the canonical form after the required row operations are completed. The basis variables are x_2, x_4, and s_1. There are no negative coefficients in the last row suggesting the solution has been obtained. From the table the solution is

$$x_1 = 0, x_2 = 7.096, x_3 = 0, x_4 = 51.61, f = 1103.22$$

The solution above is not satisfactory because the actual decision will involve integer values for the design variables. At this time *integer programming* is not an option. The adjusted solution is

$$x_1 = 0, x_2 = 7, x_3 = 0, x_4 = 52, f = 1110$$

While this choice satisfies the constraints g_3, g_2 however needs a little elasticity (limit is 423) to be satisfied. Note that s_1 is a basic variable so constraint g_1 should not be a problem.

Table 3.18 Example 3.4: Simplex Table 3

x_1	x_2	x_3	x_4	s_1	s_2	s_3	b
−3	0	0	0	1	−1	−0.2	20
0.032258	0	0.322581	1	0	0.1290322	−0.00323	51.6129
0.741935	1	0.419355	0	0	−0.032258	0.025806	7.096774
2.064516	0	1.645161	0	0	2.25806451	0.193548	f+1103.22

Using MATLAB

This is a standard LP problem. This section uses MATLAB in an interactive session to solve the problem. The following is a *diary file* for the MATLAB session for this example. Note the warning from MATLAB, as well as the subsequent use of the LP program using bounds on the design variables: The bold italicized statements are commands used to obtain the solution. The MATLAB prompt is edited out.

```
help lp
LP Linear programming.
X=LP(f,A,b) solves the linear programming problem:
  min f 'x subject to: Ax <=  b
X=LP(f,A,b,VLB,VUB) defines a set of lower and upper
bounds on the design variables, X, so that the solution
is always in the range VLB <= X <= VUB.
X=LP(f,A,b,VLB,VUB,X0) sets the initial starting point
to X0.

X=LP(f,A,b,VLB,VUB,X0,N) indicates that the first N
constraints defined by A and b are equality constraints.

X=LP(f,A,b,VLB,VUB,X0,N,DISPLAY) controls the level of
warning messages displayed. Warning messages can be
turned off with DISPLAY = -1.
[x,LAMBDA]= LP(f,A,b) returns the set of Lagrangian
multipliers,
LAMBDA, at the solution.

[X,LAMBDA,HOW] = LP(f,A,b) also returns a string how
that indicates error conditions at the final iteration.
LP produces warning messages when the solution is either
unbounded or infeasible.
c = [-6 -10 -9 -20];
A = [4 9 7 10; 1 1 3 8; 30 40 20 10];
b = [600 420 800];
x = [c, A, b]
x = lp(c, A, b)
Warning: The solution is unbounded and at infinity;
the constraints are not restrictive enough.
x =
 1.0e+015 *
 -0.0000
 3.7067
 -8.8391
 2.8513
```

```
vlb = [0 0 0];
vub = [100, 100, 100];
x = lp(c, A, b, vlb, vub)
x =
 0.0000
 7.0968
   0
 51.6129
```

This is the solution obtained by working through the Simplex method.

3.5 ADDITIONAL TOPICS IN LINEAR PROGRAMMING

This section looks briefly at additional ideas associated with the LP problem. First, there is a discussion of the *dual* problem associated with an LP problem. Duality is important in the discussion of sensitivity analysis—*variation in optimization solution due to variation in the original parameters of the problem.* Following this is a discussion of several pieces of information that can be extracted from the final table of the Simplex method.

3.5.1 Primal and Dual Problem

Associated with every LP problem, which will be referred to as a *primal* problem, there is a corresponding *dual* problem. In many ways, the dual problem is a transposition of the primal problem. This is meant to imply that if the primal problem is a *minimization* problem the dual problem will be a *maximization* one. If the primal problem has n variables, the dual problem will have n constraints. If the primal problem has m constraints, the dual problem will have m variables. For the discussion of duality, the *standard primal problem* is generally defined as a *maximization* problem with $\leq$ inequalities (though the standard problem in the LP programming discussion was a minimization problem [see Equations (3.10)–(3.12)] and the constraints were equalities). The standard primal problem in this section is defined as [3]

$$\text{Maximize} \quad z: c_1x_1 + c_1x_2 + \ldots + c_nx_n \tag{3.40}$$

$$\text{Subject to:} \quad a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq b_1 \tag{3.41}$$

$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \leq b_2$$

$$\ldots \quad \ldots \quad \ldots \quad \ldots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$$

$$x_j \geq 0, j = 1, 2, \ldots, n \tag{3.42}$$

This is also referred to as a *normal maximum problem.*

The dual of the above problem is defined as

$$\text{Minimize} \quad w: b_1y_1 + b_1y_2 + \ldots + b_my_m \tag{3.43}$$

$$\text{Subject to:} \quad a_{11}y_1 + a_{21}y_2 + \ldots + a_{m1}y_m \geq c_1 \tag{3.44}$$

$$a_{12}y_1 + a_{22}y_2 + \ldots + a_{m2}y_m \geq c_2$$

$$\ldots \quad \ldots \quad \ldots \quad \ldots$$

$$a_{1n}y_1 + a_{2n}y_2 + \ldots + a_{mn}y_m \geq c_n$$

$$y_i \geq 0, \quad i = 1, 2, \ldots, m \tag{3.45}$$

Relations (3.43)–(3.45) describe a *normal minimum problem.* There is also an inverse relationship between the definition of the two problems above as far as the identification of the primal and dual problem. If the latter is considered a primal problem, then the former is the corresponding dual problem.

The following observations can be made with respect to the pair of problems established above:

1. The number of dual variables is the same as the number of primal constraints.
2. The number of dual constraints is the same as the number of primal variables.
3. The coefficient matrix **A** of the primal problem is transposed to provide the coefficient matrix of the dual problem.
4. The inequalities are reversed in direction.
5. The maximization problem of the primal problem becomes a minimization problem in the dual problem.
6. The cost coefficients of the primal problem become the right-hand side values of the dual problem. The right-hand side values of the primal become the cost coefficients of the dual problem.
7. The primal and dual variables both satisfy the nonnegativity condition.

Example 3.5 The following example, adapted and modified from Reference 3, is used to illustrate the primal/dual versions of the same problem. The problem is solved using techniques developed in this chapter.

The Problem: The local college's School of American Craftsman has decided to participate seriously in a local charity event by manufacturing special commemorative desks, tables, and chairs. Each type of furniture requires lumber as well as two types of skilled labor: finishing and carpentry. Table 3.19 documents the resources needed. The local lumber cooperative has donated 48 board feet of lumber. Faculty and staff have volunteered 20 finishing hours and 8 carpentry hours. The desk will be sold for

Table 3.19 Example 3.5: Resources

Resources	Desk	Table	Chair
Lumber (board feet)	8	6	1
Finishing hours (hours)	4	2	1.5
Carpentry (hours)	2	1.5	0.5

$60, the table for $30, and the chair for $20. The school does not expect to sell more than five tables. The school would like to maximize revenue collected.

Standard Format—Primal Problem: Defining x_1 as the number of desks produced, x_2 as the number of tables manufactured, and x_3 as the number of chairs manufactured, the primal problem can be expressed in standard format as

$$\text{Maximize } z: \quad 60x_1 + 30x_2 + 20x_3 \tag{3.46}$$

$$\text{Subject to:} \quad g_1 : 8x_1 + 6x_2 + x_3 \le 48 \tag{3.47a}$$

$$g_2: 4x_1 + 2x_2 + 1.5x_3 \le 20 \tag{3.47b}$$

$$g_3: 2x_1 + 1.5x_2 + 0.5x_3 \le 8 \tag{3.47c}$$

$$g_4: x_2 \le 5 \tag{3.47d}$$

$$x_1, x_2, x_3 \ge 0$$

The solution to the primal problem is available in Tables 3.20–3.22. Table 3.20 is the initial table. The EBV is x_1 and the LBV is s_3. The pivot row is the third row. The subsequent row operations determine Table 3.21. In this table, the EBV is x_3, the LBV is s_2, and the pivot row is the second row. Table 3.22 is the final table establishing the optimal values for the variables. x_1^*, the optimal number of desks, equals 2. x_3^*, the

Table 3.20 Example 3.5: Primal Problem, Simplex Table 1

x_1	x_2	x_3	s_1	s_2	s_3	s_4	b
8	6	1	1	0	0	0	48
4	2	1.5	0	1	0	0	20
2	1.5	0.5	0	0	1	0	8
0	1	0	0	0	0	1	5
−60	−30	−20	0	0	0	0	f + 0

Table 3.21 Example 3.5: Primal Problem, Simplex Table 2

x_1	x_2	x_3	s_1	s_2	s_3	s_4	b
0	0	−1	1	0	−4	0	16
0	−1	0.5	0	1	−2	0	4
1	0.75	0.25	0	0	0.5	0	4
0	1	0	0	0	0	1	5
0	15	−5	0	0	30	0	f + 240

optimal number of chairs to be made, is 8. No tables will be made. The total revenue is $280.00.

Standard Format—Dual Problem: While the dual problem can be expressed and set up mechanically, it is difficult to associate the design variables in a direct manner as was done in the primal problem. Using y_1, y_2, y_3, and y_4 as the design variables, the problem can be formulated as in Equations (3.48) and (3.49) below using the definition at the beginning of this section. Some associations are possible in light of transposition of the cost coefficients and the constraints. For example, the variable y_1 can be associated with the lumber constraint because in the dual problem its cost coefficient is the lumber constraint of the primal problem. In a similar manner, y_2 is associated with the finishing hours, y_3 with the carpentry hours, and y_4 with the special table constraint. The objective function of the dual problem is expressed in Equation (3.48). In view of the form of the expression of the objective function w, there is a possibility that the design variables in the dual problem can have an economic implication as follows:

- y_1 unit price for a board foot of lumber
- y_2 unit price for a finishing hour
- y_3 unit price for an hour of carpentry
- y_4 this cannot be the unit price as the desk has a price defined in the primal problem—maybe a special price for the desk resource

Table 3.22 Example 3.5: Primal Problem, Simplex Table 3

x_1	x_2	x_3	s_1	s_2	s_3	s_4	b
0	−2	0	1	2	−8	0	24
0	−2	1	0	2	−4	0	8
1	1.25	0	0	−0.5	1.5	0	2
0	1	0	0	0	0	1	5
0	5	0	0	10	10	0	f + 280

The *dual* can be expressed as

$$\text{Minimize } w: \quad 48y_1 + 20y_2 + 8y_3 + 5y_4 \tag{3.48}$$

$$\text{Subject to:} \quad h_1: 8y_1 + 4y_2 + 2y_3 + 0y_4 \geq 60 \tag{3.49a}$$

$$h_2: 6y_1 + 2y_2 + 1.5y_3 + 1y_4 \geq 30 \tag{3.49b}$$

$$h_3: 1y_1 + 1.5y_2 + 0.5y_3 + 0y_4 \geq 20 \tag{3.49c}$$

$$y_1, y_2, y_3 \geq 0 \tag{3.50}$$

Introducing surplus variables (s_1, s_2, s_3) and artificial variables (a_1, a_2, a_3) the dual problem is expressed as a standard LP problem:

$$\text{Minimize } w: \quad 48y_1 + 20y_2 + 8y_3 + 5y_4 \tag{3.48}$$

$$\text{Subject to:} \quad h_1: 8y_1 + 4y_2 + 2y_3 - s_1 + a_1 = 60 \tag{3.51a}$$

$$h_2: 6y_1 + 2y_2 + 1.5y_3 + 1y_4 - s_2 + a_2 = 30 \tag{3.51b}$$

$$h_3: 1y_1 + 1.5y_2 + 0.5y_3 + 0y_4 - s_3 + a_3 = 20 \tag{3.51c}$$

In Phase I of the two-phase approach, the artificial variables are removed from the basis using an artificial cost function A defined as

$$\text{Minimize } A: \quad a_1 + a_2 + a_3 \tag{3.52}$$

The Phase I computations are displayed in Tables 3.23–3.26. Table 3.23 is identified as the simplex Table 1, and it includes some preprocessing by which a_1, a_2, and a_3 are made the basis variables. As a reminder, the last row represents the artificial cost function. From inspection of Table 3.23, y_1 is the EBV. The LBV is a_2 and the pivot row is the second row. Following the standard row operations, Table 3.24 is obtained. In this table, the EBV is y_2 (the largest negative coefficient). The LBV is a_3 (the third

Table 3.23 Example 3.5: Dual Problem, Phase I, Table 1

y_1	y_2	y_3	y_4	s_1	s_2	s_3	a_1	a_2	a_3	b
8	4	2	0	−1	0	0	1	0	0	60
6	2	1.5	1	0	−1	0	0	1	0	30
1	1.5	0.5	0	0	0	−1	0	0	1	20
48	20	8	5	0	0	0	0	0	0	w
−15	−7.5	−4	−1	1	1	1	0	0	0	$A - 110$

Table 3.24 Example 3.5: Dual Problem, Phase I, Table 2

y_1	y_2	y_3	y_4	s_1	s_2	s_3	a_1	a_2	a_3	b
0	1.3333	0	−1.3331	−1	1.3333	0	1	−1.3333	0	20
1	0.3333	0.25	0.1666	0	−0.1666	0	0	0.1666	0	5
0	1.1666	0.25	−0.1667	0	0.1666	−1	0	−0.1666	1	15
0	4	−4	−3	0	8	0	0	−8	0	$w - 240$
0	−2.5	−0.25	1.5	1	−1.5	1	0	2.5	0	$A - 35$

row contains the minimum of {[20/1.333], [5/0.3333], [15/1.6667]}). The pivot row is row 3. Table 3.25 represents the next table. The EBV is s_2. Note in this case there are two candidates for EBV (s_2, s_3). s_2 is chosen arbitrarily. The LBV is a_1. The pivot row is the first row. After the required row manipulations, Table 3.26 is generated. Scanning the last row, there are only positive coefficients. Phase I is now complete. The artificial variables and the artificial cost function can be removed from the LP problem.

Eliminating the artificial variables and the artificial cost function from the problem, Phase II is started with Table 3.27. Note that Table 3.27 contains no information that is not already available in Table 3.26. It only provides an uncluttered table for continuing the application of the Simplex technique, and therefore is not really necessary. In Table 3.27, the EBV is s_3. The LBV is y_1 after a toss between y_1 and s_1. The pivot row is the second row. This row is used to obtain Table 3.28. In this table, note that the value of the basic variable s_1 is 0. This is called a *degenerate basic variable*. The EBV for the next table is y_3. The LBV is s_3. The pivot row is the second row (note: the pivot row can be the row with the degenerate basic solution if the column of the EBV has a positive coefficient in that row). Table 3.29 is the final table and contains the optimal solution. The final solution of the dual problem is

$$y_1^* = 0, y_2^* = 10, y_3^* = 10, y_4^* = 0, s_2^* = 5, w^* = 280$$

To summarize, the solution of the primal problem is

$$x_1^* = 2, x_2^* = 0, x_3^* = 8, s_{1p}^* = 24, S_{4p}^* = 5, z^* = 280$$

The subscript "p" is added to associate the slack variable with the primal problem.

The solution of the dual problem is

$$y_1^* = 0, y_2^* = 10, y_3^* = 10, y_4^* = 0, s_2^* = 5, w^* = 280$$

Some Formal Observations

These observations are justified more formally in many books on linear programming, some of which have been included in the references at the end of this chapter. Here, the formal results are elaborated using the solution to the example discussed in this

Table 3.25 Example 3.5: Dual Problem, Phase I, Table 3

y_1	y_2	y_3	y_4	s_1	s_2	s_3	a_1	a_2	a_3	b
0	0	–0.2857	–1.1429	–1	1.1429	1.1429	1	–1.1429	–1.1429	2.8571
1	0	0.1786	0.2143	0	–0.2143	0.2857	0	0.2143	–0.2857	0.7143
0	1	0.2143	–0.1429	0	0.1429	–0.8571	0	–0.1429	0.8571	12.857
0	0	–4.8571	–2.4286	0	7.4286	3.4286	0	–7.4286	–3.4286	w – 291.43
0	0	0.2857	1.1429	1	–1.1429	–1.1429	0	2.1429	2.1429	A – 2.857

Table 3.26 Example 3.5: Dual Problem, Phase I, Table 4

y_1	y_2	y_3	y_4	s_1	s_2	s_3	a_1	a_2	a_3	b
0	0	–0.25	–1	–0.875	1	1	0.875	–1	–1	2.4999
1	0	0.125	–6E–06	–0.1875	6E–06	0.5	0.1875	–6E–06	–0.5	1.25
0	1	0.25	4E–05	0.125	–4E–05	–1	–0.125	4E–05	1	12.5
0	0	–3.0001	4.9998	6.4998	0.0002	–3.9998	–6.4998	–0.0002	3.9998	w – 310
0	0	0	0	0	0	0	1	1	1	A

Table 3.27 Example 3.5: Dual Problem, Phase II, Table 1

y_1	y_2	y_3	y_4	s_1	s_2	s_3	b
0	0	−0.25	−1	−0.875	1	1	2.5
1	0	0.125	0	−0.1875	0	0.5	1.25
0	1	0.25	0	0.125	0	−1	12.5
0	0	−3	5	6.5	0	−4	$w - 310$

section. No proofs are given. All of the observations may not be relevant to Example 3.5 discussed above, but are included here for completeness.

1. The primal and dual problems have the same value for the optimal objective function, $z^* = w^* = 280$. This is true, however, only if the problems have optimal solutions.
2. If $\mathbf{x}$ is any feasible solution to the primal problem, and $\mathbf{y}$ is any feasible solution to the dual problem, then $w(\mathbf{y}) \geq z(\mathbf{x})$. This feature provides an estimate of the bounds of the dual optimal value if a feasible primal solution is known. This result also holds for the reverse case when a feasible dual is known.
3. If the primal problem is unbounded, the dual problem is infeasible. The *unbounded* problem implies that the objective function value can be pushed to infinite limits. This happens if the feasible domain is not closed. Of course, practical considerations will limit the objective function value to corresponding limits on the design variables. The inverse relationship holds too. If the dual problem is unbounded, the primal is infeasible.
4. If the ith primal constraint is an equality constraint, the ith dual variable is unrestricted in sign. The reverse holds too. If the primal variable is unrestricted in sign, then the dual constraint is an equality.
5. Obtaining primal solution from dual solution:
   (i) If the ith dual constraint is a strict inequality, then the ith primal variable is nonbasic (for optimum solutions only). From the dual solution, the second constraint (3.51b) is satisfied as an inequality. Therefore, the second primal variable x_2 is nonbasic and this is evident in Table 3.22.
   (ii) If the ith dual variable is basic, then the ith primal constraint is a strict equality. From Table 3.29, y_2 and y_3 are basic variables, therefore (3.47b) and (3.47c) must be equalities. This is indeed true.

   These relations can also be established by values of slack variables.
6. Recovering primal solution from final dual table: When the primal and dual problems are in the standard form, the value of the ith primal variable equals the reduced coefficient (that is, the coefficient in the last row of the final table) of the slack/surplus variable associated with the ith dual constraint. In Table 3.29, the values of the reduced cost coefficient corresponding to the surplus variables s_1, s_2, and s_3 are 2, 0, and 8, respectively. These are precisely the optimal values of the primal variables.
7. If the ith dual variable is nonbasic, the value of its reduced cost coefficient is the value of the slack/surplus variable of the corresponding primal constraint. In Table 3.29, y_1 and y_4 are nonbasic variables with reduced cost coefficients of 24 and 5, respectively. These are the values of s_1 (s_{1p}) and s_4 (s_{4p}) in Table 3.22.
8. Obtaining dual solution from primal solution: When the primal and dual problems are in the standard form, the value of the ith dual variable equals the reduced coefficient (that is, the coefficient in the last row of the final table) of the slack/surplus variable associated with the ith primal constraint. In Table 3.22, the values of the reduced cost coefficient corresponding to the surplus variables s_1, s_2, s_3, and s_4 are 0, 10, 10, and 0, respectively. These are precisely the optimal values of the dual variables in Table 3.29.
9. If the ith primal variable is nonbasic, the value of its reduced cost coefficient is the value of the slack/surplus variable of the corresponding dual constraint. In Table 3.22, x_2 is a nonbasic variable with reduced cost coefficients of 5. This is the value of s_2 in Table 3.29. Since x_1 and x_3 are basic, with the reduced cost coefficient of 0, the corresponding slack/surplus s_1 and s_3 variables will be zero, as observed in Table 3.29.

Table 3.28 Example 3.5: Dual Problem, Phase II, Table 2

y_1	y_2	y_3	y_4	s_1	s_2	s_3	b
−2	0	−0.5	−1	−0.5	1	0	0
2	0	0.25	0	−0.375	0	1	2.5
2	1	0.5	0	−0.25	0	0	15
8	0	−2	5	5	0	0	$w - 300$

Table 3.29 Example 3.5: Dual Problem, Phase II, Table 3

y_1	y_2	y_3	y_4	s_1	s_2	s_3	b
2	0	0	−1	−1.25	1	2	5
8	0	1	0	−1.5	0	4	10
−2	1	0	0	0.5	0	−2	10
24	0	0	5	2	0	8	$f - 280$

The above list referred to the primal and dual problems in standard form. A similar list can be obtained for nonstandard form. There will be appropriate modifications for

negative values of variables as well as equality constraints. The listed references can be consulted for further information.

3.5.2 Sensitivity Analysis

The solution to the LP problem is dependent on the values for the coefficients, **c**, **b**, and **A** (also termed *parameters*), involved in the problem. In many practical situations these parameters are only known approximately and may change from its current value for any number of reasons, particularly after a solution has been established. Instead of recomputing a new solution it is possible to obtain a fresh one based on the existing one and its final Simplex table. This issue is significant in practical problems with thousands of variables and constraints. The adaptation of a new solution without re-solving the problem is called *sensitivity analysis*. In LP problems, sensitivity analysis refers to determining the range of parameters for which the optimal solution still has the same variables in the basis, even though values at the solution may change.

Example 3.1 is used for this discussion. Figures are used for illustration rather than a formal proof. Figures are not useful for more than two variables. Generally, most computer codes that solve linear programming problems also perform sensitivity analysis. Revisiting Example 3.1, it was necessary to identify the number of component placement machines of type A and B. The objective is to maximize the number of boards to be manufactured. Constraint g_1 represents the acquisition dollars available. Constraint g_2 represents the floor space constraint. Constraint g_3 represents the number of operators available. The problem statement and the results are reproduced below (note that x_3, x_4 and x_5 are the slack variables)

$$\text{Minimize} \quad f(\mathbf{X}): -990x_1 - 900x_2 - 5250 \tag{3.5}$$

$$\text{subject to:} \quad g_1(\mathbf{X}): 0.4x_1 + 0.6x_2 + x_3 = 8.5 \tag{3.6}$$

$$g_2(\mathbf{X}): 3x_1 - x_2 + x_4 = 25 \tag{3.7}$$

$$g_3(\mathbf{X}): 3x_1 + 6x_2 + x_5 = 70 \tag{3.8}$$

$$x_1 \geq 0; \quad x_2 \geq 0 \quad x_3 \geq 0, \quad x_4 \geq 0, \quad x_5 \geq 0 \tag{3.9}$$

The solution is

$$x_1^* = 10.4762, x_2^* = 6.4285, x_3^* = 0.4524, f^* = -21437.14$$

Changing Cost Coefficient Values (c): First consider the effect of changing the cost coefficients of the design variables. If c_1, the coefficient of design variable x_1, is changed from −990, will the set of basis variables remain the same?

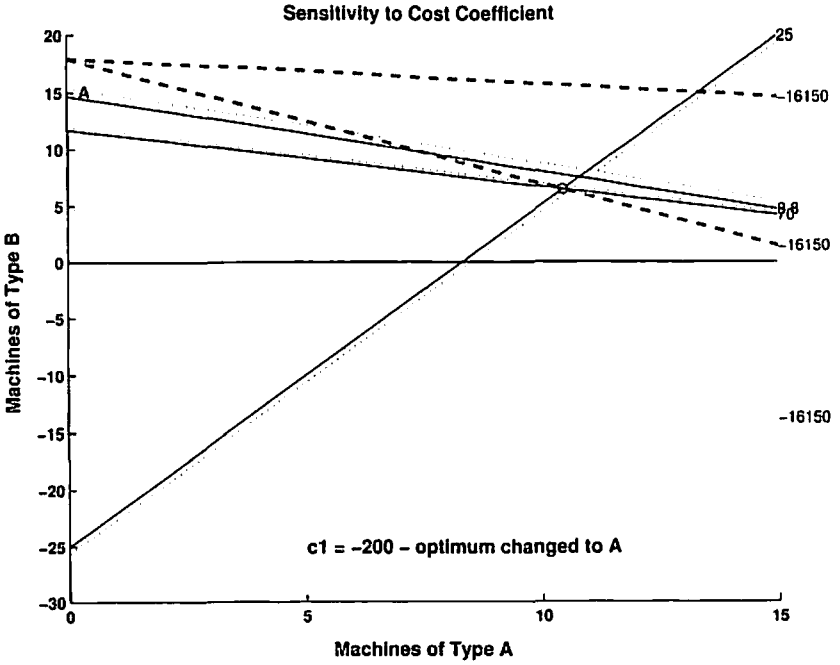

Figure 3.9 Sensitivity analysis, changing cost coefficient.

Changing the cost coefficient changes the slope of the line representing the objective function that may lead to a change in the solution. For example, Equation (3.5) can be written as

$$x_2 = -(990/900)\, x_1 - ([5250 + f]/900)$$

If c_1 were to be made −1900, then the slope of the line will be −(1900/900). The sensitivity due to c_1 can be experienced by running **Sensitivity_cost.m.**[1] The change with respect to the original objective function is shown in animation. The original solution is marked on the figure by a circle. The original solution is unchanged when the magnitude of the slope is raised, for instance $c_1 = -1900$. On the other hand, if c_1 were to be made −200, the dotted line indicates the new cost line and in this case the solution will change to point A. This is shown in Figure 3.9 that is the result of running the m-file. The floor constraint is no longer binding. It can be established by simple calculations that infinitely many solutions are possible if c_1 has a value of −450. In this case the cost function is parallel to the binding constraint g_3. From this discussion it is apparent that to keep the location of the original solution unchanged, c_1 must be greater than −450. Such analysis is possible for all the other cost coefficients involved in this problem. In particular, the analysis should indicate for

[1]Files to be downloaded from the web site are indicated by boldface sans serif type.

what range of cost coefficients the solution would still remain where it is. This is referred to as determining sensitivity to cost coefficients.

Change in the Resource Limits (b Vector): When the values on the right-hand side changes, the constraint lines/planes/boundaries are moved parallel to themselves. This changes the feasible region. Therefore, the optimal values may also change due to the changes in the feasible region. Once again the changes can be illustrated by running the **Sensitivity_rhs.m**. Figure 3.10 illustrates the change in the right-hand side value of the first constraint, b_1, from its value of 8.5 to 6.5. This change causes g_1 to become an active constraint. In this case, the optimal solution has moved from A to B. Note that in the original problem g_1 was not active. In the new formulation it is. However, in sensitivity analysis, the problem is to discover the range of b_1 so that the solution still has the same variables in the basis. If b_1 were to remain above 8.0475, then the solution would still be at A and g_1 will not be an active constraint.

Change in the Coefficient Matrix A: A change in the coefficient matrix is similar in effect to that due to the change in the cost coefficient, while directly

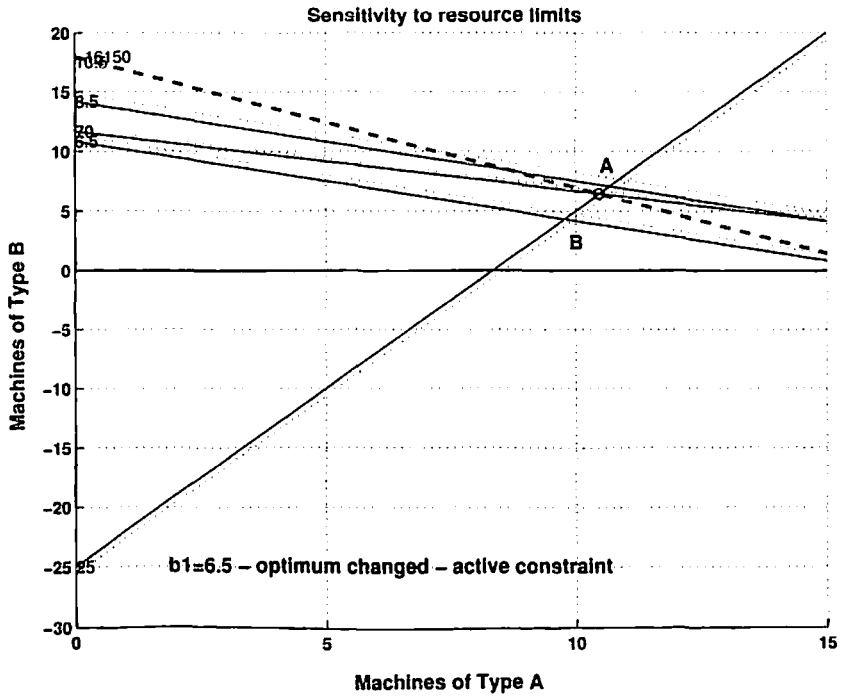

Figure 3.10 Sensitivity analysis, changing right-hand side.

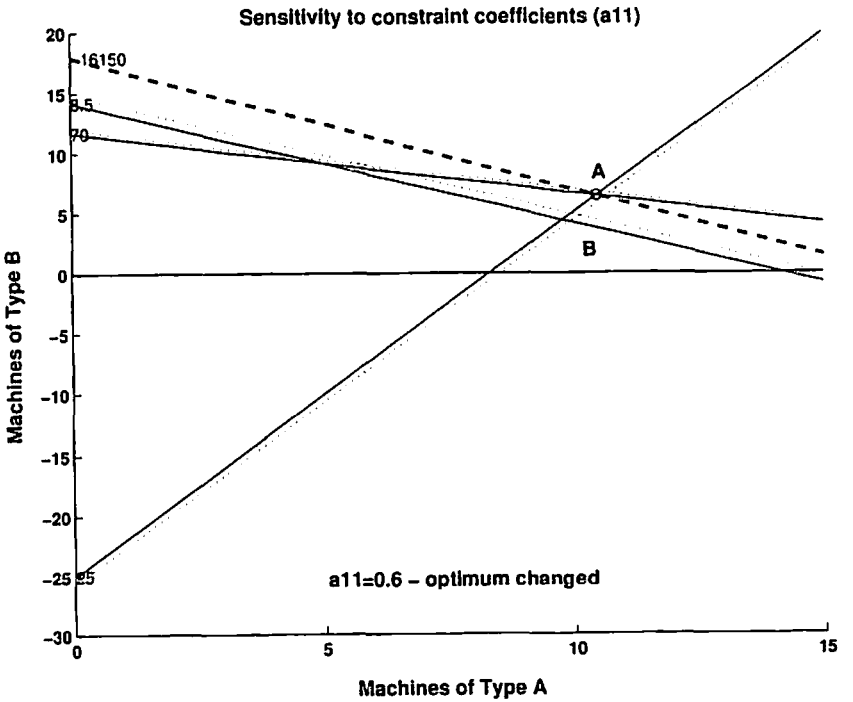

Figure 3.11 Sensitivity analysis, changing cost coefficient.

impacting the problem by changing the feasible region. These changes also depend on whether the variable in the column is a basic variable or a nonbasic variable. **Sensitivity_coeff.m** illustrates the result of changing the coefficient a_{11}. An increase in the value moves the optimal solution to B, while a decrease leaves the solution unchanged. In both situations, x_1 and x_2 are still part of the basis. In the first case, g_1 becomes a binding constraint. This is illustrated in Figure 3.11. The coefficient is in the first column, which corresponds to a basic variable. Changing coefficient values in the first column in other constraints will yield similar information. The reader is encouraged to try other changes in the coefficient using the m-files available in the code section of this chapter.

REFERENCES

1. Dantzig, G. B., *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

2. Luenberger, D. G., *Linear and Nonlinear Programming*, Addison–Wesley, Reading, MA, 1984.

3. Winston, W. L., *Introduction to Mathematical Programming, Applications and Algorithms*, Duxbury Press, Belmont, CA, 1995.
4. Arora, J. S., *Introduction to Optimal Design*, McGraw–Hill, New York, 1989.
5. Williams, G., *Linear Algebra with Applications*, Wm. C. Brown Publishers, Dubuque, IA, 1991.
6. Noble, B., and Daniel, J. W., *Applied Linear Algebra*, Prentice–Hall, Englewood Cliffs, NJ, 1977.

PROBLEMS

(For all two-variable problems, provide a graphical definition/solution of the problem.)

3.1 Solve the following linear programming problem:

Min $f(x_1, x_2)$: $x_1 + x_2$

Sub: $3x_1 - x_2 \le 3$

$x_1 + 2x_2 \le 5$

$x_1 + x_2 \le 4$

$x_1 \ge 0; x_2 \ge 0$

3.2 Solve the following linear programming problem:

Max $f(x_1, x_2)$: $x_1 + x_2$

Sub: $3x_1 - x_2 \le 3$

$x_1 + 2x_2 \le 5$

$x_1 + x_2 \le 4$

$x_1 \ge 0; x_2 \ge 0$

3.3 Solve the following linear programming problem:

Min $f(x_1, x_2)$: $x_1 - x_2$

Sub: $3x_1 - x_2 \le 3$

$x_1 + 2x_2 \le 5$

$x_1 + x_2 \le 4$

$x_1 \ge 0; x_2 \ge 0$

3.4 Solve the following linear programming problem:

Max $f(x_1, x_2)$: $x_1 + x_2$

Sub: $3x_1 - x_2 \ge 3$

$x_1 + 2x_2 \le 5$

$x_1 + x_2 \le 4$

$x_1 \ge 0; x_2 \ge 0$

3.5 Solve the following linear programming problem:

Min $f(x_1, x_2)$: $x_1 + x_2$

Sub: $3x_1 - x_2 \le 3$

$x_1 + 2x_2 \le 5$

$x_1 + x_2 \le 4$

$x_1 \ge 0$; x_2 unrestricted in sign

3.6 The local bookstore must determine how many of each of the four new books on photonics it must order to satisfy the new interest generated in the discipline. Book 1 costs \$75, will provide a profit of \$13, and requires 2 inches of shelf space. Book 2 costs \$85, will provide a profit of \$10, and requires 3 inches of shelf space. Book 3 costs \$65, will provide a profit of \$8, and requires 1 inch of shelf space. Book 4 costs \$100, will provide a profit of \$15 and requires 4 inches of shelf space. Find the number of each type that must be ordered to maximize profit. Total shelf space is 100 inches. Total amount available for ordering is \$5000. It has been decided to order at least a total of 10 of Book 2 and Book 4.

3.7 The local community college is planning to grow the biotechnology offering through new federal and state grants. An ambitious program is being planned for recruiting at least 200 students from in and out of state. They are to recruit at least 40 out-of-state students. They will attempt to recruit at least 30 students who are in the top 20% of their graduating high school class. Current figures indicate that about 8% of the applicants from in state, and 6% of the applicants from out of state belong to this pool. They also plan to recruit at least 40 students who have AP courses in biology. The data suggest that 10% and 15% of in-state and out-of-state applicants, respectively, belong to this pool. They anticipate that the additional cost per student is \$800 for each in-state student and \$1200 for each out-of-state student. Find their actual enrollment needed to minimize cost and their actual cost.

3.8 Figure 3.12 represents an optimization problem to determine the maximum total number of the smaller rectangles that will fit within the larger one. The dimensions are indicated on the figure. The number of the larger rectangle should be at least 5 more than the smaller ones. The rectangles cannot be rotated. No fractional rectangles are allowed.

(Optional: confirm all solutions using the Optimization Toolbox from MATLAB.)

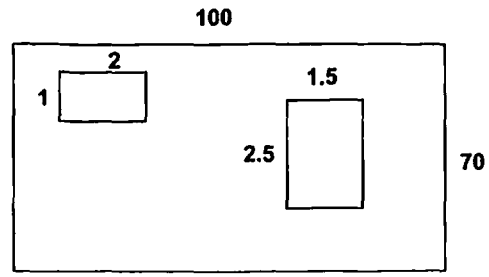

Figure 3.12 Problem 3.8.

4

NONLINEAR PROGRAMMING

Optimization problems whose mathematical models are characterized by nonlinear equations are called *Nonlinear Programming* (NLP) *problems.* In Chapter 3 it was noted that these problems also fell into the category of *mathematical programming problems*. Engineering design problems are mostly nonlinear. In Chapter 2 several problems were examined graphically and it was evident that curvature and the gradient of the functions involved had a significant influence on the solution. In subsequent chapters we will continue to center the discussion of optimality conditions and numerical techniques around two-variable problems because the ideas can also be expressed graphically. Extension to more than two variables is quite straightforward and is most simple when the presentation is made through the use of vector algebra. MATLAB will be utilized for all graphic needs.

Traditionally, there is a *bottom-up* presentation of material for nonlinear optimization. *Unconstrained* problems are discussed first followed by *constrained* problems. For constrained problems the *equality constrained* problem is discussed first. A similar progression is observed with regard to the number of variables. A single-variable problem is introduced, followed by two variables which is then extended to a general problem involving n variables. This order allows incremental introduction of new concepts, but primarily allows the creative use of existing rules to establish solutions to the extended problems.

An analytical foundation is essential to understand and establish the conditions that the optimal solution will have to satisfy. This is not for the sake of mathematical curiosity, but is an essential component of the numerical technique: notably the stopping criteria. The necessary mathematical definitions and illustrations are introduced in this chapter. References are available for refreshing the calculus and the

numerical techniques essential to the development of NLP [1,2]. The books familiar to the reader should do admirably. This chapter also introduces the *symbolic computation* (computer algebra) resource available in MATLAB, namely, Symbolic Math Toolbox [3].

4.1 PROBLEM DEFINITION

In NLP it is not essential that all the functions involved be nonlinear. It is sufficient if just one of them is nonlinear. There are many examples in engineering in which only the objective function is nonlinear while the constraints are linear. If in this case the objective function is a quadratic function, these problems are termed *linear quadratic problems* (LQP). Optimization problems for the most part rely on experience to identify the mathematical model comprising of the design variables, objective, and the constraints. A knowledge of engineering, or the appropriate discipline, is also essential to establish a mathematical model. Primarily, this involves determining the functional relationships among the design variables. The remaining task then is to establish the solution.

How does one establish the solution to the nonlinear optimization problem?

In mathematics (after all at this stage there is a mathematical model for the problem) the *solution* is obtained by satisfying the *necessary* and *sufficient* conditions related to the class of problems. The necessary conditions are those relations that a candidate for the optimum solution *must satisfy*. If it does, then, and this is important, it *may* be an optimal solution. To qualify a design vector $\mathbf{X}^p$ ($\mathbf{X}$ represents the design vector) as an optimum, it must satisfy additional relations called the *sufficient* conditions. Therefore, an optimum solution must satisfy both necessary and sufficient conditions. This chapter establishes these conditions for the optimization problem. Example 4.1 is established next and is used in several ways in the remainder of the chapter to develop the conditions mentioned above. Once these conditions are available, the numerical techniques in optimization will incorporate them to establish the solution.

4.1.1 Problem Formulation—Example 4.1

The problem is restricted to two variables to draw graphical support for some of the discussions. There are two constraints, which, during the development of this chapter, may switch between equality and inequality constraints to illustrate related features.

Problem: Find the rectangle of the largest area (in the positive quadrant) that can be transcribed within a given ellipse and satisfy a prescribed linear constraint.

From the problem specification, the ellipse will provide an inequality constraint while the linear relation among the variables will provide an equality constraint for this example.

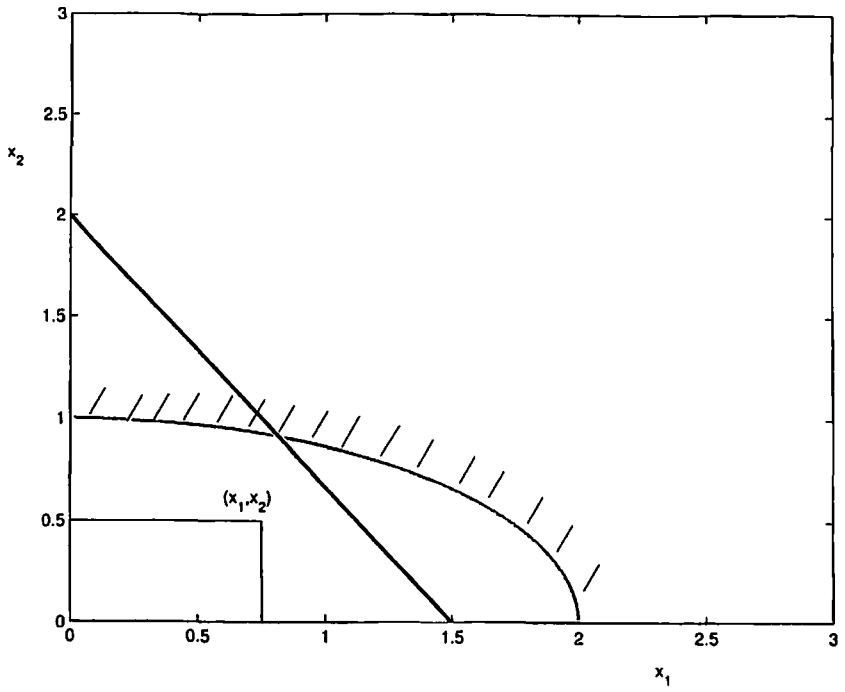

Figure 4.1 Constraints of Example 4.1.

Mathematical Model: Figure 4.1 captures the essence of the problem. Code files **Ex41_1.m**[1] and **create_ellipse.m** are files necessary for MATLAB to create the figure. There are two variables x_1 and x_2. There are standard mathematical expressions for the ellipse, and the straight line is no problem after Chapter 3.

Standard Format: The standard format of the NLP is reproduced here for convenience:

$$\text{Minimize} \quad f(x_1, x_2, \ldots, x_n) \tag{4.1}$$

$$\text{Subject to:} \quad h_k(x_1, x_2, \ldots, x_n) = 0, \; k = 1, 2, \ldots, l \tag{4.2}$$

$$g_j(x_1, x_2, \ldots, x_n) \le 0, \; j = 1, 2, \ldots, m \tag{4.3}$$

$$x_i^l \le x_i \le x_i^u \qquad i = 1, 2, \ldots, n \tag{4.4}$$

In vector notation

[1]Files to be downloaded from the web site are indicated by boldface sans serif type.

$$\text{Minimize} \quad f(\mathbf{X}), [\mathbf{X}]_n \tag{4.5}$$

$$\text{Subject to:} \quad [\mathbf{h}(\mathbf{X})]_l = 0 \tag{4.6}$$

$$[\mathbf{g}(\mathbf{X})]_m \le 0 \tag{4.7}$$

$$\mathbf{X}^{\text{low}} \le \mathbf{X} \le \mathbf{X}^{\text{up}} \tag{4.8}$$

For the specific problem being discussed, and referring to Figure 4.1, the design variables are the coordinate values x_1 and x_2 that allow the computation of the rectangular area. The optimization problem is

$$\text{Minimize} \quad f(x_1, x_2): -x_1 x_2 \tag{4.9}$$

$$\text{Subject to:} \quad h_1(x_1, x_2): 20 x_1 + 15 x_2 - 30 = 0 \tag{4.10}$$

$$g_1(x_1, x_2): (x_1^2/4) + (x_2^2) - 1 \le 0 \tag{4.11}$$

$$0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \tag{4.12}$$

The side constraints in (4.12) can also be postulated as one-sided and can be written as

$$x_1 \ge 0; \qquad x_2 \ge 0 \tag{4.13}$$

4.1.2 Discussion of Constraints

Using the relations (4.9)–(4.12) several additional classes of problems can be described by including only a subset of the relations. They are examined below.

Unconstrained Problem: There are no functional constraints although the side constraints are necessary to keep the solution finite. For this example,

$$\text{Minimize} \quad f(x_1, x_2): -x_1 x_2 \tag{4.9}$$

In this problem, if the design variables are unbounded at the upper limit, then the solution would be at the largest positive value of x_1 and x_2. A two-sided limit for the design variables is usually a good idea. The designer does have the responsibility of defining an acceptable design space.

Equality Constrained Problem 1: The functional constraints in this problem are only equalities. With reference to Example 4.1 the following problem can be set up (after changing the inequality to an equality):

$$\begin{aligned} \text{Minimize} \quad & f(x_1, x_2): -x_1 x_2 \\ \text{Subject to:} \quad & h_1(x_1, x_2): 20 x_1 + 15 x_2 - 30 = 0 \\ & h_2(x_1, x_2): (x_1^2/4) + (x_2^2) - 1 = 0 \\ & 0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \end{aligned}$$

Intuitively, such a problem may not be optimized since the two constraints by themselves should establish the values for the two design variables. The arguments used in LP for acceptable problem definition are also valid here. There is always the possibility of multiple solutions—which is a strong feature of nonlinear problems. In such an event the set of variables that yield the lowest value for the objective will be the optimal solution. Note that such a solution is obtained by scanning the acceptable solutions rather than through application of any *rigorous* conditions.

Equality Constrained Problem 2: If the problem were to include only one of the constraints, for example,

$$\begin{aligned} \text{Minimize} \quad & f(x_1, x_2): -x_1 x_2 \\ \text{Subject to:} \quad & h_2(x_1, x_2): (x_1^2/4) + (x_2^2) - 1 = 0 \\ & 0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \end{aligned}$$

This is a valid optimization problem. A similar problem can be defined with the first constraint by itself.

Inequality Constrained Problem: In this case the constraints are all inequalities. A variation on Example 4.1 would be (the equality constraint is transformed to an equality constraint)

$$\begin{aligned} \text{Minimize} \quad & f(x_1, x_2): -x_1 x_2 \\ \text{Subject to:} \quad & g_1(x_1, x_2): 20 x_1 + 15 x_2 - 30 \le 0 \\ & g_2(x_1, x_2): (x_1^2/4) + (x_2^2) - 1 \le 0 \\ & 0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \end{aligned}$$

Like its counterpart in linear programming, this is a valid optimization problem. Equally valid would be a problem that included just one of the constraints or any number of inequality constraints.

It is essential to understand both the nature and the number of constraints as well as how they affect the problem. In general, equality constraints are easy to handle mathematically, difficult to satisfy numerically, and more restrictive on the search for the solution. Inequality constraints are difficult to resolve mathematically and are more flexible with respect to the search for the optimal solution as they define a larger feasible region. A well-posed problem requires that some constraints are active (equality) at the optimal solution as otherwise it would be an unconstrained solution.

4.2 MATHEMATICAL CONCEPTS

Like LP, some mathematical definitions are necessary before the necessary and sufficient conditions for the NLP can be established. Definitions are needed for both the analytical discussion as well as numerical techniques. MATLAB provides a Symbolic Math Toolbox which permits symbolic computation integrated in the numerical environment of MATLAB. This allows the user to explore problems in calculus, linear algebra, solutions of system of equations, and other areas. In fact, using symbolic computation students can easily recover the prerequisite information for the course. A short hands-on exercise to symbolic computation is provided.[1] The primary definitions we need are derivatives, partial derivatives, matrices, derivatives of matrices, and solutions of nonlinear equations.

4.2.1 Symbolic Computation Using MATLAB

One of the best ways to get familiar with symbolic computation is to take the quick online introduction to the Symbolic Math Toolbox available in the MATLAB Demos dialog box [4]. The following sequence locates the tutorial:

```
>> demos
Symbolic Math --> Introduction
```

The computational engine executing the symbolic operations in MATLAB is the kernel of Maple marketed and supported by Waterloo Maple, Inc. If the reader is already familiar with Maple, then MATLAB provides a hook through which Maple commands can be executed in MATLAB. The symbolic computation in MATLAB is performed using a *symbolic object* or *sym*. This is another data type like the number and string data types used in earlier exercises. The Symbolic Math Toolbox uses *sym* objects to represent symbolic variables, expressions, and matrices.

In the exercise that follows, a function of one variable, and two functions of two variables (constraints from Example 4.1) are used for illustration. Drawing on the author's classroom experience this preliminary discussion is in terms of variables x and y for improved comprehension. In later sections, subscripts on x are used to define multiple variables so that the transition to the general problem can be facilitated using vector description. The functions in these exercises are

$$\begin{aligned} & f(x) = 12 + (x - 1)^2 (x - 2) (x - 3) \\ & g_1(x,y): 20 x + 15 y - 30 \\ & g_2(x,y): x^2/4 + y^2 - 1 \end{aligned}$$

[1]The release of the new *Student Edition* includes the symbolic toolbox along with the basic MATLAB software.

The following MATLAB session was captured as a diary file and edited in a text editor. Therefore the MATLAB prompt does not appear. The boldface words are commands that the reader will type at the command line.

```
x = sym('x')   % defining x as a single symbolic object
x =
x

syms y f g1 g2 g  % definition of multiple objects
whos   % types of variables in the workspace
 Name      Size         Bytes     Class
 f         1x1          126       sym object
 g         1x1          126       sym object
 g1        1x1          128       sym object
 g2        1x1          128       sym object
 x         1x1          126       sym object
 y         1x1          126       sym object

Grand total is 14 elements using 760 bytes

f= 12 + (x-1)*(x-1)*(x-2)*(x-3)  % constructing f
f =
12+(x-1)^2*(x-2)*(x-3)

diff(f)  % first derivative
 ans =
 2*(x-1)*(x-2)*(x-3)+(x-1)^2*(x-3)+(x-1)^2*(x-2)

% note the chain rule for derivatives
% note the independent variable is assumed to be x

diff(f,x,2)    % the second derivative wrt x
 ans =
 2*(x-2)*(x-3)+4*(x-1)*(x-3)+4*(x-1)*(x-2)+2*(x-1)^2

diff(f,x,3)    % the third derivative wrt x
 ans =
 24*x-42

g1= 20*x +15*y -30   % define g1
g1 =
 20*x+15*y-30

g2 = 0.25*x + y -1;  % define g2

% g1,g2 can only have partial derivatives
% independent variables have to be identified

diff(g1,x) % partial derivative
ans =
 20
```



```
diff(g1,y)
ans =
 15

g = [g1;g2]  % g column vector based on g1,g2
g =
 [ 20*x+15*y-30]
 [    1/4*x+y-1]

% g can be the constraint vector in optimization
% problems
% the partial derivatives of g with respect to design
% variables is called the Jacobian matrix
% the properties of this matrix are important for
% numerical techniques

xy = [x y]; % row vector of variables
J = jacobian(g,xy)  % calculating the Jacobian

J =
 [  20,  15]
 [ 1/4,   1]

ezplot(f)  % a plot of f for -2 pi  x  2 pi (default)

ezplot(f,[0,4])  % plot between 0 <= x  <= 4

df = diff(f);
hold on
ezplot(df,[0,4])  % plotting function and derivative
% combine with MATLAB graphics - draw a line
line([0 4],[0 0],'Color','r')
g
g =
 [ 20*x+15*y-30]
 [    1/4*x+y-1]

% to evaluate g at x = 1, y = 2.5
subs(g,{x,y},{1,2.5})
ans =
  27.5000
   1.7500
```

Additional symbolic computations will be introduced through code as appropriate. Note that the result of both numeric and symbolic computations can be easily combined along with graphics to provide a powerful computing environment.

4.2.2 Basic Mathematical Concepts

The basic mathematical elements in the discussion of NLP are derivatives, partial derivatives, vectors, matrices, Jacobian, and Hessian. We have used the Symbolic Math Toolbox in the previous section to calculate some of these quantities without defining them. These topics will have been extensively covered in the foundation courses on mathematics in most disciplines. A brief review is offered in this section using MATLAB. This opportunity will also be utilized to increase familiarity with the Symbolic Math Toolbox and incorporate it with the MATLAB commands that were used in the earlier chapters.

Function of One Variable: $f(x)$ identifies a function of one variable:

$$f(x) = 12 + (x-1)^2 (x-2) (x-3)$$

is used as a specific example of such a function. The *derivative* of the function at the location x is written as

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} \tag{4.14}$$

x is the point about which the derivative is computed. Δx is the distance to a neighboring point whose location therefore will be $x + \Delta x$. The value of the derivative is obtained as a limit of the ratio of the difference in the value of the function at the two points (Δf) to the distance separating the two points (Δx), as this separation is reduced to zero. The computation of the derivative for the specific example is usually obtained using the product rule. Results from the exercise on symbolic computation provide the result

$$\frac{df}{dx} = 2*(x-1)*(x-2)*(x-3) + (x-1)\,\hat{}\,2*(x-3) + (x-1)\,\hat{}\,2*(x-2)$$

Table 4.1 illustrates the limiting process for the derivative of the function chosen as the example. The derivative is being evaluated at point 3. As the displacements are reduced, the value of the ratio approaches the value of 4 which is the actual value of the derivative. When Δx is large (with a value of 1), the ratio is 18. This is significantly different from the value of 4. With Δx of 0.1 the value is around 4.851. With further reduction to 0.001 the ratio has a value of 4.008. From these computations it can be expected that as Δx approaches 0 the ratio will reach the exact value of 4.0.

Numerical Derivative Computation: Many numerical techniques in NLP require the computation of derivatives. Most software implementing these techniques do not use symbolic computation. Automation and ease of use require that these derivatives be computed numerically. The results in Table 4.1 justify the numerical computation of a derivative through a technique called the *first forward difference* [5]. Using a very small perturbation Δx, the derivative at a point is numerically calculated as the ratio

Table 4.1 Calculation of Derivative

x	Δx	$x + \Delta x$	Δf	$\Delta f/\Delta x$	Derivative
3	1	4	18	18	4
3	0.1	3.1	0.4851	4.851	4
3	0.01	3.01	0.0408	4.0805	4
3	0.001	3.001	0.004	4.008	4

of the change in the function value, Δf, to the change in the displacement, Δx. For the example, the derivative at $x = 3$ with $\Delta x = 0.001$ is obtained as

$$\left.\frac{df}{dx}\right|_{x=3} = \frac{f(3+0.001) - f(3)}{0.001} = 4.008 \tag{4.15}$$

The derivative for the single-variable function at any value x is also called the *slope* or the *gradient* of the function at that point. If a line is drawn tangent to the function at the value x, the tangent of the angle that this line makes with the x-axis will have the same value as the derivative. If this angle is θ, then

$$\frac{df}{dx} = \tan\theta$$

Figure 4.2 illustrates the tangency property of the derivative.

MATLAB Code: Figure 4.2 was created by the code **tangent.m**. In the same directory, the code **derivative.m** provides the illustration of tangency through a figure animation in MATLAB. In the figure window both the function (drawn as the curve) as well as the line representing Δf and Δx are drawn. As the neighboring points are closer, it is evident from the figure that the curve can be approximated by a straight line. When $\Delta x = 0.001$, the derivative is tangent to the curve as they are indistinguishable.

Higher Derivatives: The derivative of the derivative is the called the *second derivative*. Formally it is defined as

$$\frac{d^2f}{dx^2} = \frac{d}{dx}\left(\frac{df}{dx}\right) = \lim_{\Delta x \to 0} \frac{\Delta\,(df/dx)}{\Delta x} = \lim_{\Delta x \to 0} \frac{df/dx|_{x+\Delta x} - df/dx|_x}{\Delta x} \tag{4.16}$$

Similarly the *third derivative* is defined as

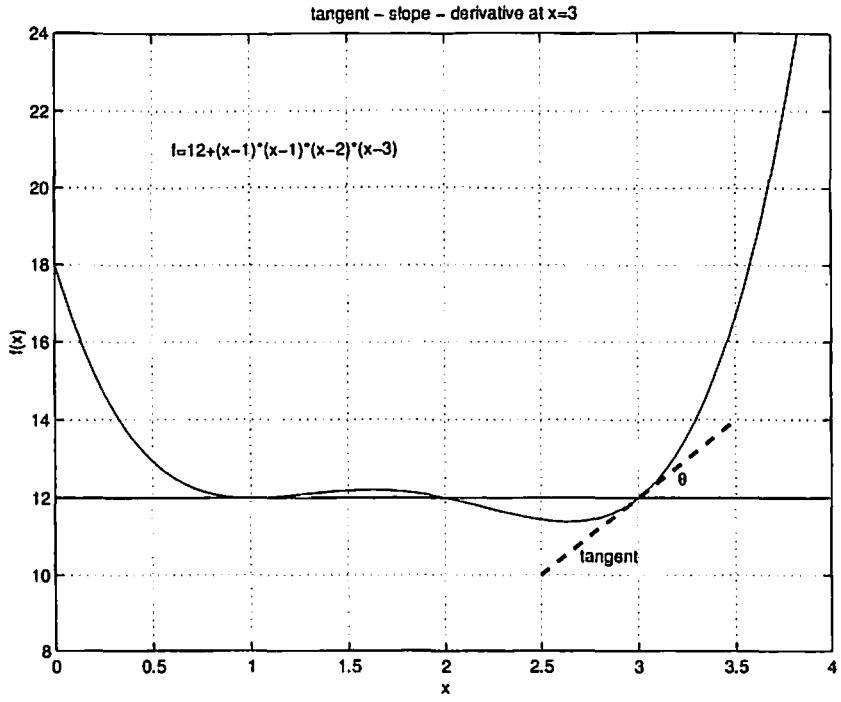

Figure 4.2 Illustration of the tangent.

$$\frac{d^3f}{dx^3} = \frac{d}{dx}\left(\frac{d^2f}{dx^2}\right) \tag{4.17}$$

This can go on provided the function has sufficient higher-order dependence on the independent variable x.

Function of Two Variables: The two-variable function

$$f(x,y): x^2/4 + y^2 - 1 \tag{4.18}$$

is chosen to illustrate the required mathematical concepts. The first important feature of two or more variables is that the derivatives defined for a single variable (ordinary derivatives) do not apply. The equivalent concept here is the *partial derivative*. Partial derivatives are defined for each independent variable. The partial derivative is denoted by the symbol ∂. These derivatives are obtained in the same way as the ordinary derivative except the other variables are held at a constant value. In the example, when computing the partial derivative of x the value of y is not allowed to change.

$$\left.\frac{\partial f}{\partial x}\right|_{(x,y)} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x, y) - f(x,y)}{\Delta x} \tag{4.19}$$

The above relation expresses the partial derivative of the function f with respect to the variable x. Graphically this suggests that the two points are displaced horizontally (assuming the x-axis is horizontal). In the above expression (and prior ones too), the subscript after the vertical line establishes the point about which the partial derivative with respect to x is being evaluated. (x,y) represents any/all points. A similar expression can be written for the partial derivative of f with respect to y. For this example:

$$\frac{\partial f}{\partial x} = \frac{x}{2}; \quad \frac{\partial f}{\partial y} = 2y$$

For the point $x = 2$, $y = 1$:

$$f(2,1) = 1; \quad \frac{\partial f}{\partial x} = 1; \quad \frac{\partial f}{\partial y} = 2$$

Until this point we have made use of symbols representing changes in values (functions, variables) without a formal definition. In this book

$\Delta(\,)$: represents finite/significant changes in the quantity $(\,)$
$d(\,), \delta(\,)$: represents differential/infinitesimal changes in $(\,)$

Changes in functions occur due to changes in the variables. From calculus [1] the differential change in $f(x,y)$ (df) due to the differential change in the variables x (dx) and y (dy) is expressed as

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy \tag{4.20}$$

For convenience and simplicity, the subscript representing the point where the expression is evaluated is not indicated. The definition of the partial derivative is apparent in the above expression as holding y at a constant value implies that $dy = 0$. Another interpretation for the partial derivative can be observed from the above definition: *change of the function per unit change in the variables.*

Gradient of the Function: In the function of a single variable, the derivative was associated with the slope. In two or more variables the slope is equivalent to the gradient. The gradient is a vector, and at any point represents *the direction in which the function will increase most rapidly.* Examining the conventional objective of NLP, *minimization of objective functions*, the gradient has a natural part to play in the development of methods to solve the problem. The gradient is composed of the partial derivatives organized as a vector. Vectors in this book are column vectors unless other noted. The gradient has a standard mathematical symbol. It is defined as

$$\nabla \mathbf{f} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix}^{\mathrm{T}} \tag{4.21}$$

At this stage it is appropriate to consolidate this information using graphics. The graphical description of the example defined in Equation (4.18) has to be three dimensional as we need one axis for x, another for y, and the third for $f(x,y)$. Chapter 2 introduced three-dimensional plotting using MATLAB. A more useful description of the problem is to use contour plots. Contour plots are drawn for specific values of the function. In the illustration the values for the contours of f are 0, 1, 2, and 3. In Figure 4.3, two kinds of contour plots are shown. In the top half a three-dimensional (3D) contour plot is shown, while on the lower half the same information is presented as a two-dimensional (2D) contour plot (this is considered the standard contour plot). The 2D contour plot is more useful for graphical support of some of the ideas in NLP. The 3D contour will be dispensed with in the remainder of the book.

MATLAB Code: **Fig4_3.m.** The annotation in Figure 4.3 (including the tangent line and the gradient arrow) was done through the plot editing commands available on the menu bar in the figure window (MATLAB version 5.2 and later). A major portion of the plots are generated through the statements in the m-file indicated above. The code mixes numeric and symbolic computation. It allows multiple plots and targets graphic commands to specific plots.

Discussion of Figure 4.3: In Figure 4.3 point P is on contour $f = 0$. Point Q is on the contour $f = 2$. Point S is on the contour $f = 1$. Point R has the same y value as point P and the same x value as point Q. For the values in the figure the contour value is 0.75 (this value should be displayed in MATLAB window when the code **Fig4_3.m** is run).

line PQ is a measure Δf

line PR represents changes in Δf when Δy is 0

line RQ represents changes in Δf when Δx is 0

Mathematically, Δf can only be estimated by adding the changes along the lines PR and RQ since the definition of the partial derivatives only permits calculating changes along the coordinate directions.

At point S the dotted line is the tangent. The gradient at the same point is normal (perpendicular) to the tangent and is directed toward increasing the value of the function (indicated by the arrow). By definition, if df represents a differential move along the gradient at any point, then (dx, dy are measured along the gradient vector)

$$df = \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial y} dy \tag{4.22}$$

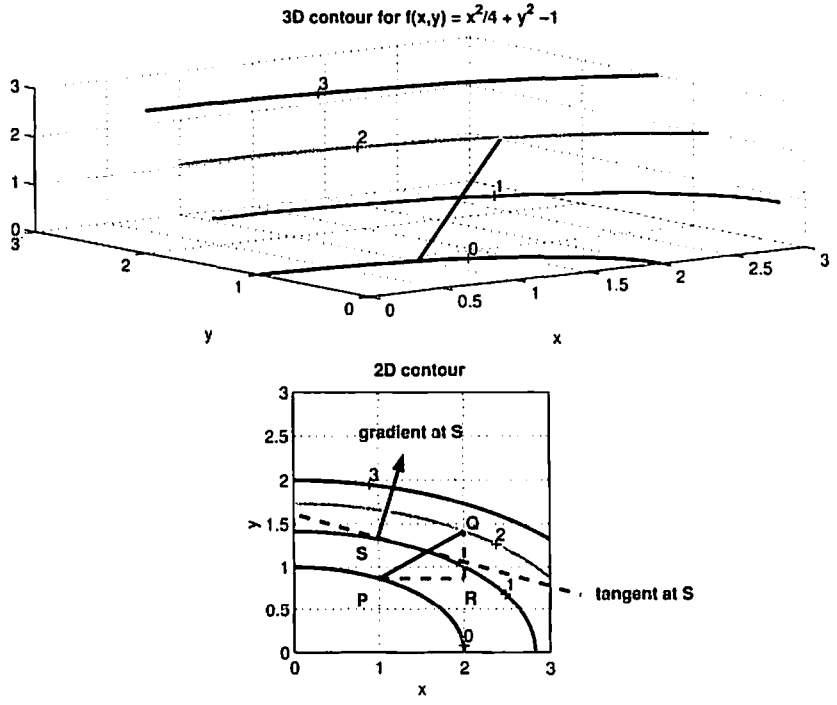

Figure 4.3 Gradient and tangent line at a point.

From the figure the value of f will change along gradient direction. If df represents a differential move along the tangent line, then ($d\tilde{x}$, $d\tilde{y}$ are measured along tangent line)

$$df = \frac{\partial f}{\partial \tilde{x}} d\tilde{x} + \frac{\partial f}{\partial \tilde{y}} d\tilde{y} = 0 \tag{4.23}$$

Equation (4.23) should be zero because moving tangentially (a small amount) the value of f is not changed.

Jacobian: The Jacobian [J] defines a useful way to organize the gradients of several functions. Using three variables and two functions $f(x,y,z)$ and $g(x,y,z)$ the definition of the Jacobian is

$$[\mathbf{J}] = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} & \frac{\partial g}{\partial z} \end{bmatrix} \tag{4.24}$$

In Equation (4.24) the gradients of the function appear in the same row. The first row is the gradient of f while the second row is the gradient of g. If the two functions are collected into a column vector, the differential changes $[df\ dg]^T$ in the functions, due to the differential change in the variables $[dx\ dy\ dz]$, can be expressed as a matrix multiplication using the Jacobian

$$\begin{bmatrix} df \\ dg \end{bmatrix} = [\mathbf{J}] \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} \tag{4.25}$$

which is similar to Equation (4.20).

Hessian: The Hessian matrix [H] is the same as the matrix of second derivatives of a function of several variables. For $f(x,y)$

$$[\mathbf{H}] = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x^2} & \dfrac{\partial^2 f}{\partial x \partial y} \\ \dfrac{\partial^2 f}{\partial y \partial x} & \dfrac{\partial^2 f}{\partial y^2} \end{bmatrix} \tag{4.26}$$

The Hessian matrix is symmetric. For the example defined by Equation (4.18),

$$[\mathbf{H}] = \begin{bmatrix} 1/2 & 0 \\ 0 & 2 \end{bmatrix}$$

Function of n Variables: For $f(\mathbf{X})$, where $[\mathbf{X}] = [x_1, x_2, \ldots, x_n]^T$, the gradient is

$$\nabla f = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} & \dfrac{\partial f}{\partial x_2} & \cdots & \dfrac{\partial f}{\partial x_n} \end{bmatrix}^T \tag{4.27}$$

The Hessian is

$$[\mathbf{H}] = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & & \cdots \\ \vdots & & \ddots & \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1} & & & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{4.28}$$

Equations (4.27) and (4.28) will appear quite often in succeeding chapters. A few minutes of familiarization will provide sustained comprehension later.

4.2.3 Taylor's Theorem/Series

Single Variable: The Taylor series is a useful mechanism to approximate the value of the function $f(x)$ at the point $(x_p + \Delta x)$ if the function is completely known at point x_p. The expansion is (for finite n)

$$f(x_p + \Delta x) \equiv f(x_p) + \left.\frac{df}{dx}\right|_{x_p} (\Delta x) + \frac{1}{2!} \left.\frac{d^2 f}{dx^2}\right|_{x_p} (\Delta x)^2 + \cdots + \frac{1}{n!} \left.\frac{d^n f}{dx^n}\right|_{x_p} (\Delta x)^n \tag{4.29}$$

The series is widely used in most disciplines to establish continuous models. It is the mainstay of many numerical techniques, including those in optimization. Equation (4.29) is usually truncated to the first two or three terms with the understanding the approximation will suffer some error whose order depends on the term that is being truncated:

$$f(x_p + \Delta x) \equiv f(x_p) + \left.\frac{df}{dx}\right|_{x_p} (\Delta x) + \frac{1}{2!} \left.\frac{d^2 f}{dx^2}\right|_{x_p} (\Delta x)^2 + O(\Delta x)^3 \tag{4.30}$$

If the first term is brought to the left, the equation, discarding the error term, can be written as

$$\Delta f = f(x_p + \Delta x) - f(x_p) = \left.\frac{df}{dx}\right|_{x_p} (\Delta x) + \frac{1}{2!} \left.\frac{d^2 f}{dx^2}\right|_{x_p} (\Delta x)^2 \tag{4.31}$$

In Equation (4.31), the first term on the right is called the *first-order/linear variation* while the second term is the *second-order/quadratic variation.*

Figure 4.4 demonstrates the approximations using Taylor's series of various orders at point 2.5 with respect to the original function (red). The principal idea is to deal with the approximating curve which has known properties instead of the original curve. The constant Taylor series is woefully inadequate. The linear expansion is only marginally better. The quadratic expansion approximates the right side of the function compared to the left side which is in significant error. The fifth-order expansion is definitely acceptable in the range shown. To see how the Taylor series is used, consider the quadratic curve about the point $x = 2.5$. At this point the value of the function $f(2.5) = 11.4375$, the value of the first derivative $f'(2.5) = -0.75$, and the value of the second derivative is $f''(2.5) = 4$. Using Equation (4.31)

$$f(2.5 + \Delta x) = 11.4375 + (-0.75)\Delta x + \frac{1}{2}(4)(\Delta x)^2 \tag{4.32}$$

For different values of Δx, both positive and negative, the value of $f(x)$ can be obtained. The plot of Equation (4.32) should be the same as the one labeled quadratic in Figure 4.4.

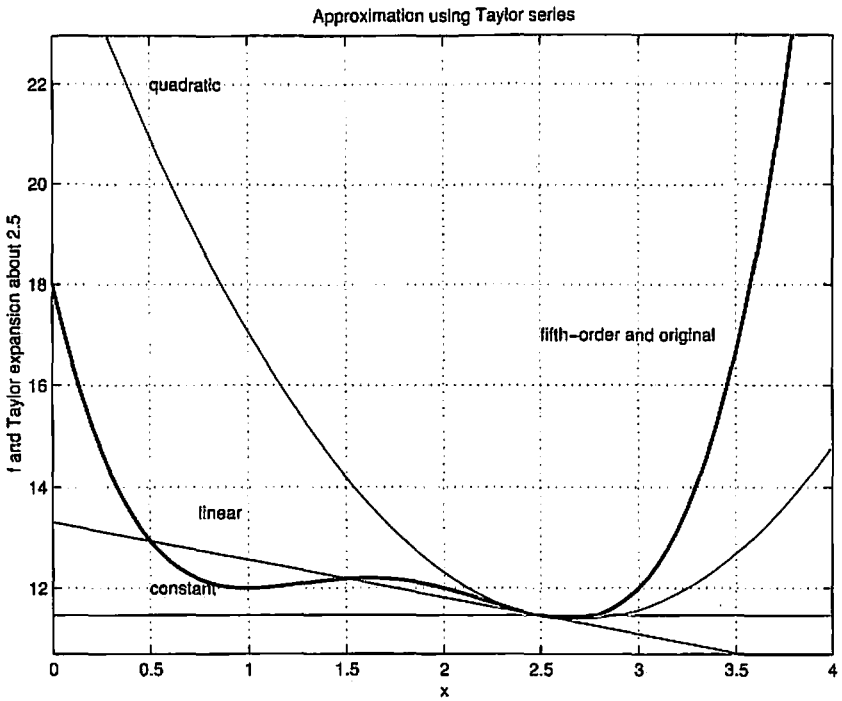

Figure 4.4 Taylor series approximation.

MATLAB Code: Figure 4.4 is completely created using the code **Fig4_4.m**. It uses the MATLAB-provided symbolic "taylor" function to generate the various expansions about point 2.5.

Two or More Variables: The series are only expanded to the quadratic terms. The truncation error is ignored. The two-variable function expansion is shown in detail and also organized in terms of vectors and matrices. The first-order expansion is expressed in terms of the gradient. The second-order expansion will be expressed in terms of the Hessian matrix

$$f(x_p + \Delta x, y_p + \Delta y) = f(x_p, y_p) + \left[\left.\frac{\partial f}{\partial x}\right|_{(x_p,y_p)} \Delta x + \left.\frac{\partial f}{\partial y}\right|_{(x_p,y_p)} \Delta y \right] +$$

$$\frac{1}{2}\left[\left.\frac{\partial^2 f}{\partial x^2}\right|_{(x_p,y_p)} (\Delta x)^2 + 2 \left.\frac{\partial^2 f}{\partial x \partial y}\right|_{(x_p,y_p)} \Delta x \Delta y + \left.\frac{\partial^2 f}{\partial y^2}\right|_{(x_p,y_p)} (\Delta y)^2 \right] \tag{4.33}$$

If the displacements are organized as a column vector $[\Delta x\ \Delta y]^T$, the expansion in (4.33) can be expressed in a condensed manner as

$$f(x_p + \Delta x, y_p + \Delta y) = f(x_p, y_p) + \nabla f^T_{(x_p,y_p)} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} +$$

$$\frac{1}{2} [\Delta x\ \Delta y]^T [\mathbf{H}(x_p,y_p)] \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \tag{4.34}$$

For n variables, with $\boldsymbol{X}_p$ the current point and $\Delta \boldsymbol{X}$ the displacement vector,

$$f(\mathbf{X}_p + \Delta \mathbf{X}) = f(\mathbf{X}_p) + \nabla f(\mathbf{X}_p)^T \Delta \mathbf{X} + \frac{1}{2} \Delta \mathbf{X}^T \mathbf{H}(\mathbf{X}_p) \Delta \mathbf{X} \tag{4.35}$$

4.3 GRAPHICAL SOLUTIONS

The graphical solutions are presented for three kinds of problems: unconstrained, equality constrained, and inequality constrained, based on the functions in Example 4.1.

4.3.1 Unconstrained Problem

$$\text{Minimize} \quad f(x_1,x_2): -x_1 x_2 \tag{4.36}$$

$$0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \tag{4.37}$$

The side constraints serve to limit the design space. Considering the objective function in (4.36), it is clear that the minimum value of f will be realized if the variables are at the maximum ($x_1^* = 3, x_2^* = 3$). The value of the function is -9. Figure 4.5 illustrates the problem. The solution is at the boundary of the design space. **Fig4_5.m** is the MATLAB code that will produce Figure 4.5. The maximum value of the function is 0 (based on the side constraints). In Figure 4.5 the tangent and the gradient to the objective function at the solution are drawn in using MATLAB *plotedit* commands. In this particular example, the side constraints are necessary to determine the solution. If the design space were increased the solution would correspondingly change.

The example chosen to illustrate the unconstrained problem defined by (4.36) and (4.37) is not usually employed to develop the optimality conditions for general unconstrained problems. Two requirements are expected to be satisfied by such problems: (1) the solution must be in the interior of the design space and (2) there must be a unique solution, or the problem is *unimodal*. Nevertheless, in practical applications these conditions may not exist. Most software are developed to apply the optimality conditions and they rarely verify these requirements are being met. It is usually the designer's responsibility to ensure these requirements are met. To develop the optimality conditions an alternate unconstrained problem is presented:

$$\text{Minimize} \quad f(x_1,x_2): (x_1 - 1)^2 + (x_2 - 1)^2 - x_1 x_2 \tag{4.38}$$

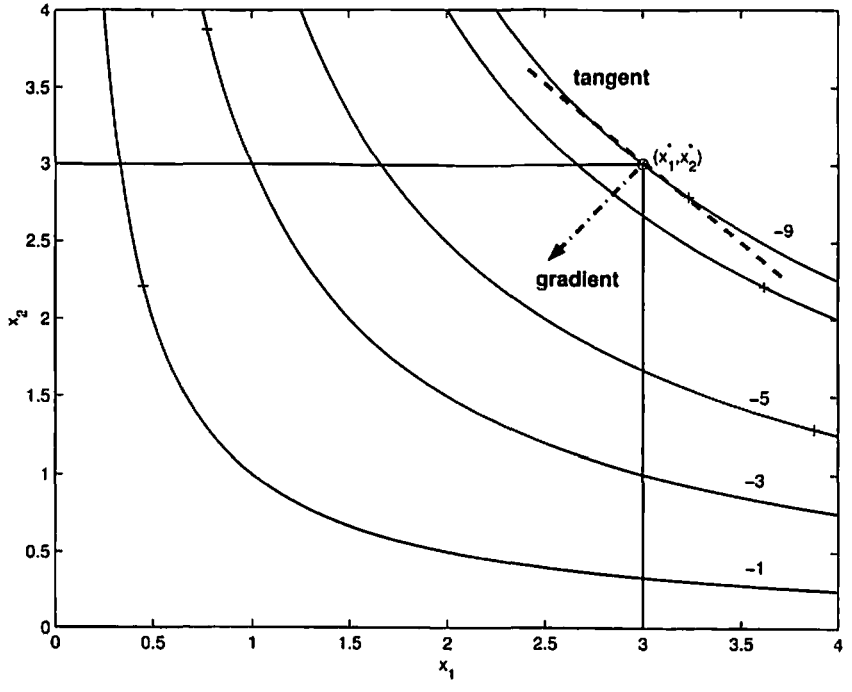

Figure 4.5 Unconstrained solution.

$$0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \tag{4.37}$$

Figure 4.6 displays the problem and its solution. **Fig4_6.m** provides the code that will generate most of Figure 4.6. It is clear that the solution appears to be at $x_1^* = 2$ and $x_2^* = 2$. The optimal value of the objective function is -2. In the contour plot, the optimum is a point making it difficult to draw the gradient. If this is the case, then in three dimensions, at the optimum, the gradient will lie in a plane tangent to the function surface. Moving in this plane should not change the value of the objective function. This observation is used to develop the necessary conditions later.

4.3.2 Equality Constrained Problem

For a two-variable problem we can only utilize one constraint for a meaningful optimization problem:

$$\text{Minimize} \quad f(x_1,x_2): -x_1 x_2 \tag{4.36}$$

$$\text{Subject to:} \quad h_1(x_1,x_2): x_1^2/4 + x_2^2 - 1 \tag{4.39}$$

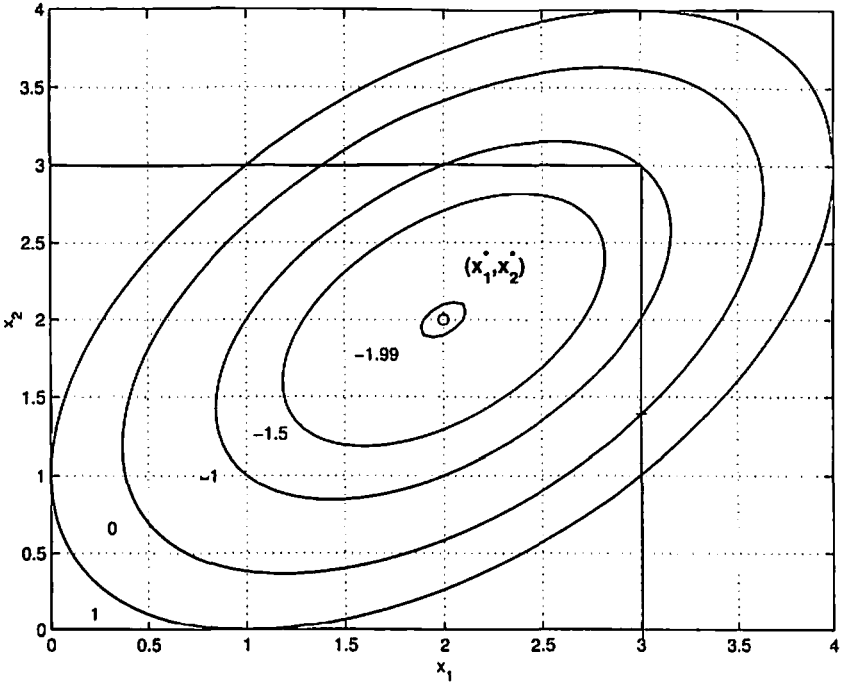

Figure 4.6 Unconstrained problem: interior solution.

$$0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \tag{4.37}$$

Figure 4.7 (through **Fig4_7.m**) illustrates the problem. The dashed line is the constraint. Since it is an equality constraint, the solution must be a point on the dashed line. The contour of $f = -1$ appears to just graze the constraint and therefore is the minimum possible value of the function without violating the constraint. In Figure 4.7, the gradient of the objective function as well as the gradient of the constraints at the solution are illustrated. It appears that at this point these gradients are parallel even though they are directed opposite to each other. By definition the gradient is in the direction of the most rapid increase of the function at the selected point. This is not a coincidence. This fact is used to establish the necessary conditions for the problem.

4.3.3 Inequality Constrained Problem

The number of inequality constraints is not dependent on the number of variables. For illustration, both constraint functions of Example 4.1 are formulated as inequality constraints.

$$\text{Minimize} \quad f(x_1, x_2): -x_1 x_2 \tag{4.36}$$

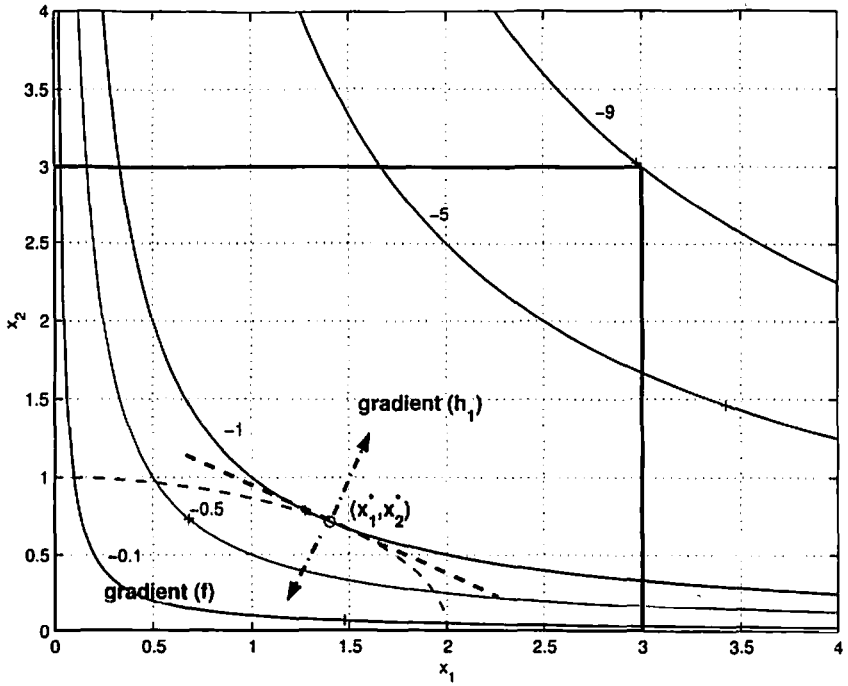

Figure 4.7 Equality constrained problem.

$$\text{Subject to:} \quad g_1(x_1,x_2)\colon 20\,x_1 + 15\,x_2 - 30 \le 0 \tag{4.40}$$

$$g_2(x_1,x_2)\colon x_1^2/4 + x_2^2 - 1 \le 0 \tag{4.41}$$

$$0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \tag{4.37}$$

Figure 4.8 illustrates the graphical solution. The solution is at the intersection of the dotted lines. The code is available in **Fig4_8.m**. The gradients are drawn using the *plotedit* functions on the menubar in the window. Also font style and font size have been adjusted using the *plotedit* commands. The optimal solution must lie on or to the left of the dashed line. Similarly it must lie below or on the dashed curve. Simultaneously it should also decrease the objective function value as much as possible.

4.3.4 Equality and Inequality Constraints

Example 4.1 is developed with both types on constraints present. It is reproduced here:

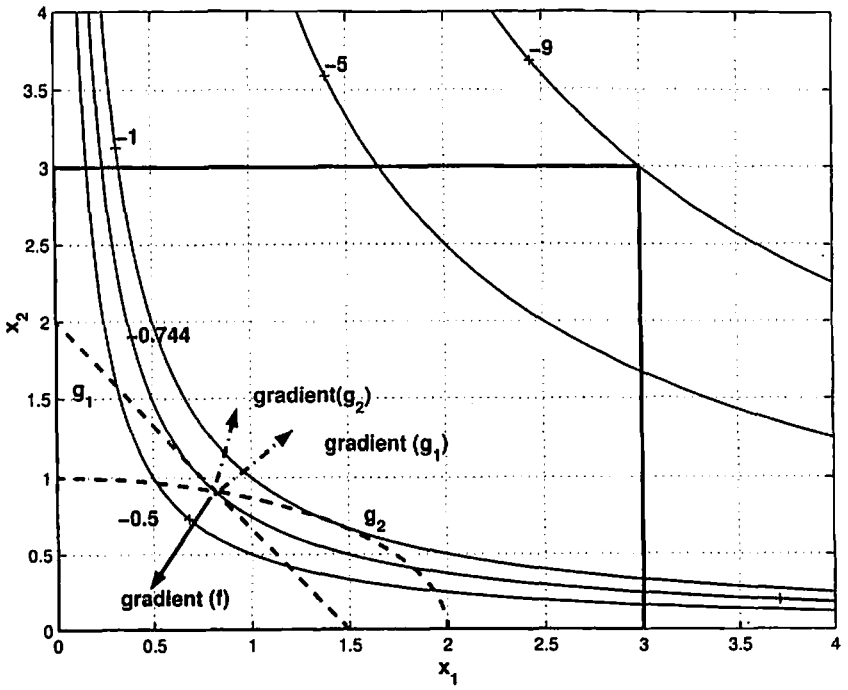

Figure 4.8 Inequality constrained problem.

$$\text{Minimize} \quad f(x_1,x_2)\colon -x_1\,x_2 \tag{4.9}$$

$$\text{Subject to:} \quad h_1(x_1,x_2)\colon 20\,x_1 + 15\,x_2 - 30 = 0 \tag{4.10}$$

$$g_1(x_1,x_2)\colon (x_1^2/4) + (x_2^2) - 1 \le 0 \tag{4.11}$$

$$0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \tag{4.12}$$

The graphical solution of Section 4.3.3 suggests that the inequality constraint will be active at the solution. If that is the case, then Figure 4.8 will once again represent the graphical solution to this section also. This will be verified in the next section.

4.4 ANALYTICAL CONDITIONS

Analytical conditions refer to the necessary and sufficient conditions that will permit the recognition of the solution to the optimal design problem. The conditions developed here empower the numerical techniques to follow later. They are

introduced in the same sequence as in the previous section. Instead of formal mathematical details, the conditions are established less formally from the geometrical description of the problem and/or through intuitive reasoning. Formal development of the analytical conditions can be found in References 6–8. For establishing the conditions it is expected that the solution is in the interior of the feasible region, and there is only one minimum.

4.4.1 Unconstrained Problem

The problem used for illustration is

$$\text{Minimize} \quad f(x_1, x_2): (x_1 - 1)^2 + (x_2 - 1)^2 - x_1 x_2 \tag{4.38}$$

$$0 \le x_1 \le 3; \quad 0 \le x_2 \le 3 \tag{4.37}$$

Figure 4.6 provided a contour plot of the problem. Figure 4.9 provides a three-dimensional plot of the same problem. It is a mesh plot. The commands to create Figure 4.9 are available in **Fig4_9.m**. The figure needs to be rotated interactively to appear as illustrated. A tangent plane is drawn at the minimum for emphasis. Figure 4.9 will be used to identify the properties of the function $f(x_1,x_2)$ at the minimum.

The minimum is identified by a superscript asterisk ($\mathbf{X}^*$ or $[x_1^*, x_2^*]^T$). Studying Figure 4.9, the function has a minimum value of -2, while $x_1^* = 2$ and $x_2^* = 2$. If the values of x_1 and/or x_2 were to change even by a slight amount from the optimal value, in any direction, the value of the function will certainly increase since $\mathbf{X}^*$ is the lowest point of the concave surface representing the function f. Representing the displacement from the optimum values of the variables as $\Delta\mathbf{X}$, and the change in the function value from the optimum as Δf, from direct observation it is clear that the optimal solution must be a point that satisfies

$$\Delta f > 0, \qquad \text{for all } \Delta\mathbf{X}$$

First-Order Conditions: The same idea can be applied in the limit, that is, for infinitesimal displacement dx_1 and dx_2 about $\mathbf{X}^*$. The function itself can be approximated by a plane tangent to the function at the solution (shown in Figure 4.9) (consider this as a first-order Taylor series expansion). Moving to any point in the plane from the optimum (see Figure 4.9) will not change the value of the function, therefore $df = 0$. Moving away from optimum implies that dx_1 and dx_2 *are not zero.* Rewriting Equation (4.20) in terms of x_1 and x_2

$$df = \frac{\partial f}{\partial x_1} dx_1 + \frac{\partial f}{\partial x_2} dx_2 = 0$$

$$df = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{bmatrix} \begin{bmatrix} dx_1 \\ dx_2 \end{bmatrix} = 0 \tag{4.42}$$

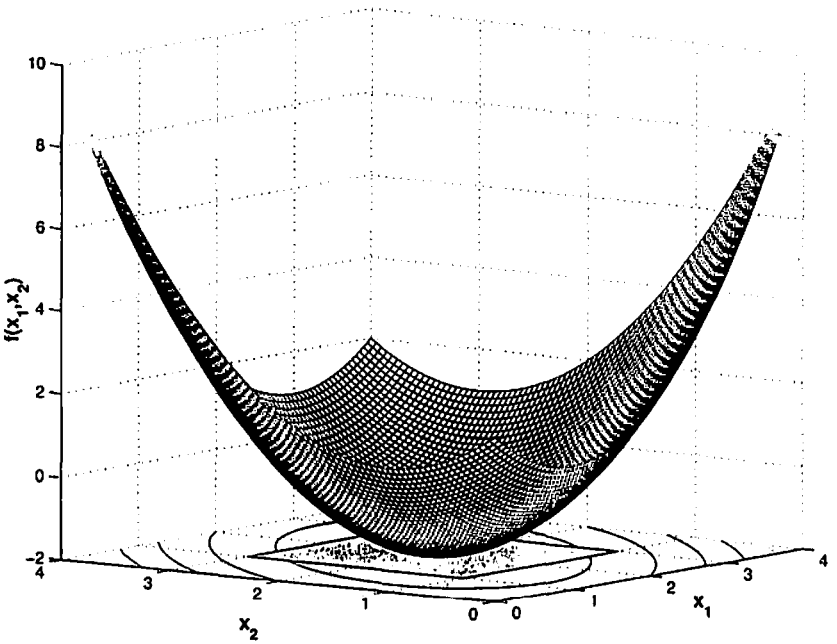

Figure 4.9 Three-dimensional plot of Figure 4.6.

Since this should hold for all points in the plane, $dx_1 \neq 0$ and $dx_2 \neq 0$. Therefore,

$$\frac{\partial f}{\partial x_1} = 0, \qquad \frac{\partial f}{\partial x_2} = 0$$

or the *gradient of f at the optimum must be zero.* That is,

$$\nabla f(x_1^*, x_2^*) = 0 \tag{4.43}$$

Equation (4.43) expresses the necessary condition, or *first-order conditions* (FOC), for *unconstrained optimization.* It is termed the first-order condition because Equation (4.43) uses the gradient or the first derivative. Equation (4.43) is used to identify the possible solutions to the optimization problem. If the function were to flip over so that the same function were to maximize the value of the function, the solution for the variables will be at the same value of the design variable. It is clear that Equation (4.43) applies to the maximum problem also. Equation (4.43) by itself will not determine the minimum value of the function. Additional considerations are necessary to ensure that the solution established by the first-order conditions is optimal, in this case a minimum. For a general unconstrained problem, the necessary conditions of Equation (4.43) can be stated as

$$\nabla f(\mathbf{X}^*) = 0 \tag{4.44}$$

which is primarily a vector expression of the relation in Equation (4.43). Equation (4.44) is used to establish the value of the design variables $\mathbf{X}^*$ both analytically and numerically.

Second-Order Conditions: The second-order conditions (SOC) are usually regarded as *sufficient conditions*. It can be inferred that these conditions will involve *second derivatives* of the function. The SOC is often obtained through the Taylor expansion of the function to second order. If $\mathbf{X}^*$ is the solution, and $\Delta\mathbf{X}$ represents the change of the variables from the optimal value which will yield a change Δf, then

$$\Delta f = f(\mathbf{X}^* + \Delta\mathbf{X}) - f(\mathbf{X}^*) = \nabla f(\mathbf{X}^*)^T \Delta\mathbf{X} + \frac{1}{2}\Delta\mathbf{X}^T\mathbf{H}(\mathbf{X}^*)\Delta\mathbf{X} \tag{4.45}$$

This is similar to Equation (4.35) except the expansion is about the solution. Δf must be greater than zero. Employing the necessary conditions (4.44), the first term on the right-hand side of Equation 4.45 is zero. This leaves the following inequality

$$\Delta f = \frac{1}{2}\Delta\mathbf{X}^T\mathbf{H}(\mathbf{X}^*)\Delta\mathbf{X} > 0 \tag{4.46}$$

where $\mathbf{H}(\mathbf{X}^*)$ is the Hessian matrix (the matrix of second derivatives) of the function f at the possible optimum value $\mathbf{X}^*$. For the relations in Equation (4.46) to hold, the matrix $\mathbf{H}(\mathbf{X}^*)$ must be positive definite. There are three ways to establish the H is positive definite.

(i) For all possible $\Delta\mathbf{X}$, $\Delta\mathbf{X}^T\mathbf{H}(\mathbf{X}^*)\Delta\mathbf{X} > 0$.

(ii) The eigenvalues of $\mathbf{H}(\mathbf{X}^*)$ are all positive.

(iii) The determinants of all lower orders (submatrices) of $\mathbf{H}(\mathbf{X}^*)$ that include the main diagonal are all positive.

Of the three, only (ii) and (iii) can be practically applied, which is illustrated below.

Example

$$\text{Minimize} \quad f(x_1,x_2): (x_1 - 1)^2 + (x_2 - 1)^2 - x_1 x_2 \tag{4.38}$$

$$0 \le x_1 \le 3; \quad 0 \le x_2 \le 3 \tag{4.37}$$

FOC

$$\frac{\partial f}{\partial x_1} = 2(x_1 - 2) - x_2 = 0 \tag{4.47a}$$

$$\frac{\partial f}{\partial x_2} = 2(x_2 - 1) - x_1 = 0 \tag{4.47b}$$

Equations (4.47a) and (4.47b) represent a pair of linear equations which can be solved as $x_1^* = 2, x_2^* = 2$. The value of the function f^* is -2. So far only the necessary conditions have been satisfied. The values above can also refer to a point where the function is a maximum, or where there is a saddle point.

SOC: For this problem [see Equation (4.26)]

$$\mathbf{H} = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$$

Is it positive definite?

(i) Not possible to test all $\Delta\mathbf{X}$

(ii) To calculate the eigenvalues of $\mathbf{H}$,

$$\begin{vmatrix} 2-\lambda & -1 \\ -1 & 2-\lambda \end{vmatrix} = (2-\lambda)(2-\lambda) - 1 = 3 - 4\lambda + \lambda^2 = (\lambda - 3)(\lambda - 1) = 0$$

The eigenvalues are $\lambda = 1$, $\lambda = 3$ and the matrix is positive definite.

(iii) To calculate determinants of all orders that include the main diagonal and include the element in the first row and first column,

$$|2| > 0$$

$$\begin{vmatrix} 2 & -1 \\ -1 & 2 \end{vmatrix} = 4 - 1 = 3 > 0$$

The matrix is positive definite.

Sec_4_4-1.m provides the confirmation of the numerical values for this example using Symbolic Toolbox and basic MATLAB commands.

4.4.2 Equality Constrained Problem

The problem:

$$\text{Minimize} \quad f(x_1,x_2): -x_1 x_2 \tag{4.36}$$

$$\text{Subject to:} \quad h_1(x_1,x_2): x_1^2/4 + x_2^2 - 1 \tag{4.39}$$

$$0 \le x_1 \le 3; \quad 0 \le x_2 \le 3 \tag{4.37}$$

Figure 4.7 illustrated the graphical solution. It was noticed that *at the solution*, the gradient of the objective function and the gradient of the constraint were parallel and oppositely directed. Examining other feasible points in Figure 4.7 (on the dashed curve) it can be ascertained that the special geometrical relationship is only possible at the solution. At the solution a proportional relationship exists between the gradients at the solution. Using the constant of proportionality λ_1 (a positive value) the relationship between the gradients can be expressed as

$$\nabla f = -\lambda_1 \nabla h_1 \quad \text{or} \quad \nabla f + \lambda_1 \nabla h_1 = 0 \tag{4.48}$$

Equation (4.48) is usually obtained in a more formal way using the method of Lagrange multipliers.

Method of Lagrange: In this method, the problem is transformed by introducing an augmented function, called the *Lagrangian*, as the objective function subject to the same equality constraints. The Lagrangian is defined as the sum of the original objective function and a *linear combination* of the constraints. The coefficients of this linear combination are known as the *Lagrange multipliers*. With reference to the example in this section

$$\text{Minimize} \quad F(x_1, x_2, \lambda_1) = f(x_1, x_2) + \lambda_1 h_1(x_1, x_2) \tag{4.49}$$

The complete problem is developed as

$$\text{Minimize} \quad F(x_1, x_2, \lambda_1) = -x_1 x_2 + \lambda_1 (x_1^2/4 + x_2^2 - 1) \tag{4.50}$$

$$\text{Subject to:} \quad h_1(x_1, x_2): x_1^2/4 + x_2^2 - 1 \tag{4.39}$$

$$0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \tag{4.37}$$

Is the solution to the transformed problem [Equations (4.50), (4.39), (4.37)] the same as the solution to the original problem [Equations (4.36), (4.39), (4.37)]? If the design is feasible, then most definitely yes. For feasible designs, $h_1(x_1, x_2) = 0$, and the objective functions in Equations (4.36) and (4.50) are the same. If design is not feasible, then by definition there is no solution anyway.

The FOC are obtained by considering $F(x_1, x_2, \lambda_1)$ as an *unconstrained function* in the variables x_1, x_2, λ_1. This provides three relations to solve for x_1^*, x_2^*, λ_1^*:

$$\frac{\partial F}{\partial x_1} = \frac{\partial f}{\partial x_1} + \lambda_1 \frac{\partial h_1}{\partial x_1} = 0$$

$$\frac{\partial F}{\partial x_2} = \frac{\partial f}{\partial x_2} + \lambda_1 \frac{\partial h_1}{\partial x_2} = 0 \tag{4.51}$$

$$\frac{\partial F}{\partial \lambda_1} = h_1 = 0$$

Equations (4.51) express the FOC or necessary conditions for an equality constrained problem in two variables. The last equation in the above set is the constraint equation. This ensures the solution is feasible. The first two equations can be assembled in vector form to yield the same information expressed by Equation (4.48), which was obtained graphically. The left-hand expressions in the first two equations above are the gradient of the Lagrangian function:

$$\nabla F = \nabla f + \lambda_1 \nabla h_1 = 0$$

Applying Equations (4.51) to the example of this section:

$$\frac{\partial F}{\partial x_1} = -x_1 + \frac{\lambda_1 x_1}{2} = 0$$

$$\frac{\partial F}{\partial x_2} = -x_2 + 2\lambda_1 x_2 = 0 \tag{4.52}$$

$$h_1 = \frac{x_1^2}{4} + x_2^2 - 1 = 0$$

Equations (4.52) represent three equations in three variables which should determine the values for $x_1^*, x_2^*, \lambda_1^*$. Note that Equations (4.52) only define the necessary conditions, which means the solution could be a maximum also.

There is one problem with respect to the set of Equations (4.52). The equations are a nonlinear set. Most prerequisite courses on numerical methods do not attempt to solve a nonlinear system of equations. Usually, they only handle a single one through Newton–Raphson or the bisection method. In fact, NLP or design optimization is primarily about techniques for solving a system of nonlinear equations, albeit of specific forms. This means Equations (4.52) cannot be solved until we advance further in this course. Fortunately, having a tool like MATLAB obviates this difficulty, and provides a strong justification of using a numerical/symbolic tool for supporting the development of the course.[2]

MATLAB Code: In MATLAB, there are two ways of solving Equations (4.52), using symbolic support functions or using the numerical support functions. Both procedures have limitations when applied to highly nonlinear functions. The symbolic function is *solve* and the numerical function is *fsolve*. The numerical technique is an iterative one and requires you to choose an initial guess to start the procedure. Quite often several different guesses may be required to find the solution. Solutions of systems of equations are fundamental to the rest of the book. The following code is included as a hands-on exercise. It is available as **Sec4_4_2.m**. It requires **eqns4_4_2.m** to execute *fsolve* command. The contents of the files are listed below

[2]There are other tools besides MATLAB such as Mathcad and Excel.

Sec4_4_2.m

```
% Necessary/Sufficient conditions for
% Equality constrained problem
%
% Optimization with MATLAB, Section 4.4.2
%  Dr. P.Venkataraman
%
%  Minimize f(x1,x2) = -x1
%
%-------------------------
% symbolic procedure
%-------------------------
% define symbolic variables
format compact
syms x1 x2 lam1 h1 F
% define F
F = -x1*x2 + lam1*(x1*x1/4 + x2*x2 -1);
h1 = x1*x1/4 +x2*x2 -1;

%the gradient of F
grad1 = diff(F,x1);
grad2 = diff(F,x2);

% optimal values
% satisfaction of necessary conditions
[lams1 xs1 xs2] = solve(grad1,grad2,h1,'x1,x2,lam1');

% the solution is returned as a vector of
% the three unknowns in case of multiple solutions
% lams1  is the solution vector for lam1 etc.
% IMPORTANT: the results are sorted alphabetically
%
% fprintf is used to print a string in the
% command window
% disp is used to print values of matrix

f = -xs1.*xs2;
fprintf('The solution (x1*,x2*,lam1*, f*):\n'), ...
  disp(double([xs1 xs2 lams1 f]))
%-----------------------------
% Numerical procedure
%-----------------------------
% solution to non-linear system using fsolve
% see help fsolve
%
% the unknowns have to be defined as a vector
% the functions have to be set up in an m-file
```



```
% define initial values
xinit=[1 1 0.5]'; % initial guess for x1, x2, lam1

% the equations to be solved are available in
% eqns4_4_2.m

xfinal = fsolve('eqns4_4_2',xinit);

fprintf('The numerical solution (x1*,x2*,lam1*): \n'), ...
  disp(xfinal);
```

eqns4_4_2.m

```
function ret = eqns4_4_2(x)
% x is a vector
% x(1) = x1, x(2) = x2, x(3) = lam1
ret=[(-x(2) + 0.5*x(1)*x(3)), ...
      (-x(1) + 2*x(2)*x(3)), ...
      (0.25*x(1)*x(1) + x(2)*x(2) -1)];
```

Output in MATLAB Command Window

```
The solution (x1*,x2*,lam1*, f*):
 1.4142    0.7071     1.0000  -1.0000
-1.4142   -0.7071     1.0000  -1.0000
-1.4142    0.7071    -1.0000   1.0000
 1.4142   -0.7071    -1.0000   1.0000
Optimization terminated successfully:
Relative function value changing by less than
  OPTIONS.TolFun
  The numerical solution (x1*,x2*,lam1*):
   1.4141
   0.7071
   0.9997
```

The symbolic computation generates four solutions. Only the first one is valid for this problem. This is decided by the side constraints expressed by Equation (4.37). This is an illustration of the a priori manner by which the side constraints affect the problem. On the other hand, the numerical techniques provide only one solution to the problem. This is a function of the initial guess. Generally, numerical techniques will deliver solutions closest to the point they started from. The solution is

$$x_1^* = 1.4141;\ x_2^* = 0.7071;\ \lambda_1^* = 1.0$$

The solutions for x_1 and x_2 can be verified from the graphical solution in Figure 4.7. The solutions can also be verified through the hand-held calculator. The value and sign for λ are usually immaterial for establishing the optimum. For a well-posed problem it should be positive since increasing the constraint value is useful only if we are trying to identify a lower value for the minimum. This forces h and f to move in opposite

directions (gradients). Increasing constraint value can be associated with enlarging the feasible domain which may yield a better design.

Lagrange Multipliers: The Lagrange multiplier method is an elegant formulation to obtain the solution to a constrained problem. In overview, it seems strange that we have to introduce an additional unknown (λ_1) to solve the constrained problem. This violates the conventional rule for NLP that the fewer the variables, the better the chances of obtaining the solution. As indicated in the discussion earlier, the Lagrangian allows the transformation of a constrained problem into an unconstrained problem. The Lagrange multiplier also has a physical significance. At the solution it expresses the ratio of the change in the objective function to the change in the constraint value. To illustrate this consider:

$$F = f + \lambda_1 h_1$$

$$dF = df + \lambda_1 dh$$

$$dF = \frac{\partial F}{\partial x_1} dx_1 + \frac{\partial F}{\partial x_2} dx_2 + \frac{\partial F}{\partial \lambda_1} d\lambda_1$$

At the solution, the FOC deems that $dF = 0$ (which can also be seen in the above detailed expansion). Hence,

$$\lambda_1 = -\frac{df}{dh} = -\frac{\Delta f}{\Delta h}$$

The above dependence does not affect the establishment of the optimal design. It does have an important role in the discussion of *design sensitivity* in NLP problems.

General Equality Constrained Problem: Remembering $n - l > 0$,

$$\text{Minimize} \quad f(\mathbf{X}), [\mathbf{X}]_n \tag{4.5}$$

$$\text{Subject to:} \quad [\mathbf{h}(\mathbf{X})]_l = 0 \tag{4.6}$$

$$\mathbf{X}^{\text{low}} \le \mathbf{X} \le \mathbf{X}^{\text{up}} \tag{4.8}$$

The augmented problem with the Lagrangian:

$$\begin{aligned} \text{Minimize} \quad F(\mathbf{X},\lambda) &= f(\mathbf{X}) + \sum_{k=1}^{l} \lambda_k h_k(\mathbf{X}) \\ &= f(\mathbf{X}) + \boldsymbol{\lambda} \cdot \mathbf{h}; \\ &= f(X) + \boldsymbol{\lambda}^{\mathrm{T}} \mathbf{h} \end{aligned} \tag{4.53}$$

Subject to: $[\mathbf{h}(\mathbf{X})]_l = 0$ (4.6)

$\mathbf{X}^{\text{low}} \le \mathbf{X} \le \mathbf{X}^{\text{up}}$ (4.8)

In Equation (4.53), three equivalent representations for the Lagrangian are shown. The FOC are

$$\frac{\partial F}{\partial x_i} = \frac{\partial f}{\partial x_i} + \sum_{k=1}^{l} \lambda_k \frac{\partial h_k}{\partial x_i} = 0; \qquad i = 1, 2, \ldots, n \tag{4.54}$$

Subject to: $[\mathbf{h}(\mathbf{X})]_l = 0$ (4.6)

$\mathbf{X}^{\text{low}} \le \mathbf{X} \le \mathbf{X}^{\text{up}}$ (4.8)

Equations (4.54) and (4.6) provide the $n + l$ relations to determine the $n + l$ unknowns $\mathbf{X}^*$, λ^*. Equation (4.8) is used after the solution is obtained, if necessary. Equation (4.54) is also expressed as

$$\nabla \mathbf{F} = \nabla f + [\nabla h_1 \; \nabla h_2 \cdots \nabla h_l] \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \cdot \\ \lambda_l \end{bmatrix} = 0 \tag{4.55}$$

Second-Order Conditions: At the solution determined by the FOC, the function should increase for changes in $\Delta\mathbf{X}$. Changes in $\Delta\mathbf{X}$ are not arbitrary. They have to satisfy the linearized equality constraint at the solution. It is taken for granted that the sufficient conditions are usually applied in a small neighborhood of the optimum. Also, changes are contemplated only with respect to X and not with respect to the Lagrange multiplier. The Lagrange method is often called the *method of undetermined coefficient*—indicating it is not a variable. In the analytical derivation of the FOC for this problem the Lagrangian F was considered unconstrained. Borrowing from the previous section (unconstrained minimization), the SOC can be expected to satisfy the following relations:

$$\Delta \mathbf{F} = F(\mathbf{X}^* + \Delta\mathbf{X}) - F(\mathbf{X}^*) = \nabla \mathbf{F}(\mathbf{X}^*) + \frac{1}{2} \Delta\mathbf{X}^{\mathrm{T}} [\nabla^2 \mathbf{F}(\mathbf{X}^*)] \Delta\mathbf{X} > 0 \tag{4.56}$$

$$\nabla \mathbf{h}^{\mathrm{T}} \Delta\mathbf{X} = 0$$

In the above $[\nabla^2 \mathbf{F}(\mathbf{X}^*)]$ is the Hessian of the Lagrangian, with respect to the design variables only evaluated at the solution. Also, the FOC require that $\nabla \mathbf{F}(\mathbf{X}^*) = 0$. With reference to two variables and one constraint:

$$\Delta F = \frac{1}{2}\left\{\frac{\partial^2 F}{\partial x_1^2}(\Delta x_1)^2 + 2\frac{\partial^2 F}{\partial x_1 \partial x_2}(\Delta x_1)(\Delta x_2) + \frac{\partial^2 F}{\partial x_2^2}(\Delta x_2)^2\right\} > 0$$

$$\Delta F = \frac{1}{2}\left\{\frac{\partial^2 F}{\partial x_1^2}\left(\frac{\Delta x_1}{\Delta x_2}\right)^2 + 2\frac{\partial^2 F}{\partial x_1 \partial x_2}\left(\frac{\Delta x_1}{\Delta x_2}\right) + \frac{\partial^2 F}{\partial x_2^2}\right\}(\Delta x_2)^2 > 0 \tag{4.57}$$

$$\frac{\Delta x_1}{\Delta x_2} = -\frac{\partial h_1/\partial x_2}{\partial h_1/\partial x_1} \tag{4.58}$$

Substitute Equation (4.58) in Equation (4.57) and the SOC requires that the expression in braces must be positive. The derivatives in Equations (4.57) and (4.58) are evaluated at the minimum.

Applying the second-order condition to the example of this section is left as an exercise for the reader. Compared to the SOC for the unconstrained minimization problem, Equations (4.57) and (4.58) are not easy to apply, especially the substitution of Equation (4.58). From a practical perspective the SOC is not imposed for equality constrained problems. It is left to the designer to ensure by other means that the solution is a minimum solution.

4.4.3 Inequality Constrained Optimization

The problem

$$\text{Minimize} \quad f(x_1, x_2): -x_1 x_2 \tag{4.36}$$

$$\text{Subject to:} \quad g_1(x_1,x_2): 20x_1 + 15x_2 - 30 \le 0 \tag{4.40}$$

$$g_2(x_1,x_2): x_1^2/4 + x_2^2 - 1 \le 0 \tag{4.41}$$

$$0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \tag{4.37}$$

The number of variables ($n = 2$) and the number of inequality constraints ($m = 2$) do not depend on each other. In fact, an inequality constrained problem in a single variable can be usefully designed and solved. Section 4.4.2 solved the equality constrained problem. If the above problem can be transformed to an equivalent equality constrained problem, then we have found the solution. The standard transformation requires a slack variable z_j for each inequality constraint g_j. Unlike LP problems, the slack variable for NLP is not restricted in sign. Therefore, the square of the new variable is added to the left-hand side of the corresponding constraint. This adds a positive value to the left-hand side to bring the constraint up to zero. Of course a zero value will be added if the constraint is already zero.

Transformation to an Equality Constrained Problem

$$\text{Minimize} \quad f(x_1, x_2): -x_1 x_2 \tag{4.36}$$

$$\text{Subject to:} \quad g_1(x_1,x_2) + z_1^2: 20x_1 + 15x_2 - 30 + z_1^2 = 0 \tag{4.59}$$

$$g_2(x_1,x_2) + z_2^2: x_1^2/4 + x_2^2 - 1 + z_2^2 = 0 \tag{4.60}$$

$$0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3 \tag{4.37}$$

There are four variables (x_1, x_2, z_1, z_2) and two equality constraints. It is a valid equality constrained problem. The Lagrange multiplier method can be applied to this transformation. To distinguish the multipliers associated with inequality constraints the symbol β is used. This is strictly for clarity.

Method of Lagrange: The augmented function or the Lagrangian is:

$$\text{Minimize} \quad F(x_1,x_2,z_1,z_2,\beta_1,\beta_2) = f(x_1,x_2) + \beta_1[g_1(x_1,x_2) + z_1^2] + \beta_2[g_2(x_1,x_2) + z_2^2] \tag{4.61}$$

If the Lagrangian is considered as an unconstrained objective function, the FOC (necessary conditions) are

$$\frac{\partial F}{\partial x_1} = \frac{\partial f}{\partial x_1} + \beta_1\frac{\partial g_1}{\partial x_1} + \beta_2\frac{\partial g_2}{\partial x_1} = 0 \tag{4.62a}$$

$$\frac{\partial F}{\partial x_2} = \frac{\partial f}{\partial x_2} + \beta_1\frac{\partial g_1}{\partial x_2} + \beta_2\frac{\partial g_2}{\partial x_2} = 0 \tag{4.62b}$$

$$\frac{\partial F}{\partial z_1} = 2\beta_1 z_1 = 0 \tag{4.62c}$$

$$\frac{\partial F}{\partial z_2} = 2\beta_2 z_2 = 0 \tag{4.62d}$$

$$\frac{\partial F}{\partial \beta_1} = g_1 + z_1^2 = 0 \tag{4.62e}$$

$$\frac{\partial F}{\partial \beta_2} = g_2 + z_2^2 = 0 \tag{4.62f}$$

Equations (4.62e) and (4.62f) are equality constraints. Equation set (4.62) provides six equations to solve for x_1^*, x_2^*, z_1^*, z_2^*, β_1^*, β_2^*. By simple recombination, Equations (4.62c) to (4.62f) can be collapsed to two equations, while the slack variables z_1 and z_2 *can be eliminated from the problem.*

First multiply Equation (4.62c) by z_1. Replace z_1^2 by $-g_1$ from Equation (4.62e). Drop the negative sign as well as the coefficient 2 to obtain

$$\beta_1 g_1 = 0 \tag{4.63a}$$

$$\beta_2 g_2 = 0 \tag{4.63b}$$

Equation (4.63b) is obtained by carrying out similar manipulations with Equations (4.62d) and (4.62f). The FOC can be restated as

$$\frac{\partial F}{\partial x_1} = \frac{\partial f}{\partial x_1} + \beta_1 \frac{\partial g_1}{\partial x_1} + \beta_2 \frac{\partial g_2}{\partial x_1} = 0 \tag{4.62a}$$

$$\frac{\partial F}{\partial x_2} = \frac{\partial f}{\partial x_2} + \beta_1 \frac{\partial g_1}{\partial x_2} + \beta_2 \frac{\partial g_2}{\partial x_2} = 0 \tag{4.62b}$$

$$\beta_1 g_1 = 0 \tag{4.63a}$$

$$\beta_2 g_2 = 0 \tag{4.63b}$$

These four equations have to be solved for x_1^*, x_2^*, β_1^*, β_2. Note that z_1^*, z_2 are not being determined—which suggests that they can be discarded from the problem altogether. It would be useful to pretend z's never existed in the first place.

Equations (4.63) lay out a definite feature for a nontrivial solution: either β_i is zero (and $g_i \neq 0$) or g_i is zero ($\beta_i \neq 0$). Since simultaneous equations are being solved, the conditions on the multipliers and constraints must be satisfied simultaneously. For Equations (4.63), this translates into the following four cases. The information on g in brackets is to emphasize an accompanying consequence.

$$\begin{aligned}
&\text{Case a: } \beta_1 = 0\ [g_1 < 0];\ \beta_2 = 0\ [g_2 < 0] \\
&\text{Case b: } \beta_1 = 0\ [g_1 < 0];\ \beta_2 \neq 0\ [g_2 = 0] \\
&\text{Case c: } \beta_1 \neq 0\ [g_1 = 0];\ \beta_2 = 0\ [g_2 < 0] \\
&\text{Case d: } \beta_1 \neq 0\ [g_1 = 0];\ \beta_2 \neq 0\ [g_2 = 0]
\end{aligned} \tag{4.64}$$

In Equation (4.64), if $\beta_i \neq 0$ (or corresponding $g_i = 0$), then the corresponding constraint is an *equality*. In the previous section a simple reasoning was used to show that the sign of the multiplier must be positive (> 0) for a well-formulated problem. While the sign of the multiplier was ignored for the equality constrained problem, it is included as part of the FOC for the inequality constrained problem. Before restating

Equation (4.64), the Lagrangian is reformulated without the slack variables as (we are going to pretend z never existed)

$$\text{Minimize} \quad F(x_1,x_2,\beta_1,\beta_2) = f(x_1,x_2) + \beta_1[g_1(x_1,x_2)] + \beta_2[g_2(x_1,x_2)] \tag{4.65}$$

which is the same formulation in Equation (4.53). The slack variable was introduced to provide the transformation to an equality constraint. It is also evident that the construction of the Lagrangian function is insensitive to the type of constraint. Since the multipliers tied to the inequality constraint are required to be positive, while those corresponding to the equality constraints are not, this book will continue to distinguish between the multipliers. This will serve to enforce clarity of presentation. The FOC for the problem:

$$\frac{\partial F}{\partial x_1} = \frac{\partial f}{\partial x_1} + \beta_1 \frac{\partial g_1}{\partial x_1} + \beta_2 \frac{\partial g_2}{\partial x_1} = 0 \tag{4.62a}$$

$$\frac{\partial F}{\partial x_2} = \frac{\partial f}{\partial x_2} + \beta_1 \frac{\partial g_1}{\partial x_2} + \beta_2 \frac{\partial g_2}{\partial x_2} = 0 \tag{4.62b}$$

$$\begin{aligned}
&\text{Case a: } \beta_1 = 0\ [g_1 < 0];\ \beta_2 = 0\ [g_2 < 0] \\
&\text{Case b: } \beta_1 = 0\ [g_1 < 0];\ \beta_2 > 0\ [g_2 = 0] \\
&\text{Case c: } \beta_1 > 0\ [g_1 = 0];\ \beta_2 = 0\ [g_2 < 0] \\
&\text{Case d: } \beta_1 > 0\ [g_1 = 0];\ \beta_2 > 0\ [g_2 = 0]
\end{aligned} \tag{4.64}$$

Equation set (4.62) and any single case in Equation (4.64) provides four equations to solve for the four unknowns of the problem. All four sets must be solved for the solution. The best design is decided by scanning the several solutions.

The sign of the multiplier in the solution is *not a sufficient* condition for the inequality constrained problem. The value is unimportant for optimization but may be relevant for sensitivity analysis. Generally a positive value of the multiplier indicates that the solution is not a *local maximum*. Formally verifying a minimum solution requires consideration of the second derivative of the Lagrangian. In practical situations, if the problem is well defined, the positive value of the multiplier usually suggests a minimum solution. This is used extensively in the book to identify the optimum solution.

Solution of the Example: The Lagrangian for the problem is defined as

$$F(x_1,x_2,\beta_1,\beta_2) = -x_1x_2 + \beta_1(20x_1 + 15x_2 - 30) + \beta_2(0.25x_1^2 + x_2^2 - 1)$$

The FOC are

$$\frac{\partial F}{\partial x_1} = -x_2 + 20\beta_1 + 0.5\beta_2 x_1 = 0 \tag{4.66a}$$

$$\frac{\partial F}{\partial x_2} = -x_1 + 15\beta_1 + 2\beta_2 x_2 = 0 \tag{4.66b}$$

$$\beta_1(20x_1 + 15x_2 - 30) = 0 \tag{4.66c}$$

$$\beta_2(0.25x_1^2 + x_2^2 - 1) = 0 \tag{4.66d}$$

The side constraints are

$$0 \le x_1 \le 3; \quad 0 \le x_2 \le 3 \tag{4.37}$$

Case a: $\beta_1 = 0; \beta_2 = 0$: The solution is trivial and by inspection of Equations (4.66a) and (4.66b).

$$x_1 = 0; \quad x_2 = 0; \quad f = 0; \quad g_1 = -30; \quad g_2 = -1 \tag{4.67}$$

The inequality constraints are satisfied. The side constraints are satisfied. The values in Equation (4.67) represent a possible solution.

The solutions for the other cases are obtained using MATLAB. The code is available in **Sec4_4_3.m**. For Cases b and c the appropriate multiplier is set to zero and the resulting three equations in three unknowns are solved. For Case d the complete set of four equations in four unknowns is solved. Case d is also solved numerically. The MATLAB code will also run code **Fig4_8.m** which contains the commands to draw the figure. The output from the Command window is patched below.

```
The solution *** Case a ***(x1*,x2*, f*, g1, g2):
  0     0     0    -30     -1
The solution ***Case (b)*** (x1*,x2*, b2* f*, g1, g2):
 1.4142    0.7071    1.0000   -1.0000    8.8909    0
-1.4142   -0.7071    1.0000   -1.0000  -68.8909    0
-1.4142    0.7071   -1.0000    1.0000  -47.6777    0
 1.4142   -0.7071   -1.0000    1.0000  -12.3223    0
The solution ***Case (c)*** (x1*,x2*, b1* f*, g1, g2):
 0.7500    1.0000    0.0500   -0.7500    0    0.1406
The solution ***Case (d)*** (x1*,x2*, b1*  b2* f*, g1, g2):
 1.8150   -0.4200    0.0426   -1.4007   0.7624  0  0
 0.8151    0.9132    0.0439    0.0856  -0.7443  0 -0.0
Maximum number of function evaluations exceeded;
 increase options.MaxFunEvals
Optimizer is stuck at a minimum that is not a root
Try again with a new starting guess
The numerical solution (x1*,x2*,b1*,b2*):
  0.8187    0.9085    0.0435    0.0913
```


The solution for Case a was discussed earlier and is confirmed above.

Case b has four solutions. The first one is unacceptable because constraint g_1 is not satisfied. The second solution is feasible as far as the functional constraints g_1 and g_2 are concerned but they do not satisfy the side constraints. The third and fourth solutions are unacceptable for the same reason. Thus, Case b is not an optimal solution.

Case c is unacceptable because constraint g_2 is in violation.

Case d has two solutions the first of which is not acceptable for several reasons. The second solution satisfies all of the requirements

- It satisfies the constraints. Both constraints are active.
- The multipliers are positive (maybe a sufficient condition).
- It satisfies the side constraints.

The solution is

$$x_1 = 0.8151; \quad x_2 = 0.9132; \quad \beta_1 = 0.0439; \quad \beta_2 = 0.0856;$$
$$f = -0.7443; \quad g_1 = 0; \quad g_2 = 0$$

This is almost confirmed by the numerical solution which appears to have a problem with convergence. It is almost at the solution. It must be noted here that the MATLAB function used to solve the problem is part of the standard package. The functions in the Optimization Toolbox will certainly do better. Nevertheless, the incomplete solution is reported below but will be ignored due to the symbolic solution available in Case d.

$$x_1 = 0.8187; \quad x_2 = 0.9085; \quad \beta_1 = 0.0435; \quad \beta_2 = 0.0913$$

There are now two candidates for the solution: the trivial solution in Case a and the solution in Case d. The solution in Case d is favored as it has a lower objective function value. All of the cases above can be explored with respect to Figure 4.8. It is an excellent facilitator for comprehending the cases and the solution.

4.4.4 General Optimization Problem

The general optimization problem is described in the set of Equations (4.1)–(4.8). The specific problem in this chapter defined in Section 4.3 is

$$\text{Minimize} \quad f(x_1,x_2): -x_1 x_2 \tag{4.9}$$

$$\text{Subject to:} \quad h_1(x_1,x_2): 20x_1 + 15x_2 - 30 = 0 \tag{4.10}$$

$$g_1(x_1,x_2): (x_1^2/4) + (x_2^2) - 1 \le 0 \tag{4.11}$$

$$0 \le x_1 \le 3; \quad 0 \le x_2 \le 3 \tag{4.12}$$

The FOC for this problem is a combination of the conditions in Sections 4.4.2 and 4.4.3. No new concepts are required. The Lagrange multiplier method is again utilized to set up the FOC. In the following development the multipliers are kept distinct for comprehension.

Lagrange Multiplier Method: The problem is transformed by minimizing the Lagrangian

$$\text{Minimize} \quad F(x_1,x_2,\lambda_1,\beta_2) = -x_1x_2 + \lambda_1(20x_1 + 15x_2 - 30) + \beta_1(0.25x_1^2 + x_2^2 - 1)$$

$$\text{Subject to:} \quad h_1(x_1,x_2)\text{: } 20x_1 + 15x_2 - 30 = 0$$

$$0 \le x_1 \le 3; \qquad 0 \le x_2 \le 3$$

The FOC

$$\frac{\partial F}{\partial x_1} = -x_2 + 20\lambda_1 + 0.5\beta_1 x_1 = 0 \tag{4.68a}$$

$$\frac{\partial F}{\partial x_2} = -x_1 + 15\lambda_1 + 2\beta_1 x_2 = 0 \tag{4.68b}$$

$$\frac{\partial F}{\partial \lambda_1} = 20x_1 + 15x_2 - 30 = 0 \tag{4.68c}$$

$$\text{Case a:} \quad \beta_1 = 0; \qquad g_1 < 0 \tag{4.69a}$$

$$\text{Case b:} \quad \beta_1 > 0; \qquad g_1 = 0 \tag{4.69b}$$

Two solutions must be examined. The first, Case a, requires the solution of a system of three equations in three unknowns λ_1, x_1, x_2 using Equations (4.68) and (4.69a). The second is a system of four equations in the four unknowns λ_1, β_1, x_1, x_2 using Equations (4.68) and (4.69b). Solution of this optimization problem is one of the exercises at the end of the chapter.

Kuhn-Tucker Conditions: The FOC associated with the general optimization problem in Equations (4.1)–(4.4) or (4.5)–(4.8) is termed the *Kuhn-Tucker conditions*. These conditions are established in the same manner as in the previous section. The general optimization problem (repeated here for sake of completeness) is

$$\text{Minimize} \quad f(x_1,x_2,\ldots,x_n) \tag{4.1}$$

$$\text{Subject to:} \quad h_k(x_1,x_2,\ldots,x_n) = 0, \; k = 1,2,\ldots,l \tag{4.2}$$

$$g_j(x_1,x_2,\ldots,x_n) \le 0, \; j = 1,2,\ldots,m \tag{4.3}$$

$$x_i^l \le x_i \le x_i^u \; i = 1,2,\ldots,n \tag{4.4}$$

The Lagrangian:

Minimize

$$F(x_1,\ldots,x_n,\lambda_1,\ldots,\lambda_l,\beta_1,\ldots,\beta_m) =$$

$$f(x_1,\ldots,x_n) + \lambda_1 h_1 + \cdots + \lambda_l h_l + \beta_1 g_1 + \cdots + \beta_m g_m \tag{4.70}$$

There are $n + l + m$ unknowns. The same number of equations are required to solve the problem. These are provided by the FOC or the Kuhn-Tucker conditions:

n equations are obtained as

$$\frac{\partial F}{\partial x_i} = \frac{\partial f}{\partial x_i} + \lambda_1\frac{\partial h_1}{\partial x_i} + \cdots + \lambda_l\frac{\partial h_l}{\partial x_i} + \beta_1\frac{\partial g_1}{\partial x_i} + \cdots + \beta_m\frac{\partial g_m}{\partial x_i} = 0; \quad i = 1,2,\ldots,n \tag{4.71}$$

l equations are obtained directly through the equality constraints

$$h_k(x_1,x_2,\ldots,x_n) = 0; \qquad k = 1,2,\ldots,l \tag{4.72}$$

m equations are applied through the 2^m cases. This implies that there are 2^m possible solutions. These solutions must include Equations (4.71) and (4.72). Each case sets the multiplier β_j or the corresponding inequality constraint g_j to zero. If the multiplier is set to zero, then the corresponding constraint must be feasible for an acceptable solution. If the constraint is set to zero (active constraint), then the corresponding multiplier *must* be positive for a minimum. With this in mind the m equations can be expressed as

$$\beta_j g_j = 0 \rightarrow \text{if } \beta_j = 0 \quad \text{then} \quad g_j < 0$$
$$\text{if } g_j = 0 \quad \text{then} \quad \beta_j > 0 \tag{4.73}$$

If conditions in Equations (4.73) are not met, the design is not acceptable. In implementing Equation (4.73), for each case a simultaneous total of m values and equalities must be assigned. Once these FOC conditions determine a possible solution, the side constraints have to be checked. As evidenced in the examples earlier, this is not built into the FOC. It is only confirmed after a possible solution has been identified. Equations (4.71)–(4.73) are referred to as the Kuhn-Tucker conditions [see Example 4.3 (Section 4.5.2) for additional discussion of the Kuhn-Tucker conditions].

4.5 EXAMPLES

Two examples are presented in this section. The first is an unconstrained problem that has significant use in data reduction. Specifically, it illustrates the problem of curve fitting or regression. The second is the beam design problem explored in Chapters 1 and 2.

4.5.1 Example 4.2

Problem: A set of y, z data is given. It is necessary to find the best straight line through the data. This exercise is termed curve fitting or data reduction or regression. To keep this example simple the data points are generated using the parabolic equation $z = 0.5y^2$. The data are generated in arbitrary order and nonuniform intervals.

Least Squared Error: The objective function that drives a large number of curve fitting/regression methods is the minimization of the least squared error. The construction of the objective requires two entities—the data and the type of mathematical relationship between the data. In this example it is a straight line. Generally it can be any polynomial, or any function for that manner. This is usually the way correlation equations are determined in experimental fluid dynamics and heat transfer. For this example the data are the collection of points (y_i, z_i), $i = 1,2, \ldots, n$. The expected straight line can be characterized by

$$z = x_1 y + x_2$$

where x_1 is the slope and x_2 is the intercept. Using optimizing terminology, there are two design variables. For a cubic polynomial, there will be four design variables. If z_{pi} is the fitted value for the independent variable y_i, the objective function, which is the minimum of the square of the error over all of the data points, can be expressed as

$$\text{Minimize} \quad f(x_1,x_2) = \sum_{i=1}^{n} [z_i - z_{pi}]^2 = \sum_{i=1}^{n} [z_i - (x_1 y_i + x_2)]^2 \tag{4.74}$$

It is possible to complete the formulation with side constraints although that is not necessary. The FOC are

$$\frac{\partial f}{\partial x_1} = \sum_{i=1}^{n} 2[z_i - (x_1 y_i + x_2)](-y_i) = 0 \tag{4.75a}$$

$$\frac{\partial f}{\partial x_2} = \sum_{i=1}^{n} 2[z_i - (x_1 y_i + x_2)](-1) = 0 \tag{4.75b}$$

Expanding the expressions in the brackets and reorganizing as a linear matrix equation

$$\begin{bmatrix} \sum_{i=1}^{n} y_i^2 & \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} y_i & \sum_{i=1}^{n} 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = - \begin{bmatrix} \sum_{i=1}^{n} z_i y_i \\ \sum_{i=1}^{n} z_i \end{bmatrix}$$

Note that the matrices can be set up easily and solved using MATLAB. The Hessian matrix is the square matrix on the left. The SOC requires that it be positive definite.

***Code:* Sec4_5_1.m:** Example 4.2 is solved using MATLAB and the results from the Command window are shown below. Compare this code with that in Chapter 1. The results are displayed on a plot (not included in the book). New features in the code are the use of random numbers, using the sum command, and writing to the Command window. From the plot and the tabular values it is clear that a linear fit is not acceptable for this example. The quality of the fit is not under discussion in this example as it is apparent that a nonlinear fit should have been chosen. This is left as an exercise. The following will appear in the Command window when the code is run.

```
Results from Linear fit
objective function: 16.5827
design variables x1, x2:
  2.3458
 -1.5305
   yi        zi         zp        diff
 4.7427    11.2465     9.5951     1.6514
 3.7230     6.9304     7.2031    -0.2727
 4.3420     9.4266     8.6552     0.7714
 0.3766     0.0709    -0.6470     0.7179
 2.4040     2.8895     4.1089    -1.2193
 4.1054     8.4272     8.1002     0.3270
 4.1329     8.5404     8.1646     0.3757
 3.3369     5.5674     6.2973    -0.7299
 2.7835     3.8739     4.9991    -1.1253
 4.2032     8.8334     8.3296     0.5039
 3.1131     4.8456     5.7723    -0.9267
 3.1164     4.8561     5.7802    -0.9241
 3.2415     5.2538     6.0737    -0.8199
 0.1954     0.0191    -1.0720     1.0911
 0.7284     0.2653     0.1781     0.0871
 4.5082    10.1619     9.0450     1.1168
 1.9070     1.8183     2.9430    -1.1247
 0.4974     0.1237    -0.3638     0.4874
```

```
1.9025     1.8098     2.9325    -1.1227
0.1749     0.0153    -1.1203     1.1356

eigenvalues of Matrix A:
205.4076
  4.5422
```

4.5.2 Example 4.3

This example is the same as Example 2.3, the flagpole problem. It is quite difficult to solve and is chosen here to reveal several practical features that are not easily resolved. These are situations where the designer will bring his experience and intuition into play. The problem was fully developed in Section 2.3.2 and is not repeated here. The graphical solution is available in Figures 2.9 and 2.10. These figures are exploited in this section to determine the correct combination of β's for applying the Kuhn-Tucker conditions. If all of the parameters are substituted in the various functions, the relations can be expressed with numerical coefficients as:

$$\text{Minimize} \quad f(x_1,x_2) = 6.0559\,E\,05(x_1^2 - x_2^2) \tag{4.76}$$

Subject to:

$$g_1(x_1,x_2): 7.4969E\,05x_1^2 + 40000x_1 - 9.7418E\,06(x_1^4 - x_2^4) \le 0 \tag{4.77a}$$

$$g_2(x_1,x_2): (5000 + 1.4994E\,05x_1)(x_1^2 + x_1x_2 + x_2^2) - 1.7083E\,07(x_1^4 - x_2^4) \le 0 \tag{4.77b}$$

$$g_3(x_1,x_2): 1.9091E-03x_1 + 6.1116E-04 - 0.05(x_1^4 - x_2^4) \le 0 \tag{4.77c}$$

$$g_4(x_1,x_2): x_2 - x_1 + 0.001 \le 0 \tag{4.77d}$$

$$0.02 \le x_1 \le 1.0; \qquad 0.02 \le x_2 \le 1.0 \tag{4.78}$$

Kuhn-Tucker Conditions: Four Lagrange multipliers are introduced for the four inequality constraints and the Lagrangian is

$$\begin{aligned} F(X,\beta) = {} & 6.0559\,E\,05(x_1^2 - x_2^2) + \\ & \beta_1(7.4969E\,05x_1^2 + 40{,}000x_1 - 9.7418E\,06(x_1^4 - x_2^4)) + \\ & \beta_2((5000 + 1.4994E\,05x_1)(x_1^2 + x_1x_2 + x_2^2) - 1.7083E\,07(x_1^4 - x_2^4)) + \\ & \beta_3(1.9091E-03x_1 + 6.1116E-04 - 0.05(x_1^4 - x_2^4)) + \\ & \beta_4(x_2 - x_1 + 0.001) \end{aligned} \tag{4.79}$$

where $\mathbf{X} = [x_1, x_2]$ and $\beta = [\beta_1, \beta_2, \beta_3, \beta_4]^T$. The Kuhn-Tucker conditions require two equations using the gradients of the Lagrangian with respect to the design variables

$$\frac{\partial F}{\partial x_1} = 0; \quad \frac{\partial F}{\partial x_2} = 0 \tag{4.80}$$

The actual expression for the gradients obtained from Equation (4.80) is left as an exercise for the student. The Kuhn-Tucker conditions are applied by identifying the various cases based on the activeness of various sets of constraints.

For this problem $n = 2$ and $m = 4$. There are $2^m = 2^4 = 16$ cases that must be investigated as part of the Kuhn-Tucker conditions given by Equation (4.73). While some of these cases are trivial, nevertheless it is a formidable task. This problem was solved graphically in Chapter 2 and the same graphical solution can be exploited to identify the particular case that needs to be solved for the solution. Visiting Chapter 2 and scanning Figure 2.9, it can be identified that constraints g_1 and g_3 are active (as confirmed by the zoomed solution in Figure 2.10). The solution is $x_1^* = 0.68$ m and $x_2^* = 0.65$ m.

If g_1 and g_3 are active constraints, then the multipliers β_1 and β_3 must be positive. By the same reasoning the multipliers associated with the inactive constraints g_2 and g_4, that is, β_2 and β_4, must be set to zero. This information on the active constraints can be used to solve for x_1^*, x_2^* as this is a system of two equations in two unknowns. This does not, however, complete the satisfaction of the Kuhn-Tucker conditions. β_1 and β_3 must be solved and verified that they are positive. g_2 and g_4 must be evaluated and verified that they are less than zero.

Sec4_5_2.m is the code segment that is used to solve the particular case discussed above. The problem is solved symbolically. Two versions of the problem are solved: the original problem and a scaled version. The objective function (4.76) is modified to have a coefficient of unity. This should not change the value of the design variables (why?). You can experiment with the code and verify if this is indeed true. Note the values of the Lagrange multipliers during this exercise and infer the relation between the multipliers and the scaling of the objective function. The following discussion assumes that the code has been run and values available.

Referring to the solution of the case above, the optimal values for the design variables are $x_1^* = 0.6773$ m and $x_2^* = 0.6443$ m. This is very close to the graphical solution. Before we conclude that the solution is achieved, take note of the multipliers: $\beta_1^* = 5.2622e-010$ and $\beta_3^* = -0.1270$. If the multipliers are negative, then it is not a minimum. The values of the constraints at the optimal values of the design are $g_2 = -4.5553e+007$ and $g_4 = -0.0320$. Both values suggest that the constraints are inactive. At least the solution is feasible—all constraints are satisfied. The value of β_1^* is zero and it is also an active constraint ($g_1 = 0$). This corresponds to the trivial case in the Kuhn-Tucker conditions.

Examining other cases that involve intersection of different pairs of inequalities that are assumed to be active, the trivial case shows up often (left as an exercise to explore). The only case when the multipliers are positive is when g_2 and g_3 are

considered active. The solution, however, violates the side constraints. The graphical solution is rather explicit in identifying the minimum at the points established in the first solution. The trivial case of the multiplier being zero when the constraint is active is observed with respect to the first two constraints that express stress relations. In general, the terms in these inequalities have large orders of magnitude, especially if there are other equations that express constraints on displacements, areas, or the like. Large values exert a large influence on the problem so much as to cause lower value constraints to have little or no significance for the problem. This is usually a severe problem in numerical investigations. The remedy for this is scaling of the functions.

Scaling: Consider the order of magnitude in Equations (4.77a) and (4.77c). Numerical calculations are driven by larger magnitudes. Inequality (4.77c) will be ignored in relation to the other functions even though the graphical solution indicates that g_3 is active. This is a frequent occurrence in all kinds of numerical techniques. The standard approach to minimize the impact of large variations in magnitudes among different equations is to *normalize* the relations. In practice this is also extended to the variables. This is referred to as *scaling the variables* and *scaling the functions.* Many current software will scale the problem without user intervention.

Scaling Variables: The presence of side constraints in problem formulation allows a natural definition of scaled variables. The user-defined upper and lower bounds are used to scale each variable between 0 and 1. Therefore,

$$\tilde{x}_i = \frac{x_i - x_i^l}{x_i^u - x_i^l}; \quad \tilde{x}_i = \text{scaled } i\text{th variable} \tag{4.81a}$$

$$x_i = \tilde{x}_i(x_i^u - x_i^l) + x_i^l \tag{4.81b}$$

In the original problem Equation (4.81b) is used to substitute for the original variables after which the problem can be expressed in terms of scaled variable. An alternate formulation is to use only the upper value of the side constraint to scale the design variable:

$$\hat{x}_i = x_i / x_i^u \tag{4.82a}$$

$$x_i = x_i^u \tilde{x}_i \tag{4.82b}$$

While this limits the higher scaled value to 1, it does not set the lower scaled value to zero. For the example of this section, there is no necessity for scaling the design variables since their order of magnitude is one, which is exactly what scaling attempts to achieve.

Scaling the Constraints: Scaling of the functions in the problem is usually critical for a successful solution. Numerical techniques used in optimization are iterative. In each iteration usually the gradient of the functions at the current value of the design variables is involved in the calculations. This gradient, expressed as a matrix, is called the *Jacobian matrix* or simply the *Jacobian.* Sophisticated scaling techniques [7,8] employ the diagonal entries of this matrix as metrics to scale the respective functions. These entries are evaluated at the starting values of the design variables. The function can also be scaled in the same manner as Equations (4.81) and (4.82). For the former an expected lower and upper values of the constraints are necessary. In this exercise, the constraints will be scaled using relations similar to the relations expressed by equalities (4.82). The scaling factor for each constraint will be determined using the starting value or the initial guess for the variables. A starting value of 0.6 for both design variables is selected to compute the values necessary for scaling the functions. The scaling constants for the equations are calculated as

$$\tilde{g}_{10} = 293888.4; \quad \tilde{g}_{20} = 102561.12; \quad \tilde{g}_{30} = 1.7570E-03; \quad \tilde{g}_{40} = 1$$

The first three constraints are divided through by their scaling constants. The last equation is unchanged. The objective function has a coefficient of one. The scaled problem is

$$\text{Minimize} \quad \tilde{f} = x_1^2 - x_2^2 \tag{4.83}$$

$$\tilde{g}_1: 2.5509x_1^2 + 0.1361x_1 - 33.148(x_1^4 - x_1^4) \le 0 \tag{4.84a}$$

$$\tilde{g}_2: (0.0488 + 1.4619x_1)(x_1^2 + x_1x_2 + x_2^2) - 166.5641(x_1^4 - x_2^4) \le 0 \tag{4.84b}$$

$$\tilde{g}_3: 1.0868x_1 + 0.3482 - 28.4641(x_1^4 - x_2^4) \le 0 \tag{4.84c}$$

$$\tilde{g}_4: x_2 - x_1 + 0.001 \le 0 \tag{4.84d}$$

$$0.02 \le x_1 \le 1.0; \qquad 0.02 \le x_2 \le 1.0 \tag{4.79}$$

Sec4_5_2_scaled.m: The code (could have been part of Sec4_5_2.m) investigates the scaled problem in the same way that the original problem was investigated. Primarily, this exploits the information from the graphical solution. It is expected that the solution for the design variables will be the same although the multiplier values are expected to be different. From the information in the Workspace window the optimal values for the design variables are $x_1^* = 0.6774$ and $x_2^* = 0.6445$. All of the constraints are feasible. The optimal values of the multipliers have changed. There is no longer a zero multiplier—it has turned positive. There is still one multiplier with a negative sign implying the point has not met the Kuhn-Tucker conditions. Actually, it should have been anticipated since *changing of the sign* of the multiplier is not possible with scaling.

There is still the matter of inconsistency between the graphical solution and the dissatisfaction of the Kuhn-Tucker conditions. Exploring this particular example has been very torturous but has provided the opportunity to explore important but related quantities like scaling. Another important consideration is the scope of the Kuhn-Tucker conditions. *These conditions apply only at regular points.*

Regular Points: Regular points [8] arise when equality constraints, $\mathbf{h}(\mathbf{X})$, are present in the problem. They are also extended to *active* inequality constraints (pseudo equality constraints). Kuhn-Tucker conditions are valid only for regular points. The two essential features of a regular point $\mathbf{X}^*$ are

- The point $\mathbf{X}^*$ is feasible (satisfies all constraints).
- The gradients of the equality constraints as well as the active inequality constraints at $\mathbf{X}^*$ must form a linear independent set of vectors.

In the scaled problem of Example 4.3 the constraints g_1 and g_3 are active. The solution $x_1^* = 0.6774$ and $x_2^* = 0.6445$ is feasible. The gradient of g_1 is $\nabla g_1 = [-37.631435.4548]^T$. The gradient of g_2 is $\nabla g_2 = [-34.3119\ 30.4786]^T$. It appears that the two gradients are almost parallel to each other. This is also evident in the graphical solution. The gradients therefore are not linearly independent—Kuhn-Tucker conditions cannot be applied at this point. This does not mean the point is not a local minimum. There is graphical evidence that it is indeed so.

Kuhn-Tucker are the only available formal conditions for recognition of the optimum values. It is universally applied without regard to *regularity*. Some additional considerations should be kept in mind [8].

If equality constraints are present and all of the inequality constraints are inactive, then the points satisfying the Kuhn-Tucker conditions may be minimum, maximum, or a saddle point. Higher-order conditions are necessary to identify the type of solution.

If the multipliers of the active inequality constraints are positive, the point cannot be a local maximum. It may not be a local minimum either. A point may be a maximum if the multiplier of an active inequality constraint is zero.

REFERENCES

1. Stein, S. K. *Calculus and Analytical Geometry,* McGraw–Hill, New York. (1987)
2. Hostetler, G. H., Santina, M. S., and Montalvo, P. D., *Analytical, Numerical, and Computational Methods for Science and Engineering,* Prentice–Hall, Englewood Cliffs, NJ. (1991)
3. Moler, C. and Costa, P. J., *Symbolic Math Toolbox—for use with MATLAB—Users Guide,* MathWorks Inc., MA.

4. *MATLAB—Demos Symbolic Tool Box—Introduction,* online resource in MATLAB, MathWorks Inc., MA.
5. Burden, R. L. and Faires, J. D., *Numerical Analysis, 4th ed.,* PWS_KENT Publishing Company, Boston. (1989)
6. Fox, R. L., *Optimization Methods for Engineering Design,* Addison–Wesley, Reading, MA. (1971)
7. Vanderplaats, G. N., *Numerical Optimization Techniques for Engineering Design,* McGraw–Hill, New York. (1984)
8. Arora, J. S., *Introduction to Optimal Design,* McGraw–Hill, New York. (1989)
9. Kuhn, H. W. and Tucker, A. W., *Nonlinear Programming, Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability,* J. Neyman (ed.), University of California Press, 1951.

PROBLEMS

(In many of the problems below, you are required to obtain the numerical solution.)

4.1 Define two nonlinear functions in two variables. Find their solution through contour plots.

4.2 For the functions in Problem 4.1 obtain the gradients of the function and the Jacobian. Confirm them using the Symbolic Math Toolbox.

4.3 Define the design space (chooses side constraints) for a two-variable problem. Define two nonlinear functions in two variables that do not have a solution within this space. Graphically confirm the result.

4.4 Define a design space for a two-variable problem. Define two nonlinear functions in two variables that have at least two solutions within the space.

4.5 Define a nonlinear function of two variables. Choose a contour value and draw the contour. Identify a point on the contour. Calculate the value of the gradient at that point. Draw the gradient at the point using the computed value. Calculate the Hessian at the above point.

4.6 Using the relation in Equation (4.23), establish that the gradient is normal to the tangent [a physical interpretation for Equation (4.23)].

4.7 Define a nonlinear function of three variables. Choose a point in the design space. Find the gradient of the function at the point. Calculate the Hessian matrix at the same point.

4.8 Express the Taylor series expansion (quadratic) of the function $f(x) = (2 - 3x + x^2) \sin x$ about the point $x = 0.707$. Confirm your results through the Symbolic Math Toolbox. Plot the original function and the approximation.

4.9 Expand the function $f(x,y) = 10\,(1 - x^2)^2 + (y - 2)^2$ quadratically about the point (1,1). How will you display the information? Draw the contours of the original function and the approximation.

4.10 Obtain the solution graphically for the case when Equation (4.10) is added to the problem defined by Equations (4.36), (4.39), and (4.37). Obtain the solution using the Symbolic Math Toolbox.

4.11 Obtain the solution to the optimization problem in Section 4.4.4 using MATLAB.

4.12 In Section 4.5.1, obtain the coefficients for a quadratic fit to the problem.

4.13 Use an example to show that optimum values for the design variables do not change if the objective function is multiplied by a constant. Prove the same if a constant is added to the function.

4.14 How does scaling of the objective function affect the value of the multipliers? Use an example to infer the result. How does scaling the constraint affect the multipliers ? Check with an example.

5

NUMERICAL TECHNIQUES—THE ONE-DIMENSIONAL PROBLEM

The one-dimensional unconstrained optimization problem is often introduced and solved numerically in most courses that deal with numerical analysis/techniques. It appears indirectly, notably as finding the root of a nonlinear equation, which is the same as satisfying the FOC for an unconstrained optimization problem. Two of the popular techniques are the Newton-Raphson and the bisection technique. They are introduced in Section 5.2 to provide a comparison with the methods used in optimization. In engineering design the single-variable optimization problem is probably a triviality, but one-dimensional optimization is a critical component of multivariable design and is discussed in Section 5.3. Meanwhile the classical root finding methods identified above do not play any significant part in numerical optimization techniques as they are considered computationally expensive and not robust enough over a large class of problems. An important consideration in nonlinear design optimization is that during the initial iterations, when you are typically far away from the solution, accuracy can be traded for speed. In these iterations it is also important to move through to the next iteration, instead of faltering at the current one. These criteria have required that the one-dimensional techniques be simple in concept as well as easily implementable. Two different techniques, the polynomial approximation and the golden section, provide popular support for most optimization software. Very often the two techniques are implemented in combination. They are discussed in Section 5.2.

5.1 PROBLEM DEFINITION

In order to connect with the discussions later, the one-dimensional variable will be identified as α instead of x or x_1. In the following chapters this is also referred to as the one-dimensional *stepsize* computation.

Example 5.1 This example is similar to one in Chapter 4:

$$\text{Minimize} \quad f(\alpha) = (\alpha - 1)^2(\alpha - 2)(\alpha - 3) \tag{5.1}$$

$$\text{Subjectto:} \quad 0 \le \alpha \le 4 \tag{5.2}$$

The problem does not represent any real design. It was constructed to have multiple local minimums within the area of interest. Expression (5.1) represents an unconstrained problem. The side constraints accompany all problems in this book to convey the idea that there are not truly unconstrained problems. Side constraints also define an acceptable design region for all problems. Section 5.2 will explore this problem through the various methods mentioned earlier.

5.1.1 Constrained One-Dimensional Problem

The only constraint that Example 5.1, a single-variable problem, can accommodate is *inequality* constraint. From the previous chapter there can be more than one such constraint. A single-variable problem cannot have an equality constraint (Chapter 4). In a constrained multivariable optimization problem it will often be necessary to find the stepsize α such that the constraint just becomes active. It is possible to define the problem.

Example 5.1a

$$\text{Minimize} \quad f(\alpha) = (\alpha - 1)^2(\alpha - 2)(\alpha - 3) \tag{5.1}$$

$$\text{Subject to:} \quad g(\alpha): 0.75\alpha^2 - 1.5\alpha - 1 \le 0 \tag{5.3}$$

$$0 \le \alpha \le 4 \tag{5.2}$$

In actual applications it is more likely that expressions (5.1)–(5.3) establish two problems that provide different solutions to the stepsize α. The first is the solution to expressions (5.1) and (5.2), which is the same as Example 5.1. The second is the solution to

$$g(\alpha): 0.75\alpha^2 - 1.5\alpha - 1 = 0 \tag{5.3}$$

$$0 \le \alpha \le 4 \tag{5.2}$$

The second solution is useful only if expression (5.3) is an active constraint. Otherwise the solution to Example 5.1 will prevail. The final solution chosen ensures expressions (5.1)–(5.3) are satisfied. A graphical investigation in the next section should illustrate this feature clearly.

5.2 SOLUTION TO THE PROBLEM

The solution to Examples 5.1 and 5.1a is examined graphically. Next the classical root finding techniques of Newton-Raphson and the bisection method are presented. These are followed by the polynomial approximation and the golden section method.

5.2.1 Graphical Solution

The example has already made its appearance in Chapter 4. Figure 5.1 is a plot of the objective function in Example 5.1. It also includes the solution to Example 5.1a. If expression (5.3) represents an inequality constraint, the feasible region for one-variable functions is quite different from the two-variable problems encountered in

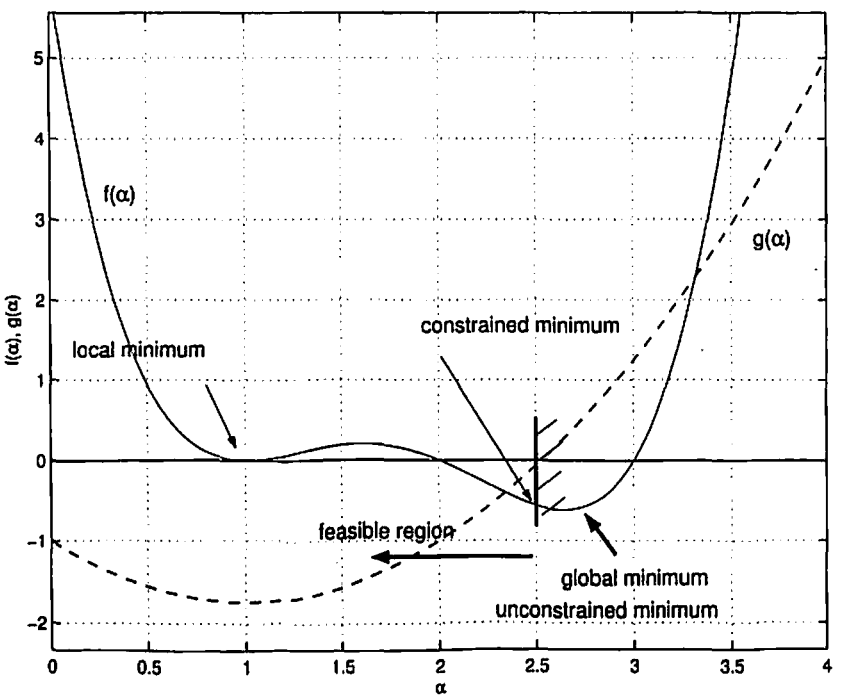

Figure 5.1 Graphical solution: Example 5.1.

Chapter 2. It is shown by the hash marks drawn by invoking the *plotedit* utility through the figure window. Several annotations are included on the figure through editing the plot. **Sec5_2_1.m**[1] will generate the basic plot. From the figure and a few symbolic computations in the accompanying MATLAB Command window, the following solutions are obtained.

Example 5.1

$$\alpha^* = 2.6484; \quad f^* = -0.6195 \tag{5.4a}$$

Example 5.1a

$$\alpha^* = 2.5275; \quad f^* = -0.5815 \tag{5.4b}$$

5.2.2 Newton-Raphson Technique

The Newton-Raphson technique, also referred to as Newton's technique, is a gradient-based solution to finding the root of a single nonlinear equation. Root finding problems involve identifying the variable for which the equation has a value of zero. Typically they are postulated as the solution to

$$\phi(\alpha) = 0 \tag{5.5}$$

where $\phi(\alpha)$ is a nonlinear equation in the variable α.

The technique has additional features:

- It has a geometric underpinning.
- It uses the Taylor series expanded linearly.
- It is iterative.
- It has the property of quadratic convergence.

Any iterative technique addressing Equation (5.5) can be summarized as

Generic Algorithm (A5.1)

Step 1. Assume α

Step 2. Calculate $\Delta\alpha$

Step 3. Update $\tilde{\alpha} = \alpha + \Delta\alpha$

If converged ($\phi(\tilde{\alpha}) = 0$) then exit

If not converged ($\phi(\tilde{\alpha}) \neq 0$) then $\alpha \leftarrow \tilde{\alpha}$

go to Step 2

[1]Files to be downloaded from the web site are indicated by boldface sans serif type.

The above sequence is fairly standard in iterative techniques for many classes of problems. It also captures the essence for most of the techniques to follow in this book. Step 1 indicates the primary feature of iterative methods—the *starting solution*. This is an initial guess provided by the user to start the iterative process. The iterative process is continued to the solution by calculating changes in the variable ($\Delta\alpha$). The value of the variable is updated. Convergence is checked. If Equation (5.5) is satisfied, then the solution has been found. If convergence is not achieved, then the updated value provides the value to proceed with the next iteration. Of course, $\Delta\alpha$ can be calculated in many ways and Newton-Raphson is just one of the ways to establish the change in the variable.

Calculation of $\Delta\alpha$ (Newton-Raphson): Let α be the current value of the variable. It is assumed that Equation (5.5) is not satisfied at this value of the variable (the reason for us to iterate). Let $\tilde{\alpha}$ be a neighboring value. Ideally we would like to achieve convergence at this value (even as we are aware it might take us several iterations to achieve convergence). Using Taylor's theorem expanded to the linear term only,

$$\phi(\tilde{\alpha}) = \phi(\alpha + \Delta\alpha) = \phi(\alpha) + \frac{d\phi}{d\alpha}\Delta\alpha = 0 \tag{5.6}$$

$$\Delta\alpha = -\frac{\phi(\alpha)}{\frac{d\phi}{d\alpha}} = -\left[\frac{d\phi}{d\alpha}\right]^{-1}\phi(\alpha) \tag{5.7}$$

For Equation (5.7) to be effective, the gradient of the function should not be zero. It is also inevitable that changes in α will be large where ϕ is flat, and small where the slope is large. For the Newton-Raphson method to be effective the iterations should avoid regions where the slope is small. This is a serious disadvantage of this method. Figure 5.2 illustrates the geometrical construction of the technique at the point $\alpha = 3$.

Example 5.1 The Newton-Raphson technique is applied to the first-order conditions (FOC) of Example 5.1.

$$\text{Minimize} \quad f(\alpha) = (\alpha - 1)^2(\alpha - 2)(\alpha - 3) \tag{5.1}$$

$$\text{Subject to:} \quad 0 \leq \alpha \leq 4 \tag{5.2}$$

The FOC is

$$\phi(\alpha) = \frac{df}{d\alpha} = 2*(\alpha - 1)*(\alpha - 2)*(\alpha - 3) + (\alpha - 1)\text{^}2*(\alpha - 3) + (\alpha - 1)\text{^}2*(\alpha - 2) = 0 \tag{5.8}$$

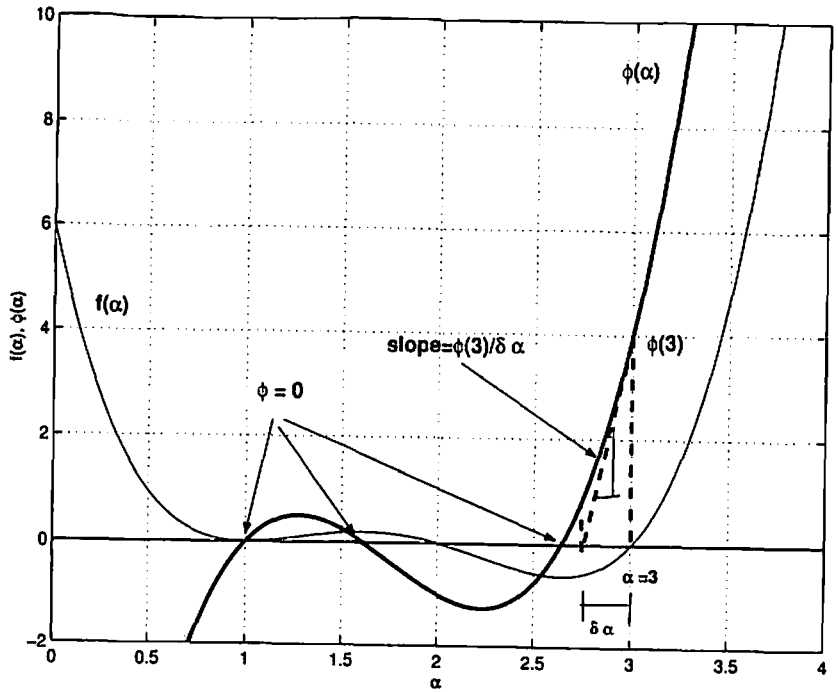

Figure 5.2 Development of Newton-Raphson technique.

The Technique: Two iterations are shown here. The complete application is done through MATLAB code **Sec5_2_2.m**. The code uses symbolic calculation. In running the code the figure window displays an animation of the iterative travel to the solution. Also shown is the change in $\Delta\alpha$ with each iteration. The iterative results are summarized in a table at the end. The calculations require the gradient of $\phi(\alpha)$:

$$\phi'(\alpha) = 2*(\alpha-2)*(\alpha-3) + 4*(\alpha-1)*(\alpha-3) + 4*(\alpha-1)*(\alpha-2) + 2*(\alpha-1)\text{^}2$$

Iteration 1:

$$\alpha = 3; \quad f(\alpha) = 0.0; \quad \phi(\alpha) = 4 \quad \phi'(\alpha) = 16 \quad \Delta\alpha = -0.25$$

$\phi(\alpha + \Delta\alpha) = 0.875$—not converged, at least another iteration necessary

Iteration 2:

$$\alpha = 2.75; \quad f(\alpha) = -0.5742; \quad \phi(\alpha) = 0.0875; \quad \phi'(\alpha) = 9.25; \quad \Delta\alpha = -0.0946$$

$\phi(\alpha + \Delta\alpha) = 0.1040$—not converged, at least another iteration necessary

The code **Sec5_2_2.m** requires five iterations to converge the value of ϕ to 1.0E-08 (considered zero here). This value of 1.0E-08 is termed the *convergence* (or *stopping*) criterion. The acceleration of convergence or rate of convergence to the solution is formally defined in many different ways. One of the ways is to monitor the ratio of $\phi(\alpha + \Delta\alpha)/\phi(\alpha)$. Another is to track $\Delta\alpha$. In applying and understanding numerical techniques it is important to develop an instinct and feel for the numbers corresponding to various functions as the iterations progress. The speed or the quality of convergence can be gauged by the shift in the decimal point in the quantity of interest through the iterations. Consider the value of ϕ through the five iterations [4.0000, 0.8750, 0.1040, 0.0023, 0.0000]. Definitely the changes in ϕ are nonlinear.

The Newton-Raphson technique can be directly applied to Equation (5.3)—after all the method is designed to operate on these kinds of problems. It is left as an exercise for the reader. There is a very important feature of most iterative techniques that explore multiple solutions—as is the case of Example 5.1. Rerun the code **Sec5_2_2.m** with a starting value of 0.5 for α. After six iterations, α converges to the value of 1.0, which is a local minimum. Where would iterations lead if the start value is 1.5 ? These answers are easy to see graphically in a one- or maybe a two-variable problem. In a typical engineering design problem, significant experience and intuition are required to recognize these situations when they occur. A healthy suspicion of the solution must always be in the back of one's mind when exploring multivariable problems. In summary, *most iterative numerical techniques are designed to converge to solutions that are close to where they start from.*

5.2.3 Bisection Technique

This is another popular technique to find the root of a function. It is also called a binary search or interval halving procedure. Unlike the Newton-Raphson method this procedure does not require the evaluation of the gradient of the function whose root is being sought. One of the reasons the method is being included is to compare it with the golden section method discussed later. The numerical technique in this method is based on the reasoning that the root (or zero) of a function is trapped or bound between a positive and a negative value of the function. The actual solution achieved by the method is an interval in which the zero or the solution can be located. The final interval is typically determined to be a very small value, called the *tolerance*, of the order of 10E-08. Since this is a root finding procedure, when applied to optimization it is applied to the gradient of the objective which we identified as $\phi(\alpha)$ Equation (5.5). Establishing the minimum of the objective is thereby translated to locating the root of the gradient function. It is possible to develop a bisection technique for directly handling the minimization problem.

The method requires two points to start, say α_a and α_b. The value of ϕ at these points must be opposite in sign. These can be the side constraints of the one-dimensional optimization problem. It is assumed that at least one solution exists in this initial interval. During each iteration, this interval is halved with the midpoint of the interval replacing either α_a or α_b, while keeping the root still trapped between

function values of opposite sign. The iterative technique is expressed as the following algorithm (a set of procedures that are repeated).

Algorithm for Bisection Method (A5.2)

Step 1: Choose α_a and α_b to start. Let $\alpha_a < \alpha_b$

Step 2: Set $\alpha = \alpha_a + (\alpha_b - \alpha_a)/2$

Step 3: If $\phi(\alpha) = 0.0$ – Converged Solution – exit

Else If $(\alpha_b - \alpha_a) < 10$ E-04 – tolerance met – exit

Else If $\phi(\alpha) * \phi(\alpha_a) > 0.0$; **then** $\alpha_a \leftarrow \alpha$

Else $\alpha_b \leftarrow \alpha$

go to Step 2

We consider Example 5.1 which is reproduced here

$$\text{Minimize} \quad f(\alpha) = (\alpha - 1)^2(\alpha - 2)(\alpha - 3) \tag{5.1}$$

$$\text{Subject to:} \quad 0 \le \alpha \le 4 \tag{5.2}$$

The FOC is

$$\phi(\alpha) = \frac{df}{d\alpha} = 2*(\alpha - 1)*(\alpha - 2)*(\alpha - 3) + (\alpha - 1)\text{^}2*(\alpha - 3) + (\alpha - 1)\text{^}2*(\alpha - 2) = 0 \tag{5.8}$$

Sec5_2_3.m: This MATLAB code runs the bisection method on Example 5.1. It illustrates the trapping of the minimum by the two values on either side of the root. There are 17 iterations for a tolerance of 1.0 E-04 which terminated the program. Compare this with 5 iterations for the Newton-Raphson method, for a better solution. The lower number of iterations can be related to the quality of information used to update the design variable change. The gradient-based information used in the Newton-Raphson technique is more sophisticated, though the gradient computation is an additional work load.

The problem described by Equation (5.3) can be handled directly by the bisection method since $g(\alpha)$ is the same as $\phi(\alpha)$. In a way, these two methods also provide an opportunity to revisit prior knowledge, particularly with respect to defining and translating an algorithm into running code. There is usually a strong correspondence between the step-by-step description of the numerical technique (the algorithm) and the code that will implement the steps to effect a solution (for example, **Sec5_2_3.m**). Both methods handle the minimization indirectly as a matter of finding the root of the FOC for the unconstrained one-variable problem. The following two methods are used extensively for one-dimensional minimization. They handle the minimization problem directly.

5.2.4 Polynomial Approximation

The method is simple in concept. Instead of minimizing a difficult function of one variable, minimize a polynomial that approximates the function. The optimal value of the variable that minimizes the polynomial is then considered to approximate the optimal value of the variable for the original function. It is rare for the degree of the approximating polynomial to exceed three. A quadratic approximation is standard unless the third degree is warranted. It is clear that serious errors in approximation are expected if the polynomial is to simulate the behavior of the original function over a large range of values of the variable. Mathematical theorems exist that justify a quadratic representation of the function, with a prescribed degree of error, within a small neighborhood of the minimum. What this ensures is that the polynomial approximation gets better as the minimum is being approached. Example 5.1, reproduced for convenience, is used for illustration of the procedure.

Example 5.1

$$\text{Minimize} \quad f(\alpha) = (\alpha - 1)^2(\alpha - 2)(\alpha - 3) \tag{5.1}$$

$$\text{Subject to:} \quad 0 \le \alpha \le 4 \tag{5.2}$$

A quadratic polynomial $P(\alpha)$ is used for the approximation. This polynomial is expressed as

$$P(\alpha) = b_0 + b_1\alpha + b_2\alpha^2 \tag{5.9}$$

Two elements need to be understood prior to the following discussion. The first concerns the evaluation of the polynomial, and the second concerns the inclusion of expression (5.2) in the discussion. The polynomial is completely defined if the coefficients b_0, b_1, and b_2 are known. To determine them, three data points $[(\alpha_1, f_1), (\alpha_2, f_2), (\alpha_3, f_3)]$ are generated from Equation (5.1). This sets up a linear system of three equations in three unknowns by requiring that the values of the function and the polynomial must be the same at the three points. The solution to this system of equations is the values of the coefficients. The consideration of expression (5.2) depends on the type of one-dimensional problem being solved. If the one-dimensional problem is a genuine single-variable design problem, then expression (5.2) needs to be present. If the one-dimensional problem is a subproblem from the multidimensional optimization problem, then expression (5.2) is not available. In that case a *scanning* procedure is used to define α_1, α_2, and α_3.

Scanning Procedure: The process is started from the lower limit for α. A value of zero for this value can be justified since it refers to values at the current iteration. A constant interval for α, $\Delta\alpha$ is also identified. For a well-scaled problem this value is usually 1. Starting at the lower limit, the interval is doubled until three points are

determined such that the minimum is bracketed between them. With respect to Example 5.1, the scanning procedure generates the following values:

$\alpha = 0;\ f(0) = 6;\ \alpha_1 = 0;\ f_1 = 6$

$\alpha = 1;\ f(1) = 0;\ \alpha_2 = 1;\ f_2 = 0$

$\alpha = 2;\ f(2) = 0;$ this cannot be α_2 as the minimum is not yet trapped

$\alpha = 4;\ f(4) = 18;\ \alpha_2 = 4;\ f_2 = 18$

A process such as that illustrated is essential as it is both indifferent to the problem being solved and can be programmed easily. The important requirement of any such process is to ensure that the minimum lies between the limits established by the procedure. This procedure is developed as MATLAB m-file **UpperBound_1Var.m**.

The set of linear equations to establish the coefficients of the polynomial in Equation (5.9) is

$$6 = b_0 + b_1(0) + b_2(0) \tag{5.10a}$$

$$0 = b_0 + b_1(1) + b_2(1) \tag{5.10b}$$

$$18 = b_0 + b_1(4) + b_2(16) \tag{5.10c}$$

Using MATLAB the solution to the equations is

$$b_0 = 6.0;\ b_1 = -9;\ b_2 = 3 \tag{5.11}$$

and the polynomial is

$$P(\alpha) = 6 - 9\alpha + 3\alpha^2$$

The minimum for the polynomial is the solution to

$$\frac{dP}{d\alpha} = -9 + 6\alpha = 0;\ \alpha_p^* = \alpha^* = 1.5$$

The approximate solution to Equation (5.1) is 1.5. Figure 5.3 describes the polynomial approximation procedure. It is clear that the approximation leaves much to be desired. On the other hand, if the data for the polynomial coefficients were around a smaller region near $\alpha = 2.6$, then the results would be more impressive. Note that rather than use a higher-order polynomial, a better set of data is preferred.

The scanning procedure and the polynomial approximation techniques will find significant use exploring multidimensional optimization problems. The strategy for developing MATLAB code from now on must include the possibility of code reuse. The same code segments can be used in many different numerical techniques. Such generic

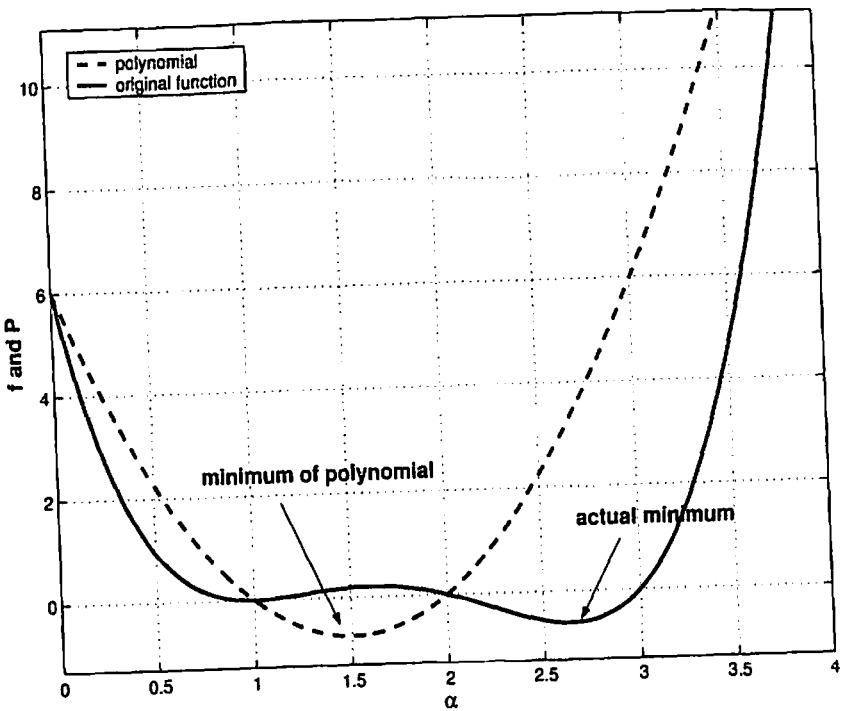

Figure 5.3 Polynomial approximation.

code development is termed a *block structured* approach. In this way, large code development is possible using smaller pieces of existing code.

UpperBound_1Var.m: This code segment implements the determination of the upper bound of a function of one variable. The input to the function is the name of the function m-file, for example, "Example5_1" the start value for the scan (a0), the scanning interval (da), and the number of scanning steps (ns). The function outputs a vector of two values. The first is the value of the variable and the second is the corresponding value of the function.

The function referenced by the code must be a MATLAB m-file, in the same directory **(Example5_1.m)**. The input for **Example5_1.m** is the value at which the function needs to be computed, and its output is the value of the function.

Usage: `UpperBound_1Var('Example5_1',0,1,10)` (5.12)

PolyApprox_1Var.m: This code segment implements the polynomial approximation method for a function of one variable. This function uses

UpperBound_1Var.m to determine the range of the variable. The input to the function is the *name* of the function; the *order* (2 or 3) of the approximation; *lowbound*—the start value of the scan passed to **UpperBound_1Var.m**; *intvlstep*—the scanning interval passed to **UpperBound_1Var.m**; intrials—the number of scanning steps passed to **UpperBound_1Var.m**. The output of the program is a vector of two values. The first element of the vector is the location of the minimum of the approximating polynomial, and the second is the function value at this location.

The function referenced by the code must be a MATLAB m-file, in the same directory (**Example5_1.m**). The input for Example5_1 is the value at which the function needs to be computed, and its output is the value of the function.

Usage: `Value = PolyApprox_1Var('Example5_1',2,0,1,10)` (5.13)

The two programs use the *switch/case* and the *feval* statements from MATLAB. The code also illustrates calling and returning from other functions.

5.2.5 Golden Section Method

The method of golden section is the cream of the family of interval reducing methods (for example, the bisection method). It reduces the interval by the same fraction with each iteration. The intervals are derived from the *golden section ratio*, 1.61803. This ratio has significance in aesthetics as well as mathematics [3,4]. The method is simple to implement. It is indifferent to the shape and continuity properties of the function being minimized. Most important, the number of iterations to achieve a prescribed tolerance can be established before the iterations start.

Algorithm for Golden Section Method (A5.3)

Step 1. Choose α^{low}, α^{up},
$\tau = 0.38197$ (from Golden Ratio),
ε = tolerance = $(\Delta\alpha)_{final}/(\alpha^{up}-\alpha^{low})$
N = number of iterations = $-2.078 \ln \varepsilon$
i = 1

Step 2. $\alpha_1 = (1-\tau)\alpha^{low} + \tau\alpha^{up}; f_1 = f(\alpha_1)$
$\alpha_2 = \tau\alpha^{low} + (1-\tau)\alpha^{up}; f_2 = f(\alpha_2)$
The points are equidistant from bounds

Step 3. If (i < N)
If $f_1 > f_2$
$\alpha^{low} \leftarrow \alpha_1; \alpha_1 \leftarrow \alpha_2; f_1 \leftarrow f_2$
$\alpha_2 = \tau\alpha^{low} + (1-\tau)\alpha^{up}; f_2 = f(\alpha_2)$
$i \leftarrow i+1$
Go To Step 3
If $f_2 > f_1$

$\alpha^{up} \leftarrow \alpha_2; \alpha_2 \leftarrow \alpha_1; f_2 \leftarrow f_1$
$\alpha_1 = (1-\tau)\alpha^{low} + \tau\alpha^{up}; f_1 = f(\alpha_1)$
$i \leftarrow i+1$
Go To Step 3

In the above the sign ← implies *replace the value on the left-hand side with the value on the right-hand side.* Several iterations for Example 5.1 are shown below. The complete set of iterations can be obtained by executing the code **Golden_Sect_1Var.m**. The example is reintroduced for convenience. This m-file only calculates the values. The algorithm is implemented in a different file.

Example 5.1

$$\text{Minimize} \quad f(\alpha) = (\alpha-1)^2(\alpha-2)(\alpha-3) \tag{5.1}$$

Golden Section Method

Step 1. $\alpha^{low} = 0.0; f^{low} = 6; \alpha^{up} = 4; f^{up} = 18$
$\Delta\alpha_{final} = 0.001; \varepsilon = (0.001/4); N = 17$
$i = 1$

Step 2. $\alpha_1 = 1.5279; f_1 = 0.1937; \alpha_2 = 2.4721; f_2 = -0.5401$

Figure 5.4 illustrates the layout of information at this stage. It can be verified that α_1, α_2 are located symmetrically with respect to outer bounds $\alpha^{low}, \alpha^{up}$.

Step 3. i = 1
$f_1 > f_2$
$\alpha^{low} = 1.5279; f^{low} = 0.1937; \alpha_1 = 2.4721; f_1 = -0.5401$
$\alpha_2 = 3.0557; f_2 = 0.2486$
i = 2

Step 3. continued . . .
$f_2 > f_3$
$\alpha^{up} = 3.0557; f^{up} = 0.2486; \alpha_2 = 2.4721; f_2 = -0.5401$
$\alpha_1 = 2.1115; f_2 = -0.1224$
i = 3
Go To Step 3 etc

In each iteration there is only one set of calculation to evaluate the variable and the value of the function. The rest of the computation is reassigning existing information. The difference in the outer bounds is also changing with each iteration. In Example 5.1

Start: $\Delta\alpha = 4.0$

Iteration 1: $\Delta\alpha = 4.0 - 1.5279 = 2.4721$

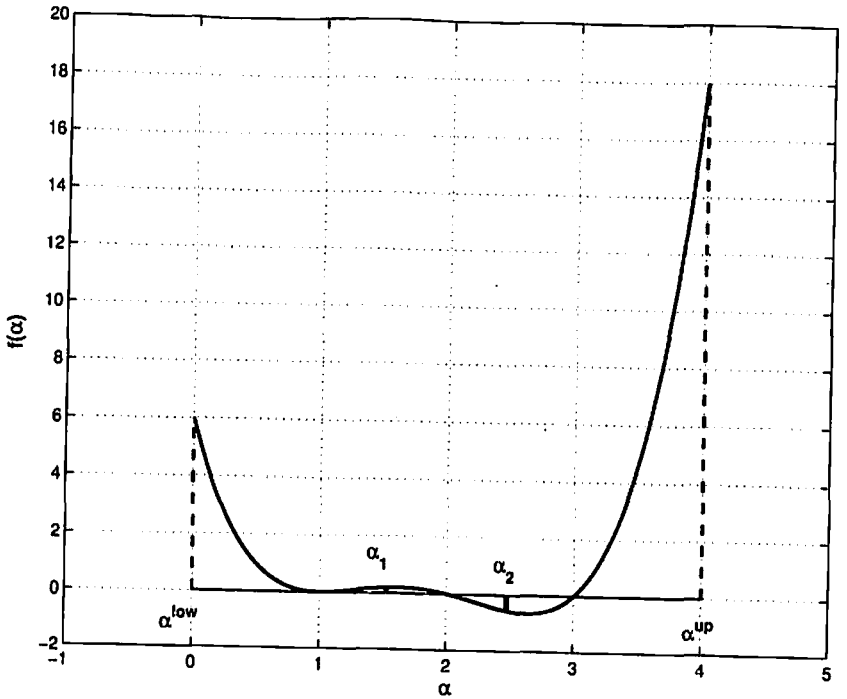

Figure 5.4 Golden section—starting values.

Iteration 2: $\Delta\alpha = 3.0557 - 1.5279 = 1.5278$

. . .

The final iteration is reproduced by running the following code:

iteration 17

2.6395	2.6399	2.6402	2.6406
−0.6197	−0.6197	−0.6197	−0.6197

The tolerance $\alpha_2 - \alpha_1$ in the final iteration is about 0.001 as expected.

GoldSection_1Var.m: The code translates the algorithm for the golden section method (A 5.3) into MATLAB code. The input to the function is the name of the function *(functname)* whose minimum is being sought; the tolerance *(tol)* of the approximation; the start value *(lowbound)* of the scan passed to **UpperBound_1Var.m**; the scanning interval *(intvl)* passed to **UpperBound_1Var.m**; the number of scanning steps *(ntrials)* passed to **UpperBound_1Var.m**. The output of the program is a

vector of four pairs of variable and function values after the final iteration. With a small tolerance any one of these pairs provides the minimum location of the function. It is recommended that the second pair be accessed for this purpose. The function referenced by the code must be a MATLAB m-file, in the same directory (**Example5_1.m**). The input for **Example5_1.m** is the value at which the function needs to be computed, and its output is the value of the function.

Usage:

```
Value = GoldSection_1Var('Example5_1',0.001,0,1,10)
```
(5.14)

There are no new commands. The organizing and displaying of information in the Command window is worth noting. If this code is part of a larger program, then the printing of information in the command window needs to be switched off. It can be turned on for debugging.

Comparison with Polynomial Approximation: The two popular methods for minimizing a function of one variable are polynomial approximation or golden section. The algorithms for polynomial approximation and the golden section have significant differences. The former is a one-shot approach and is accompanied by significant error in the estimation of the minimum, which will improve as the data for approximation get better. Implied in the implementation of the polynomial approach is the continuity of the function. The golden section method, on the other hand, is an iterative technique. The number of iterations depends on the tolerance expected in the final result and is known prior to the start of the iterations—a significant improvement relative to the Newton-Raphson method where the number of iterations cannot be predicted a priori. The implementation is simple and the results impressive as it is able to home in on the minimum. It is indifferent to the nature of the function.

There is no reason why the two cannot be combined. The golden section can be used to establish four data points with a reasonable tolerance (instead of a low value) and a cubic polynomial can be fit to identify the approximate location of minimum.

5.3 IMPORTANCE OF THE ONE-DIMENSIONAL PROBLEM

The one-dimensional subproblem in a multivariable optimization problem is employed for the determination of the stepsize after the search direction has been identified. It is easier to understand these terms by recognizing the generic algorithm for unconstrained optimization. In order to focus the discussion, rather than the general objective function representation, a specific one, Example 5.2, is introduced.

Example 5.2

Minimize $f(x_1,x_2,x_3) = (x_1 - x_2)^2 + 2(x_2 - x_3)^2 + 3(x_3 - 1)^2$ (5.15)

A cursory glance at Example 5.2 indicates that the solution is at $x_1 = 1; x_2 = 1; x_3 = 1$; and the minimum value of f is 0. Three variables are chosen because we are not going to use any graphics for illustration (it is essential to follow the vector algebra).

Generic Algorithm (A5.4): The generic algorithm is an iterative one and is also referred to as a *search algorithm*, as the iterations take place by moving along a *search direction*. These search directions can be determined in several ways (Chapter 6). The algorithm without any convergence/stopping criteria can be expressed as

Step 1. Choose $\mathbf{X}_0$

Step 2. For each iteration i

Determine search direction $\mathbf{S}_i$ vector

Step 3. Calculate $\Delta\mathbf{X}_i = \alpha_i \mathbf{S}_i$

Note: $\Delta\mathbf{X}_i$ is now a function of the scalar α_i as $\mathbf{S}_i$ is known from Step 2

α_i is called the step size as it establishes the length of $\Delta\mathbf{X}_i$

$\mathbf{X}_{i+1} = \mathbf{X}_i + \Delta\mathbf{X}_i$

α_i is determined by Minimizing $F(\mathbf{X}_{i+1})$

This is referred to as one-dimensional step size computation

$i \leftarrow i + 1$; Go To Step 2

Application of Generic Algorithm (A5.4) to Example 5.2

Step 1. $\mathbf{X}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$ (assume)

Step 2. $\mathbf{S}_1 = \begin{bmatrix} 0 \\ 0 \\ 6 \end{bmatrix}$ (assume—this is negative of the gradient of f at X_0)

Step 3. $\Delta\mathbf{X}_1 = \alpha_1 \begin{bmatrix} 0 \\ 0 \\ 6 \end{bmatrix}; \mathbf{X}_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \alpha_1 \begin{bmatrix} 0 \\ 0 \\ 6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 6\alpha_1 \end{bmatrix}$

$$f(\mathbf{X}_1) = f(\alpha_1) = (0-0)^2 + 2(0-6\alpha_1)^2 + 3(6\alpha_1 - 1)^2 = 72\alpha_1^2 + 3(6\alpha_1 - 1)^2$$

Minimizing $f(\alpha_1)$, the value of $\alpha_1 = 0.1$. This is a one-dimensional optimization problem.

While the solution above was obtained analytically because it was a simple linear expression, the polynomial approximation or the golden section method could have been used instead.

$$X_1 = \begin{bmatrix} 0 \\ 0 \\ 0.6 \end{bmatrix} \text{ Go To Step 2} \tag{5.16}$$

and so on.

5.4 ADDITIONAL EXAMPLES

In this section additional examples of single-variable optimization problems are explored. The available code for the polynomial approximation and golden section should help solve any and all single-variable optimization problems. This is largely left to the reader to exploit. In this section only extensions, modifications, or creative applications of the one-dimensional optimization problem are considered. First, Example 5.2 is revisited to modify the golden section method so that it can be used to calculate the stepsize in multidimensional problems. Example 5.3 is a solution to the Blassius problem. This is an example of the exact solution to the Navier-Stokes equation for flow over a flat plate. Example 5.4 is an examination of the inequality constraint in Example 5.1a by the golden section method so that equality and inequality constraints can be handled.

5.4.1 Example 5.2—Illustration of General Golden Section Method

The example from the previous section is visited again to modify the golden section method to determine the stepsize for multidimensional problem. This book is also about getting MATLAB to work for you. In a sense this subsection is largely an exercise in developing the code that will see considerable use in the next chapter. An equivalent development with respect to the polynomial approximation method is left as an exercise for the student.

The golden section method requires the establishment of the upper bound. Therefore, two functions will need to be changed to handle design vectors and the search direction vectors. The code is available as **UpperBound_nVar.m** and **GoldSection_nVar.m**. The modifications are not very challenging and can be inferred by comparing the usage of the two functions.

For the single variable:

Usage: `UpperBound_1Var('Example5_1',0,1,10)` (5.12)

For many variables:

Usage: `UpperBound_nVar('Example5_2',x,s,0,1,10)` (5.17)

`x: current position vector or design vector`

`s: prescribed search direction`

Of course, **Example5_2.m** returns the value of the function corresponding to a vector of design variables. This implies that the function call to **Example5_2.m** can only take place after the design variables are evaluated from the value of stepsize applied along the prescribed search direction from the current location of the design.

The usage of GoldSection_nVar.m must accommodate function (5.17) and the need to deal with a design vector. Comparing with the single-variable case

Usage:

```
Value = GoldSection_1Var('Example5_1',0.001,0,1,10)
```
(5.14)

Usage:

```
Value = GoldSection_nVar('Example5_2',0.001,x,s,0,1,10)
```
(5.18)

Visit the code to see the details.

Comparison with Section 5.3: The code **GoldSection_nVar.m** was run from the Command window using the following listing:

```
>> x = [0 0 0 ];
>> s = [ 0 0 6];
>> GoldSection_nVar('Example5_2',0.001,x,s,0,1,10)
```

After 11 iterations the final line in the Command window was

```
ans =
0.1000    1.2000         0         0 0.5973
```

The above is interpreted as:

$$\alpha_1 = 0.1000; \quad f(\alpha_1) = 1.200; \quad x_1 = 0; \quad x_2 = 0.0; \quad x_3 = 0.5973 \tag{5.19}$$

which matches Section 5.3 and Equation (5.16).

5.4.2 Example 5.3—Two-Point Boundary Value Problem

A real example for one-variable optimization can be found in the numerical solution of the laminar flow over a flat plate [5]. The problem, usually attributed to Blassius, represents an example of the exact solution of the formidable Navier-Stokes equation. The mathematical formulation, allowing for similarity solutions, is expressed by a third-order nonlinear ordinary differential equation with boundary conditions specified at two points—a two-point boundary value problem (TPBVP). What follows is the essential mathematical description. The interested reader can follow the details in most books on fluid mechanics, including the suggested reference.

Mathematical Formulation

$$ff'' + 2f''' = 0 \tag{5.20}$$

$$x = 0; \quad f(0) = 0; \quad f'(0) = 0 \tag{5.21a}$$

$$x = \infty; \quad f'(\infty) = 1 \tag{5.21b}$$

The solution, nondimensional velocity in the boundary layer, is obtained as

$$\frac{u}{U_\infty} = f'(x) \tag{5.22}$$

The solution to Equations (5.20) to (5.21) is largely through special iterative techniques. These techniques use a numerical integration procedure like the Runge-Kutta method to integrate the system by guessing and adjusting the missing boundary conditions at the initial point such that the final point boundary conditions are realized. This is due to the fact that the integration procedures usually solve *initial value* problems only. In the Blassius problem, this would imply the missing initial value can be regarded as a design variable.

$$f''(0) = \alpha \tag{5.23}$$

The objective function therefore will be

$$\text{Minimize:} \quad (f'(\infty) - 1)^2 \tag{5.24}$$

Implied in the formulation is that $f'(\infty)$ is obtained by integrating the differential equations (5.20) (these can be considered as differential constraints). There is one other consideration to take into account. The integration methods require the problem to be expressed in state space form, which requires an n^{th}-order differential equation to be expressed as a system of n first-order equations. The conversion is fairly standard and is done through transformation and introducing additional variables. For this example:

$$y_1 = f$$

$$y_1' = f' = y_2$$

$$y_2' = f'' = y_3$$

$$y_3' = f''' = -ff''/2$$

The optimization can be formulated as: Find α

$$\text{Minimize:} \quad [y_2(\infty) - 1]^2 \tag{5.25}$$

using the Runge-Kutta method (or any other method) to integrate

$$\begin{bmatrix} y_1 \\ y_2 \\ y_2 \end{bmatrix}' = \begin{bmatrix} y_2 \\ y_3 \\ -0.5y_1y_3 \end{bmatrix}; \quad \begin{bmatrix} y_1(0) \\ y_2(0) \\ y_3(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \alpha \end{bmatrix} \tag{5.26}$$

From published work [5] the value of $\alpha^* = 0.33206$. Instead of the final point at ∞ it is considered to be 5.0. Leaving this value as a variable can define a corresponding two-variable optimization problem. In the implementation to follow, the power and convenience of MATLAB are readily apparent.

Example5_3.m: This m-file is used to implement expression (5.25). It calls the MATLAB built-in function *ode45* to use the Runge-Kutta 4/5 order method. The function *ode45* requires the state equations to be available in another m-file. It also requires the definition of the interval of integration as well as the initial condition. The following line from the code describes the usage (the code is well commented too)

```
[t y ] = ode45('Ex5_3_state',tintval,bcinit);
```

t is the independent variable and *y* the dependent vector.

Ex5_3_state.m: The state equations in (5.26) are available in this file.

```
stst = [ y(2) , y(3) , -0.5*y(1)*y(3)]';
```

Refer to the code for details. **GoldSection_1Var.m** is used from the command line for this problem as

Usage: `GoldSection_1Var('Example5_3',0.001,0,0.1,10)`

Start iteration:

```
start
alphal(low)      alpha(1)        alpha(2)   alpha(up)
          0        0.1528          0.2472      0.4000
     1.0000        0.1904          0.0378      0.0161
```

The final iteration:

```
iteration 12
alphal(low)      alpha(1)        alpha(2)   alpha(up)
     0.3359        0.3364          0.3367      0.3371
     0.0000        0.0000          0.0000      0.0000
```


The result expected is $\alpha = 0.33206$. However, the tolerance specified was only 0.001. The initial bounding interval was only 0.4. The golden section has produced an impressive solution to the TPBVP. Also note the convenience of modular programming. No changes were made to **UpperBound_1Var.m** or **GoldSection_1Var.m** to run this example.

5.4.3 Example 5.4—Root Finding with Golden Section

The application of the golden section method to the root finding problem is important to accommodate equality and inequality constraints. Consider Example 5.4:

$$g(\alpha): 0.75\alpha^2 - 1.5\alpha - 1 = 0 \tag{5.27}$$

$$0 \le \alpha \le 4 \tag{5.2}$$

This is the inequality constraint of Example 5.1a except that it is transformed to an equality constraint. In this form the α necessary to make it into an active constraint can be established. The simplest way to accomplish this is to convert the problem to a minimization problem for which the numerical techniques have already been established. Squaring Equation (5.27) ensures that the minimum value would be zero (if it exists). Therefore the solution to Equation (5.27) is the same as

$$\text{Minimize: } f(\alpha): [0.75\alpha^2 - 1.5\alpha - 1]^2 \tag{5.28}$$

which can be handled by **GoldSection_1Var.m**. While this was quick and painless there are some attendant problems using this approach. First, the nonlinearity has increased. This is usually not encouraged in numerical investigations. Second, the number of solutions has satisfying FOC increased. Figure 5.5 represents the functions in Equations (5.27) and (5.28).

***Solution Using* GoldSection_1Var.m:** The following discussion is based on the experience of attempting to run the golden section method with the lower bound at the value of 0, the case with the other examples. There is a problem since the solution cannot be found. The problem appears to be in the calculation of the upper bound. In many of the numerical techniques there is an assumption that the objective function being dealt with is unimodal, that is, has a single minimum in the region of interest. For this to happen, it is assumed that the function being minimized will always decrease from the lower bound. Take a look at the function *f* in Figure 5.5. It is *increasing* at the value of zero. **UpperBound_1Var.m** assumes that the initial interval is too large and continues to bisect it to find a function value that is less than the one at the lower bound. This drives the interval to zero and the function is exited with the lower bound and with no solution.

Remedy 1: The easiest way out of the difficulty is to change the value of the lower bound and try again. It is useful to understand that the problem is with the calculation

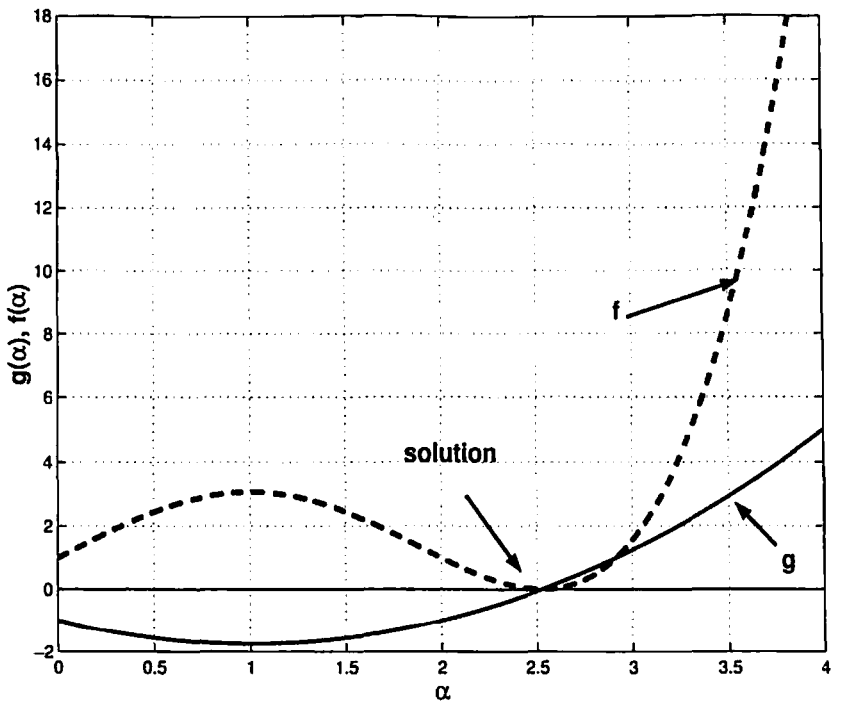

Figure 5.5 Example 5.4.

of the upper bound (bracketing the minimum) rather than with the golden section method. Changing the lower bound to 1 and rerunning the golden section method provides a successful result which matches the graphical solution. While success was realized, such a procedure needs user intervention as well as some knowledge about the problem—in this case there was the graphical solution. These are not acceptable qualities for an automatic procedure.

Remedy 2: Since the positive slope at the lower bound is indicative of the problem, it should be possible to sense this *positive sign of the slope* and move the lower bound to the point where the *slope is negative*. Such a point can be discovered by the same scanning procedure. The slope computation is numerical and is based on forward first difference. Once more this change in the lower bound is only needed for the upper bound calculation. **UpperBound_m1Var.m** is the modified upper bound calculation which relocates the lower bound to the point where the slope is negative. The usage and information from running the **GoldSection_1Var.m** incorporating the changed upper bound calculation are shown below. The same usage with the old **UpperBound_1Var.m** does not produce any useful results.

Usage: `GoldSection_1Var('Example5_4',0.001,0,.1,20)`

```
start
alphal(low)      alpha(1)        alpha(2)      alpha(up)
          0        0.9931          1.6069         2.6000
     1.0000        3.0624          2.1720         0.0289
iteration 16
alphal(low)      alpha(1)        alpha(2)      alpha(up)
     2.5266        2.5270          2.5273         2.5278
     0.0000        0.0000          0.0000         0.0000
```

REFERENCES

1. Burden, R. L., and Faires, J. D., *Numerical Analysis*, 4th ed., PWS_KENT Publishing Company, Boston, 1989.
2. Hosteller, G. H., Santina, M. S., and Montalou, P. D., *Analytical, Numerical, and Computational Methods for Science and Engineering*, Prentice–Hall, Englewood Cliffs, NJ, 1991.
3. Vanderplaats, G. N., *Numerical Optimization Techniques for Engineering Design*, McGraw–Hill, New York, 1984.
4. Huntley, H. E., *The Divine Proportion: A Study in Mathematical Beauty*, Dover Publications, New York, 1970.
5. Schlichting, H., *Boundary-Layer Theory*, McGraw–Hill, New York, 1979.

PROBLEMS

5.1 Extend Example 5.1 to include two inequality constraints for which a solution exists. Display the functions graphically and identify the results.

5.2 Extend Example 5.1 to include two inequality constraints for which there is no solution. Display the function graphically.

5.3 Reprogram the code **Sec5_2_2.m** to obtain the values of f, ϕ, and ϕ' through MATLAB function programs and apply to Example 5.1.

5.4 Reprogram the code **Sec5_2_2.m** to calculate numerical derivatives using first forward difference and first central difference schemes and apply to Example 5.1.

5.5 Identify the value of α where the constraint (5.3) becomes active using the Newton-Raphson technique.

5.6 Implement a bisection procedure for the minimization of the function of one variable. Apply it to solve Example 5.1.

5.7 For the polynomial approximation implementation, extend the code to display the original function and the approximating polynomial.

5.8 For the golden section implementation, extend the code to display the original function and the final external bounds on the variable.

5.9 Verify that the bounds on the variable are decreasing by the same ratio. Why is this better than the bisection method?

5.10 How would you set up the golden section method to determine the *zero* of the function instead of the minimum.

5.11 Combine the golden section and the polynomial approximation to find the minimum of the function.

5.12 Improve the accuracy of the solution to the Blassius solution over that presented in the text.

5.13 Obtain the solution to the Blassius equation using the polynomial approximation method.

6

NUMERICAL TECHNIQUES FOR UNCONSTRAINED OPTIMIZATION

This chapter illustrates many numerical techniques for multivariable unconstrained optimization. While unconstrained optimization is not a common occurrence in engineering design, nevertheless the numerical techniques included here demonstrate interesting ideas and also capture some of the early intensive work in the area of design optimization [1]. They also provide the means to solve constrained problems after they have been transformed into an unconstrained one (indirect methods—Chapter 7).

This book is unique for the presentation of nongradient techniques (Section 6.2) which can be empowered by the ubiquitous availability of incredible desktop computing power. Until very recently, application of design optimization required mainframe computers that relied on large programs written in FORTRAN. Today's desktop can run programs that require only limited programming skills. The desire for global optimization has brought into focus numerical techniques that are largely heuristic, have limited need for sophisticated gradient-based methods, require limited programming skills, and require limited programming resources. Usually such techniques require that the solution space be scanned using a large number of iterations. These methods can ideally be run on personal desktops for unlimited time. The pressure of limited computing resources is no longer a significant constraint on these techniques. While they may compromise on elegance and sophistication, it is important to recognize the opportunity to engage desktop resources for greater usefulness.

6.1 PROBLEM DEFINITION

The unconstrained optimization problem requires only the objective function. The book has chosen to emphasize an accompanying set of side constraints to restrict the

solution to an acceptable design space/region. Most numerical techniques in this chapter ignore the side constraints in laying out the method. It is usually the responsibility of the designer to verify the side constraints as part of his exploration of the optimum. This checking of the side constraint can be easily programmed in the code. Thus, the problem for this chapter can be defined as

$$\text{Minimize} \quad f(\mathbf{X}); \quad [\mathbf{X}]_n \tag{6.1}$$

$$\text{Subject to:} \quad x_i^l \le x_i \le x_i^u; \quad i = 1, 2, \ldots, n \tag{6.2}$$

6.1.1 Example 6.1

A two-variable problem is selected so that wherever possible graphical description can provide a better illustration of the algorithm.

$$\text{Minimize} \quad f(\mathbf{X}) = f(x_1, x_2) = 3 + (x_1 - 1.5x_2)^2 + (x_2 - 2)^2 \tag{6.3}$$

$$\text{Subject to:} \quad 0 \le x_1 \le 5; \quad 0 \le x_2 \le 5 \tag{6.4}$$

Figure 6.1 provides the contours of the problem and the solution can be located at $x_1^* = 3, x_2^* = 2$. The optimal value is $f^* = 3$. The problem is quadratic and the solution is also a *global* optimum.

6.1.2 Necessary and Sufficient Conditions

This section briefly reviews the necessary and sufficient conditions for unconstrained problems using Example 6.1. The FOC for this example (i.e., the Kuhn-Tucker conditions) are that the gradients must vanish at the solution

$$\frac{\partial f}{\partial x_1} = 2(x_1 - 1.5x_2) = 0 \tag{6.5a}$$

$$\frac{\partial f}{\partial x_2} = 2(-1.5)(x_1 - 1.5x_2) + 2(x_2 - 2) = 0 \tag{6.5b}$$

Equations (6.5) can be solved to obtain $x_1^* = 3, x_2^* = 2$. This matches the graphical solution. The sufficient conditions, also regarded as SOC, require that the Hessian matrix evaluated at the solution to FOC be positive definite. For this example the Hessian matrix is

$$[H] = \begin{bmatrix} 2 & -3 \\ -3 & 6.5 \end{bmatrix} \tag{6.6}$$

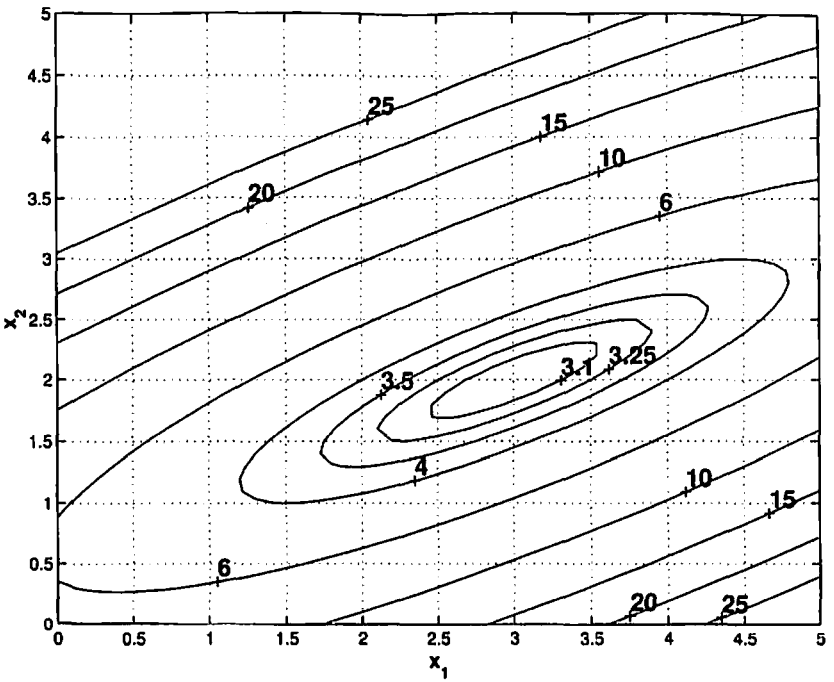

Figure 6.1 Example 6.1.

The Hessian is positive definite considering the determinants alone. **Sec6_1_1.m**[1] provides the code for the symbolic calculation and also displays the graphical description to the problem. The results of the calculation are displayed in the command window. The satisfaction of the SOC identifies that the solution for the design is a minimum.

6.1.3 Elements of a Numerical Technique

The elements of a typical numerical technique or algorithm, can be associated with the *generic algorithm* introduced during the discussion of the relevance of the one-dimensional optimization for multivariable problems (Chapter 5). This algorithm is iterative and is also referred to as a *search algorithm*, as the iterations take place by moving along a *search direction* or a *search vector* during an iteration. These search directions can be determined in many ways. The different techniques that are presented in this chapter primarily differ in how the search direction is established. The remaining elements of the algorithm, except for the convergence/stopping criteria, are the same for almost all of the methods. The algorithm can be expressed as follows.

[1]Files to be downloaded from the web site are indicated by boldface sans serif type.

General Algorithm (A6.1)

Step 1. Choose $\mathbf{X}_0$

Step 2. For each iteration i

Determine search direction vector $\mathbf{S}_i$

(It would be nice if the objective decreased along this direction)

Step 3. Calculate $\Delta\mathbf{X}_i = \alpha_i\mathbf{S}_i$

Note: $\Delta\mathbf{X}_i$ is now a function of the scalar α_i as $\mathbf{S}_i$ is known from Step 2

α_i is called the step size as it establishes the length of $\Delta\mathbf{X}_i$

α_i is determined by Minimizing $f(\mathbf{X}_{i+1})$, where $\mathbf{X}_{i+1} = \mathbf{X}_i + \Delta\mathbf{X}_i$

Here, the convergence criteria (FOC/SOC), and the stopping criteria (Is design changing? Exceeding iteration count? etc.) must be checked.

$i \leftarrow i + 1$;

Go To Step 2

In this chapter, this algorithmic structure will be used for all of the methods. The one-dimensional optimization will be implemented using the golden section method. Example 6.1 will be used to develop the different numerical methods.

6.2 NUMERICAL TECHNIQUES—NONGRADIENT METHODS

These methods are also called *zero-order* methods. The order refers to the order of the derivative of the objective function, needed to establish the search direction during any iteration. Zero order signifies that no derivatives are used [2], which in turn implies that only function values are used to establish the search vector. There is another important significance that accompanies the missing derivative computation—*the FOC should not be applied* in these methods. If we were computing derivatives we might as well include this information for establishing the search vector. Only the changes in the objective function or the design variables can provide convergence and/or stopping criteria.

Three techniques are included in this section. The first is a heuristic one based on random search directions. The second one cycles through a search direction based on each variable. The last is Powell's method which has the property of *quadratic convergence.*

6.2.1 Random Walk

The search direction during each iteration is a random direction. Random numbers and sets of random numbers are usually available through software. MATLAB includes the ability to generate several types of random numbers and matrices. Most computer-generated random numbers are called *pseudorandom numbers* as they cycle after a sufficient number of them have been created. The one-dimensional stepsize computation is done by the **GoldSection_nVar.m** function (*Chapter 5*).

Algorithm: Random Walk (A6.2)

Step 1. Choose $\mathbf{X}_0$, N (number of iterations)

Set $i = 1$

Step 2. For each iteration i

$\mathbf{S}_i$ = Random vector

Step 3. $\mathbf{X}_{i+1} = \mathbf{X}_i + \alpha_i\mathbf{S}_i$

α_i is determined by minimizing $f(\mathbf{X}_{i+1})$

$i \leftarrow i + 1$

If $i < N$ Go To Step 2

else Stop (iterations exceeded)

Prior to developing the code, see help on random numbers in MATLAB (>> help rand). Note that the random search direction vector created using *rand()* by MATLAB will always lie in the positive quadrant. Even though the design space is in the positive quadrant, the search for the next step should be in any direction. To allow for all directions, in the implementation A6.2, the elements of the search vector will switch sign based on a test of random number.

RandomWalk.m: The random walk algorithm (A6.2) is implemented in this m-file. This book strongly encourages translation of all algorithms into functional code components. The reader is strongly encouraged to examine the code as there are several that contain commented lines of information and not seen in the text. Some of the elements in the code will also be new. Graphical features and properties of MATLAB are exploited to describe the movement of the iterations in the design space. For its relative simplicity as well as total lack of sophistication, the random walk method functions incredibly well, sometimes approaching close to the solution in 20 iterations. This simplicity and tenacity is viewed as a significant advantage in global optimization. The code of the random walk function also includes the following features:

- For two-variable problems the contour of the objective is drawn.
- For two-variable problems the successful iterations are tracked on the contour plot and the iteration number identified.
- The multivariable golden section method (developed in Chapter 5) is used for the one-dimensional stepsize computation.
- The multivariable upper bound calculation (developed in Chapter 5) is used for scanning the upper bound.
- In the particular implementation here, those search directions that cause the function to increase are ignored.
- Before exit from the program the design variables and the functions for all of the iterations are printed.

Use of previous code illustrates code reuse and modular programming. Once more readers are strongly encouraged to visit the code to understand the translation of the algorithm and subsequent application.

Usage:

```
RandomWalk('Example6_1', [0.5 0.5],20,0.001,0,1,20)   (6.7)
```

Input: (They are the items enclosed within the outermost parenthesis in expression (6.7).)

'Example6_1':	The file **Example6_1.m** where the objective is described
[0.5 0.5]	starting design vector (dvar0)
20	number of iterations of the method (niter)
0.001	tolerance for the golden section procedure (tol)
0	initial stepsize for scanning upper bound (lowbound)
1	step interval for scanning (intvl)
20	number of scanning steps (ntrials)

Output:

A vector of $n + 1$ values of design variables at the last iteration and the value of the objective function.

Executing function (6.7) in the Command window produces the following information (the comments are added later):

```
  - The design vector and function during the iterations

ans =
  0.5000    0.5000    5.3125   % start
  0.5000    0.5000    5.3125   % did not move
  0.7091    1.0440    4.6481   % decreased function
  0.7091    1.0440    4.6481   % did not move
  0.7091    1.0440    4.6481   % did not move
  0.7091    1.0440    4.6481   % did not move
  0.6982    0.9756    4.6349   % decreased function
  0.6982    0.9756    4.6349   % etc ...
  2.9308    2.1093    3.0663
  3.0838    2.0481    3.0025
  3.0804    2.0350    3.0020
  3.0804    2.0350    3.0020
  3.0804    2.0350    3.0020
  3.0804    2.0350    3.0020
  3.0804    2.0350    3.0020
  3.0804    2.0350    3.0020
```



```
  3.0804    2.0350    3.0020
  3.0804    2.0350    3.0020
  3.0804    2.0350    3.0020
  3.0804    2.0350    3.0020   % 20 iterations
ans =
  3.0804    2.0350    3.0020
```

Compare these values to the actual solution of $x_1^* = 3, x_2^* = 2, f^* = 3$. This is remarkable for a method that has no sense of preferred direction. If the execution is repeated with identical input, an entire new solution will be obtained. Figure 6.2 tracks the changes of the design variables on the contour plot. The movement of the design variables are paired with the corresponding iteration number. The random walk function can be executed for several different starting sets of design variables. It should not disappoint. The **GoldSection_nVar.m** is the same one as in Chapter 5 except that the printing has been suppressed.

While executing the **RandomWalk.m** the printing of the iteration numbers creates a mess as you approach the solution. You can switch it off by commenting the text (. . .) statement. The code in **RandomWalk.m** also serves as a template for other algorithms. It is worth exploring in detail.

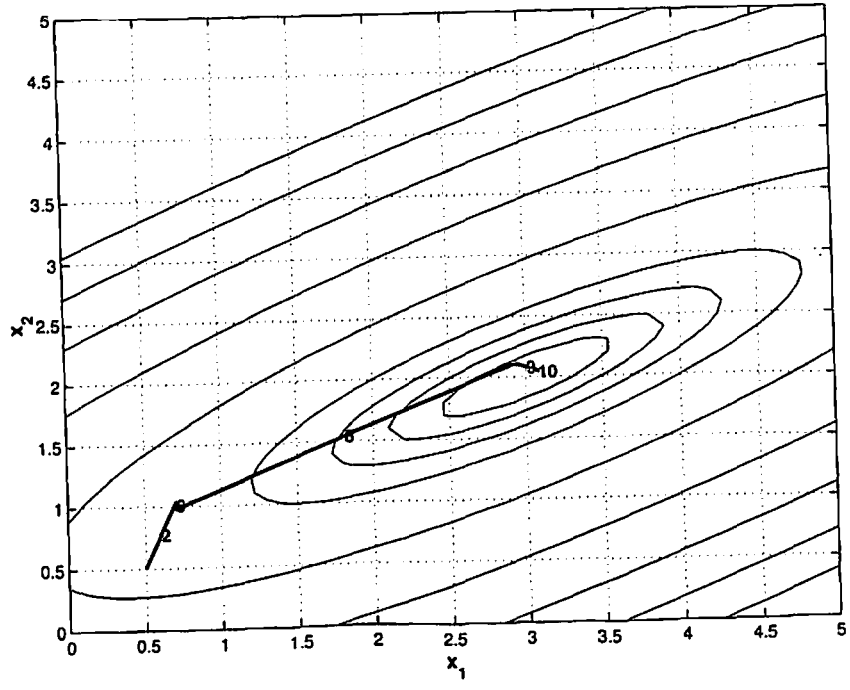

Figure 6.2 Random walk: Example 6.1.

6.2.2 Pattern Search

The Pattern Search method is a minor modification to the *Univariate* method with a major impact. As the Pattern Search uses the Univariate method the latter is not discussed separately. In the Univariate method, also known as the *Cyclic Coordinate Descent* method, each design variable (considered a coordinate) provides a search direction. This is also referred to as a coordinate direction, and is easily expressed through the unit vector for that coordinate. The search direction is cycled through the number of variables in sequence. Therefore, each cycle will have used n (number of design variables) iterations one for each of the n search directions. It can be shown by application that for problems with considerable nonlinearity, the Univariate method tends to get locked into a zigzag pattern of smaller and smaller moves as it approaches the solution.

The Pattern Search procedure attempts to disrupt this zigzag behavior by executing one additional iteration for each cycle. In each cycle, at the end of n Univariate directions, the $n + 1$ search direction is assembled as a linear combination of the previous n search directions and the optimum value of the stepsize for that direction. A one-dimensional optimal stepsize is then computed and the next cycle of iteration begins.

Algorithm: Pattern Search (A6.3)

Step 1. Choose $\mathbf{X}_1$, N_c (number of cycles)

$f(1) = f(\mathbf{X}_1)$; $\mathbf{X}_c(1) = \mathbf{X}_1$

ε_1, ε_2: tolerance for stopping criteria

Set $j = 1$ (initialize cycle count)

Step 2. For each cycle j

For $i = 1, n$

$\mathbf{S}_i = \hat{\mathbf{e}}_i$ (univariate step)

$\mathbf{X}_{i+1} = \mathbf{X}_i + \alpha_i \mathbf{S}_i$

α_i is determined by minimizing $f(\mathbf{X}_{i+1})$

(store values of α_i^* and $\mathbf{S}_i$)

end of For loop

$$\mathbf{S}_j = \sum_{i=1}^{n} \alpha_i^* \mathbf{S}_i \equiv \mathbf{X}_{n+1} - \mathbf{X}_1 \quad \text{(Pattern step)}$$

$\mathbf{X}_j = \mathbf{X}_{n+1} + \alpha_j \mathbf{S}_j$

$\mathbf{X}_c(j + 1) \leftarrow \mathbf{X}_j$; $f_c(j + 1) = f(\mathbf{X}_j)$ (store cycle values)

Step 3. $\Delta f = f_c(j + 1) - f_c(j)$; $\Delta \mathbf{X} = \mathbf{X}_c(j + 1) - \mathbf{X}_c(j)$

If $|\Delta f| \le \varepsilon_1$, stop

If $\Delta \mathbf{X}^T \Delta \mathbf{X} \le \varepsilon_2$, stop

If $j = N_c$, stop

$\mathbf{X}_1 \leftarrow \mathbf{X}_j$

$j \leftarrow j + 1$

Go To Step 2

The Pattern Search method has a few additional stopping criteria. These are based on how much the function is decreasing for each cycle and how much change is taking place in the design variable itself. For the latter this information is based on the length of the change in the design vector for the cycle. Once again the approach is simple and the implementation is direct. Programming such a technique will not be too forbidding.

PatternSearch.m: To continue the exposure to MATLAB programming this method is implemented very differently from the other algorithms that have been translated into code so far. It is a stand-alone program. There are prompts for user inputs with appropriate default values in case the user decides not to enter a value. The one-dimensional stepsize computation is implemented by the multivariable golden section method. The interesting features are as follows.

- It is a freestanding program.
- The file containing the objective function is selected through a list box.
- Several program control parameters are entered from the keyboard after a suitable prompt indicating default values.
- Default values are initialized if the user chooses not to enter the values.
- The design vector is tracked and printed after the program is stopped.
- Since the upper bound calculations are based on positive values of stepsize, the search direction is reversed if the returned stepsize is the same as the lower bound.
- Iteration counts are recorded.
- For two-variable problem the objective function contours are plotted.
- The first nine or less iterations are traced on the contour plot. The Univariate and the pattern steps are color coded.

Usage: `PatternSearch` (6.8)

The problem is Example 6.1. The file is **Example 6_1.m.**

Output (MATLAB Command window): With the default values and the starting vector [4 3] the following are written to the Command window.

```
>> PatternSearch
The function for which the minimum is sought must be a
MATLAB function M - File. Given a vector dependent variable
it must return a scalar value. This is the function to be
MINIMIZED. Please select function name in the dialog box
and hit return:
```

```
    The function you have chosen is ::Example6_1
    maximum number of cycles [1000]:
    convergence tolerance for difference in f[1e-8]:
    convergence tolerance on change in design x [1e-8]:
Input the starting design vector. This is mandatory as
there is no default vector setup. The length of your
vector indicates the number of unknowns. Please enter it
now and hit return:
  [4 3]

    The initial design vector [4.00 3.00]
  Convergence in f: 3.999E-009 reached in 10 iterations
  Number of useful calls to the Golden Section Search
Method: 31
  Total number of calls to the Golden Section Search
Method: 40

  The values for x and f are

 4.0000    3.0000    4.2500    0
 4.4992    3.0000    4.0000    0.4996
 4.4992    2.6924    3.6916    0.3081
 4.3868    2.7616    3.6398    0.2254
 4.1432    2.7616    3.5801    0.2446
 4.1432    2.5287    3.4022    0.2339
 3.8229    2.2224    3.2890    1.3148
 3.3338    2.2224    3.0494    0.4896
 3.3338    2.1543    3.0343    0.0681
 3.1750    2.1321    3.0180    0.3248
 3.1980    2.1321    3.0175    0.0234
 3.1980    2.0916    3.0121    0.0405
 3.1944    2.0980    3.0118    0.1579
 3.1473    2.0980    3.0096    0.0471
 3.1473    2.0682    3.0067    0.0298
 3.0346    1.9969    3.0016    2.3916
 2.9958    1.9969    3.0000    0.0391
 2.9958    1.9981    3.0000    0.0013
 2.9971    1.9981    3.0000    0.0322
 2.9972    1.9981    3.0000    0.0001
 2.9972    1.9986    3.0000    0.0005
 2.9972    1.9988    3.0000    0.4563
 2.9977    1.9988    3.0000    0.0005
 2.9977    1.9990    3.0000    0.0002
 2.9999    1.9999    3.0000    4.9305
 2.9998    1.9999    3.0000    0.0002
```



```
 2.9998    1.9999    3.0000    0.0000
 2.9998    1.9999    3.0000    0.3398
 2.9998    1.9999    3.0000    0.0000
 2.9998    1.9999    3.0000    0.0000
 2.9998    1.9999    3.0000    0.2020
```

To verify that the program is executing the algorithm, scan the design vectors in the first two columns. The univariate steps can be recognized when one component of the vector changes while the other maintains its value. The pattern directions can be recognized when both components change.

The number of iterations is a significant amount. One reason is that the tolerances are much smaller. It can be noted that the difference between two iterations changes linearly. Such methods are expected to run for a large number of iterations. The program control parameters control the accuracy of the results as well as the number of iterations. The solution is more accurate than the Random Walk method.

Figure 6.3 illustrates the algorithm graphically. The univariate and the pattern directions are quite distinguishable. When executed on your machine these steps are explicitly color coded. Readers are encouraged to go over the code, and modify it to

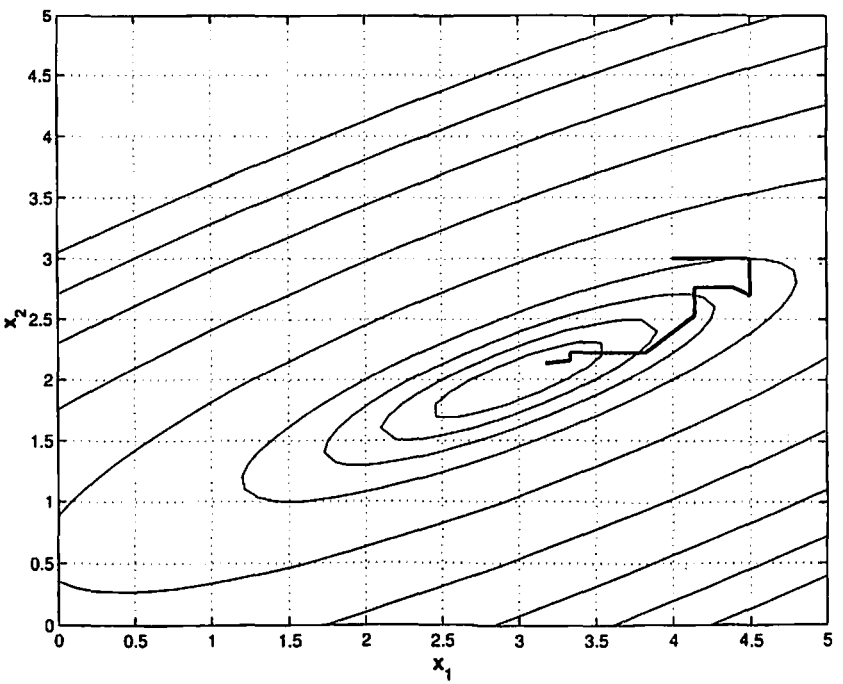

Figure 6.3 Pattern search: Example 6.1.

reflect their preference. The number of lines is around 250. We are starting to get into serious programming.

6.2.3 Powell's Method

If there were only one zero-order method that must be programmed, the overwhelming choice would be Powell's method [3]. The principal reason for the decision would be that it has the property of *quadratic convergence*, namely, for a quadratic problem with n variables convergence will be achieved in less than or equal to n Powell cycles. A *quadratic problem* is an unconstrained minimization of a function that is expressed as a *quadratic polynomial*—a polynomial with no term having a degree greater than two. Example 6.1 is an example of a quadratic polynomial in two variables. Engineering design optimization problems are rarely described by a quadratic polynomial. This does not imply that you cannot use Powell's method. What this means is that the solution should not be expected to converge quadratically. For nonquadratic problems, as the solution is approached iteratively, the objective can be approximated very well by a quadratic function. It is at this stage that the quadratic convergence property is realized in the computations.

The actual algorithm (A6.4) is a simple (the word is not being used lightly) modification to the Pattern Search algorithm. In each cycle (after the first), instead of using univariate directions in the first n iterations, the search directions are obtained from the previous cycle. The new search directions are obtained by *left shifting* the directions of the previous cycle. In this way a history of the previous search directions is used to establish the next. With this change, Powell's method should converge at most in three cycles for Example 6.1 rather than the several listed in the previous section.

Algorithm: Powell's Method (A6.4)

Step 1. Choose $\mathbf{X}_1$, N_c (number of cycles)
$f(1) = f(\mathbf{X}_1)$; $\mathbf{X}_c(1) = \mathbf{X}_1$
$\varepsilon_1, \varepsilon_2$: tolerance for stopping criteria
Set $j = 1$ (initialize Powell cycle count)
For $i = 1, n$
$\mathbf{S}_i = \hat{\mathbf{e}}_i$ (univariate step)

Step 2. For each cycle j
For $i = 1, n$
If $j \geq 2$, $\mathbf{S}_i \leftarrow \mathbf{S}_{i+1}$
$\mathbf{X}_{i+1} = \mathbf{X}_i + \alpha_i \mathbf{S}_i$
α_i is determined by minimizing $f(\mathbf{X}_{i+1})$
end of For loop

$$\mathbf{S}_j^p = \mathbf{S}_{i+1} = \sum_{i=1}^{n} \alpha_i^* \mathbf{S}_i = \mathbf{X}_{n+1} - \mathbf{X}_1 \qquad \text{(Pattern step)}$$

$\mathbf{X}_j^p = \mathbf{X}_{n+1} + \alpha_j \mathbf{S}_j$
$\mathbf{X}_c(j+1) \leftarrow \mathbf{X}_j^p$; $f_c(j+1) = f(\mathbf{X}_j^p)$ (store cycle values)

Step 3. $\Delta f = f_c(j+1) - f_c(j)$; $\Delta \mathbf{X} = \mathbf{X}_c(j+1) - \mathbf{X}_c(j)$
If $|\Delta f| \leq \varepsilon_1$, stop
If $\Delta \mathbf{X}^T \Delta \mathbf{X} \leq \varepsilon_2$, stop
If $j = N_c$, stop
$\mathbf{X}_1 \leftarrow \mathbf{X}_j$
$j \leftarrow j + 1$
Go To Step 2

Application of Powell's Method: The translation of the algorithm into MATLAB code is left as an exercise for the student. Step-by-step application to Example 6.1 is shown here. Example 6.1 is restated as

Example 6.1

$$\text{Minimize} \quad f(\mathbf{X}) = f(x_1, x_2) = 3 + (x_1 - 1.5x_2)^2 + (x_2 - 2)^2 \tag{6.3}$$

$$\text{Subject to:} \quad 0 \leq x_1 \leq 5; \quad 0 \leq x_2 \leq 5 \tag{6.4}$$

Step 1. $\mathbf{X}_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$; $f(\mathbf{X}_1) = 5.3125$; $\varepsilon_1 = 1.0E-08$; $\varepsilon_2 = 1.0E-08$

$\mathbf{S}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$; $\mathbf{S}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

$j = 1$

Step 2. $j = 1$

$i = 1$: $\mathbf{X}_2 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} + \alpha_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 + \alpha_1 \\ 0.5 \end{bmatrix}$

$f(\alpha_1) = 21/4 + (-1/4 + \alpha_1)^2$; $\alpha_1^* = 1/4$; $\mathbf{X}_2 = \begin{bmatrix} 0.75 \\ 0.5 \end{bmatrix}$

$f(\mathbf{X}_2) = 5.25$

$i = 2$: $\mathbf{X}_3 = \begin{bmatrix} 0.75 \\ 0.5 \end{bmatrix} + \alpha_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.75 \\ 0.5 + \alpha_2 \end{bmatrix}$

$\alpha_2^* = 0.4615$; $\mathbf{X}_3 = \begin{bmatrix} 0.75 \\ 0.9615 \end{bmatrix}$; $f(\mathbf{X}_3) = 4.5576$

$i = 3$: $\mathbf{S}_3 = 0.25 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + (0.4615) \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.25 \\ 0.4615 \end{bmatrix}$

$\mathbf{X}_4 = \begin{bmatrix} 0.75 \\ 0.9615 \end{bmatrix} + \alpha_3 \begin{bmatrix} 0.25 \\ 0.4615 \end{bmatrix}$

$$\alpha_3^* = 0.4235; \quad \mathbf{X}_4 = \begin{bmatrix} 0.8558 \\ 1.1570 \end{bmatrix}; \quad f(\mathbf{X}_4) = 4.4843$$

Step 3. $|\Delta f| = 1.353 > \varepsilon_1$, continue

$|\Delta \mathbf{X}^T \Delta \mathbf{X}| = 1.8992 > \varepsilon_2$, continue

$j = 2$: One cycle over

Go To Step 2

Step 2: $j = 2$

$i = 1$:

$$\mathbf{X}_1 = \begin{bmatrix} 0.8558 \\ 1.1570 \end{bmatrix}; \quad \mathbf{S}_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}; \quad \alpha_1^* = -0.1466; \quad \mathbf{X}_2 = \begin{bmatrix} 0.8558 \\ 1.0104 \end{bmatrix}; \quad f(\mathbf{X}_2) = 4.4145$$

$i = 2$:

$$\mathbf{X}_2 = \begin{bmatrix} 0.8558 \\ 1.0104 \end{bmatrix}; \quad \mathbf{S}_2 = \begin{bmatrix} 0.25 \\ 0.4615 \end{bmatrix}; \quad \alpha_2^* = 0.4035; \quad \mathbf{X}_3 = \begin{bmatrix} 0.9567 \\ 1.1966 \end{bmatrix}; \quad f(\mathbf{X}_3) = 4.3479$$

$i = 3$:

$$\mathbf{X}_3 = \begin{bmatrix} 0.9567 \\ 1.1966 \end{bmatrix}; \quad \mathbf{S}_3 = \begin{bmatrix} 0.1009 \\ 0.0396 \end{bmatrix}; \quad \alpha_3^* = 20.25; \quad \mathbf{X}_4 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}; \quad f(\mathbf{X}_4) = 3$$

The analytical solution is available and the computations in this exercise are stopped as the solution has been obtained.

If the algorithm was coded as written, would it stop here? You are encouraged to think about it. Assessing the calculations.

- The method only took two cycles to converge (as expected).
- The extent of computation is the same as the Pattern Search but the number of iterations is significantly smaller.
- The stepsize for the first iteration in the second cycle is negative. Hence, reversing the search direction is necessary.
- The stepsize for the last iteration is 20.25. The supporting algorithm for scanning and golden section must be able to trap the minimum at these large values.
- While not illustrated here, the maximum number of cycles to convergence for a quadratic problem is independent of the starting point. You can verify this through the application of your code.

Conjugacy: The formal reason for the quadratic convergence property for the Powell method is that the search directions at the $n + 1$ iterations are *conjugate with respect to the Hessian matrix.* For a quadratic problem the Hessian matrix is a constant matrix. If search directions S_i and S_j are conjugate directions, then the conditions for conjugacy are

$$S_i^T [\mathbf{H}] S_j = 0 \tag{6.9}$$

Verify if it is true for the example.

6.3 NUMERICAL TECHNIQUES—GRADIENT-BASED METHODS

In light of the previous definition, these would be referred to as *first-order* methods. The search directions will be constructed using the gradient of the objective function. Since gradients are being computed, the Kuhn-Tucker conditions (FOC) for unconstrained problems, $\nabla f = 0$, can be used to check for convergence. The SOC are hardly ever applied. One of the reasons is that it would involve the computation of an $n \times n$ second derivative matrix which is considered computationally expensive, particularly if the evaluation of the objective function requires a call to a finite element method for generating required information. Another reason for not calculating the Hessian is that the existence of the second derivative in a real design problem is not certain even though it is computationally possible or feasible. For problems that can be described by symbolic calculations, MATLAB should be able to handle computation of second derivative at the possible solution and its eigenvalues.

Without SOC these methods require user's vigilance to ensure that the solution obtained is a minimum rather than a maximum or a saddle point. A simple way to verify this is to perturb the objective function through perturbation in the design variables at the solution and verify it is a local minimum. This brings up an important property of these methods—*they only find local optimums*. Usually this will be close to the design where the iterations are begun. Before concluding the design exploration, it is necessary to execute the method from several starting points to discover if other minimums exist and select the best one by head to head comparison. The bulk of existing unconstrained and constrained optimization methods belong to this category.

Four methods are presented. The first is the *Steepest Descent* method. While this method is not used in practice, it provides an excellent example for understanding the algorithmic principles for the gradient-based techniques. The second is the *Conjugate Gradient* technique which is a classical workhorse particularly in industry usage. The third and fourth belong to the category of *Variable Metric* methods, or *Quasi-Newton* methods as they are also called. These methods have been popular for some time, and will possibly stay that way for a long time to come.

The general problem and specific example are reproduced for convenience

$$\text{Minimize} \quad f(X); \quad [X]_n \tag{6.1}$$

$$\text{Subject to} \quad x_i^l \le x_i \le x_i^u; \quad i = 1, 2, \ldots, n \tag{6.2}$$

$$\text{Minimize} \quad f(X) = f(x_1, x_2) = 3 + (x_1 - 1.5x_2)^2 + (x_2 - 2)^2 \tag{6.3}$$

$$\text{Subject to:} \quad 0 \le x_1 \le 5; \quad 0 \le x_2 \le 5 \tag{6.4}$$

6.3.1 Steepest Descent Method

This method provides a natural evolution for the gradient based techniques [4]. The gradient of a function at a point is the direction of the most rapid increase in

the value of the function at that point. The descent direction can be obtained reversing the gradient (or multiplying it by -1). The next step would be to regard the descent vector as a search direction, after all we are attempting to decrease the function through successive iterations. This series of steps give rise to the Steepest Descent algorithm.

Algorithm: Steepest Descent (A6.5)

Step 1. Choose $\mathbf{X}_1$, N (number of iterations)
$f_s(1) = f(\mathbf{X}_1)$; $\mathbf{X}_s(1) = \mathbf{X}_1$ (store values)
$\varepsilon_1, \varepsilon_2, \varepsilon_3$: (tolerance for stopping criteria)
Set $i = 1$ (initialize iteration counter)

Step 2. $\mathbf{S}_i = -\nabla f(\mathbf{X}_i)$ (this is computed in Step 3)
$\mathbf{X}_{i+1} = \mathbf{X}_i + \alpha_i \mathbf{S}_i$
α_i is determined by minimizing $f(\mathbf{X}_{i+1})$
$\mathbf{X}_s(i+1) \leftarrow \mathbf{X}_{i+1}$; $f_s(i+1) = f(\mathbf{X}_{i+1})$ (store values)

Step 3. $\Delta f = f_s(i+1) - f_s(i)$; $\Delta \mathbf{X} = \mathbf{X}_s(i+1) - \mathbf{X}_s(i)$
If $|\Delta f| \le \varepsilon_1$; stop (function not changing)
If $\Delta \mathbf{X}^T \Delta \mathbf{X} \le \varepsilon_2$; stop (design not changing)
If $i + 1 = N$; stop
If $\nabla f(\mathbf{X}_{i+1})^T \nabla f(\mathbf{X}_{i+1}) \le \varepsilon_3$; converged
$i \leftarrow i + 1$
Go To Step 2

SteepestDescent.m: This is an m-file that executes algorithm A6.5. It uses the golden section and the upper bound scanning process. The features of this program are as follows.

- For two variables it will draw the contour plot.
- For two variables the design vector changes can be seen graphically in slow motion with steps in different color.
- The design variables, the function value, and the square of the length of the gradient vector (called *KT value*) at each iteration are displayed in the Command window at completion of the number of iterations.
- The gradient of the function is numerically computed using first forward finite difference. The gradient computation is therefore automatic.
- From a programming perspective, basic string handling to change line color is introduced.

Usage:
```
SteepestDescent('Example6_1',[0.5 0.5],
20,0.0001,0,1,20)
```
(6.10)

Output (written to the Command window): A couple of iterations at the start and at the end (out of 20 iterations) are copied below.

0.5000	0.5000	5.3125	5.2969
0.5996	0.9483	4.7832	2.8411
1.0701	0.8429	4.3766	3.1379
⋮			
2.6901	1.8659	3.0298	0.0507
2.7455	1.8502	3.0233	0.0462
2.7585	1.8960	3.0181	0.0316

Figure 6.4 represents the graphical motion of the design variables for 20 iterations. The graphical picture and the changes in the design are not impressive. The Steepest Descent method is woefully inadequate compared to Powell's method even if the latter is a zero-order method. Moreover, this conclusion is drawn with respect to an easy problem (quadratic) Example 6.1. This performance justifies the lack of serious interest in the Steepest Descent method.

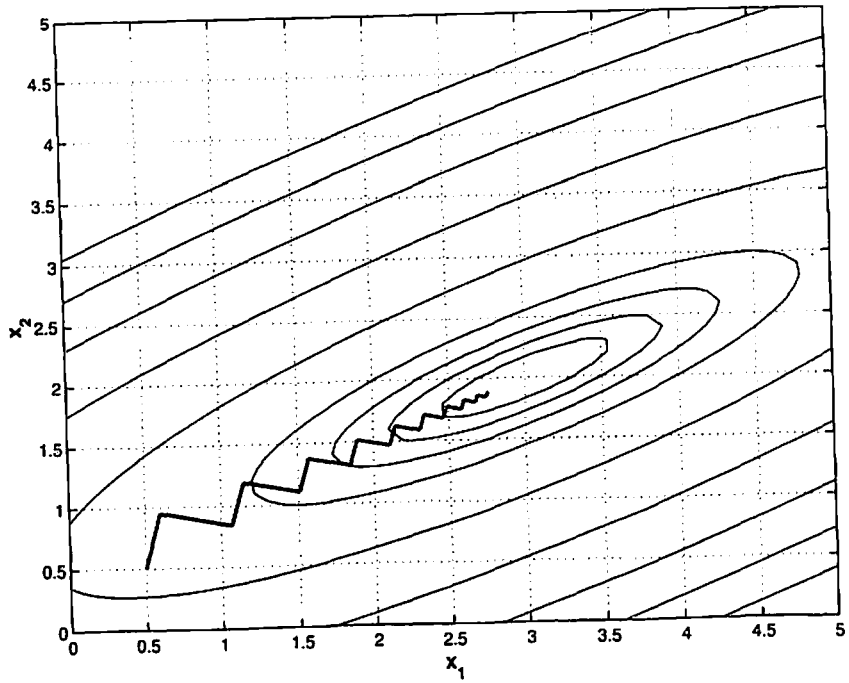

Figure 6.4 Steepest descent: Example 6.1.

A close observation of Figure 6.4 illustrates the zigzag motion—which was referred to earlier with respect to the Univariate method. Good methods are expected to overcome this pattern. In the case of the Univariate method this was achieved through Pattern Search method in the zero-order family. An iteration breaking out of the zigzag pattern (or preventing getting locked into one) is necessary to improve the method. Such a feature is implemented in the remaining methods of this section.

6.3.2 Conjugate Gradient (Fletcher-Reeves) Method

The Conjugate Gradient method, originally due to Fletcher and Reeves [5], is a small modification to the Steepest Descent method with an enormous effect on performance. This improvement mirrors the achievement in the transformation of the Pattern Search to Powell's method. The most impressive gain is the property of quadratic convergence because the search directions are *conjugate* with respect to the Hessian matrix at the solution. A quadratic problem in n variables will converge in no more than n iterations.

Algorithm: Conjugate Gradient (A6.6)

Step 1. Choose $\mathbf{X}_1$, N (number of iterations)

$f_s(1) = f(\mathbf{X}_1)$; $\mathbf{X}_s(1) = \mathbf{X}_1$(store values)

$\varepsilon_1, \varepsilon_2, \varepsilon_3$: (tolerance for stopping criteria)

Set $i = 1$ (initialize iteration counter)

Step 2. If $i = 1$, $\mathbf{S}_i = -\nabla f(\mathbf{X}_i)$

Else $$\beta = \frac{\nabla f(\mathbf{X}_i)^T \nabla f(\mathbf{X}_i)}{\nabla f(\mathbf{X}_{i-1})^T \nabla f(\mathbf{X}_{i-1})}$$

$\mathbf{S}_i = -\nabla f(\mathbf{X}_i) + \beta \mathbf{S}_{i-1}$

$\mathbf{X}_{i+1} = \mathbf{X}_i + \alpha_i \mathbf{S}_i$

α_i is determined by minimizing $f(\mathbf{X}_{i+1})$

$\mathbf{X}_s(i+1) \leftarrow \mathbf{X}_{i+1}$; $f_s(i+1) = f(\mathbf{X}_{i+1})$ % (store values)

Step 3. $\Delta f = f_s(i+1) - f_s(i)$; $\Delta \mathbf{X} = \mathbf{X}_s(i+1) - \mathbf{X}_s(i)$

If $|\Delta f| \le \varepsilon_1$; stop (function not changing)

If $\Delta \mathbf{X}^T \Delta \mathbf{X} \le \varepsilon_2$; stop (design not changing)

If $i + 1 = N$; stop

If $\nabla f(\mathbf{X}_{i+1})^T \nabla f(\mathbf{X}_{i+1}) \le \varepsilon_3$; converged

$i \leftarrow i + 1$

Go To Step 2

A comparison between the algorithms for the Steepest Descent and the Conjugate Gradient method reveals that the latter carries one additional computation—β, and the successive adjustment of the search direction incorporating this value. β represents the ratio of the square of the current gradient vector to the square of the previous gradient vector. The first thing to notice is that a degree of robustness is built into the method by carrying information from the previous iteration. This is like maintaining a history of the method albeit for just one iteration. Since the FOC is based on the length of the gradient approaching zero at the solution, this particular form of incorporation of β is ingenious. If the previous iteration is close to the solution, then β is large and the previous iteration plays a significant role in the current iteration. On the other hand, if β is large, suggesting the solution is still far away, then the current value of the gradient determines the new search direction.

Application of Conjugate Gradient Method: This method is important to warrant working through the iterations. These calculations, similar to Powell's method, can be done using a hand calculator, using a spreadsheet, or using MATLAB itself. The author recommends the calculator as the process is slow enough to develop an intuition working with the pattern of numbers. In the following, and unlike Powell's method, only the relevant calculations are recorded. The reader should follow along verifying the numbers are indeed correct.

Step 1. $\mathbf{X}_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$; $f(\mathbf{X}_1) = 5.3125$

Step 2. $\mathbf{S}_1 = \begin{bmatrix} 0.5 \\ 2.25 \end{bmatrix}$; $\alpha_1^* = 0.1993$; $f(\alpha_1^*) = 4.7831$; $\mathbf{X}_2 = \begin{bmatrix} 0.5996 \\ 0.9484 \end{bmatrix}$

$i = 2$

Step 2: $\beta = 0.5351$; $\nabla f(\mathbf{X}_2) = \begin{bmatrix} -1.6459 \\ 0.3657 \end{bmatrix}$

$\mathbf{S}_2 = \begin{bmatrix} 1.9135 \\ 0.8383 \end{bmatrix}$; $\alpha_2^* = 1.2544$; $f(\alpha_2^*) = 3.0$; $\mathbf{X}_3 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$

ConjugateGradient.m: This m-file will execute the Conjugate Gradient method. For two-variable problems there will be a contour plot over which the iterative progress in the design space is recorded. The property of quadratic convergence can be observed as the solution is obtained after two iterations. This is definitely impressive compared with the Steepest Descent method. Keep in mind that the modification to the algorithm is minor. As the various codes are run in order, some changes in the figure can be observed. These are largely due to the use of some string processing functions. You are encouraged to see how they are implemented. The author is certain there are things you would do differently. Do not hesitate to try, as that is the only way to learn to use MATLAB effectively.

Usage:
```
ConjugateGradient('Example6_1',[0.5 0.5],
20,0.0001,0,1,20);
```
(6.11)

Output (Command window)
The problem: Example6_1
The design vector, function value and KT value during the iterations

0.5000	0.5000	5.3125	0.2490
0.5996	0.9483	4.7832	2.8411
2.9995	1.9982	3.0000	0.0001

Keep in mind the solution is dependent on the tolerance for the one-dimensional stepsize search and that the derivatives are computed numerically. The execution of the code should display Figure 6.5 in the figure window.

6.3.3 Davidon-Fletcher-Powell Method

The Davidon-Fletcher-Powell (DFP) [6] method belongs to the family of *Variable Metric Methods* (VMM). It was first introduced by Davidon and several years later was developed in its current form by Fletcher and Powell. Being aware of the Conjugate Gradient method, these methods would not merit inclusion in this book if they did not have the property of *quadratic convergence*, which they do. Generally

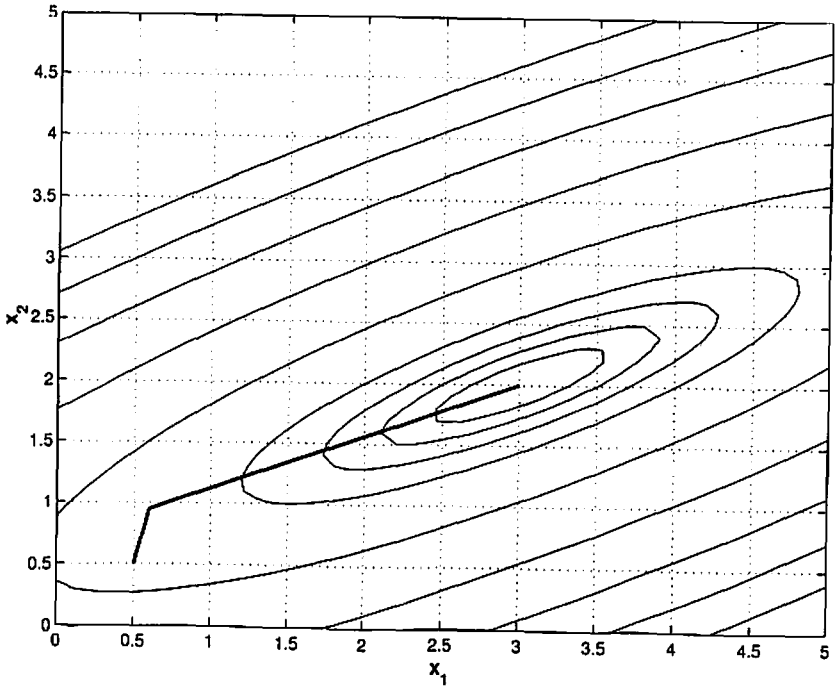

Figure 6.5 Conjugate gradient method: Example 6.1.

this family of methods go beyond that. As the solution is approached, they behave like Newton's method (Section 6.4). The quadratic convergence of Newton's method is quite impressive—it will locate the solution to a quadratic problem in *one* iteration. This Newton-like behavior of the VMM, as the solution is approached, has given them another name termed *Quasi-Newton* or *Newton-like methods.*

The DFP method is presented here for historical interest and because it is a little easier to understand than the others. In fact, Huang [7] documents a generic procedure from which most of the popular methods can be obtained, and from which you could also derive your own.

The Conjugate Gradient method's improvement over the Steepest Descent method was possible because of the inclusion of the history from the previous iteration. In the quasi-Newton methods the history from all previous iterations is available. This information is collected in an $n \times n$ matrix called the *metric*. The metric is updated with each iteration and is used to establish the search direction. An initial choice for the metric is also required. It must be a *symmetric positive definite* matrix. For the method to converge, the metric must hold on to its positive definite property through the iterations. In the DFP method, the metric approaches the inverse of the Hessian at the solution.

Algorithm: Davidon-Fletcher-Powell (A6.7)

Step 1. Choose $\mathbf{X}_1$, $[\mathbf{A}_1]$ (initial metric), N
$\varepsilon_1, \varepsilon_2, \varepsilon_3$: (tolerance for stopping criteria)
Set $i = 1$ (initialize iteration counter)

Step 2. $\mathbf{S}_i = -[\mathbf{A}_i]\nabla f(\mathbf{X}_i)$
$\mathbf{X}_{i+1} = \mathbf{X}_i + \alpha_i \mathbf{S}_i$; $\Delta \mathbf{X} = \alpha_i \mathbf{S}_i$
α_i is determined by minimizing $f(X_{i+1})$

Step 3. If $\nabla f(X_{i+1})^T \nabla f(X_{i+1}) \le \varepsilon_3$; converged
If $|f(X_{i+1}) - f(X_i)| \le \varepsilon_1$; stop (function not changing)
If $\Delta \mathbf{X}^T \Delta \mathbf{X} \le \varepsilon_2$, stop (design not changing)
If $i + 1 = N$, stop (iteration limit)
Else
$\mathbf{Y} = \nabla f(\mathbf{X}_{i+1}) - \nabla f(\mathbf{X}_i)$
$\mathbf{Z} = [\mathbf{A}_i]\mathbf{Y}$

$$[\mathbf{B}] = \frac{\Delta \mathbf{X}\, \Delta \mathbf{X}^T}{\Delta \mathbf{X}^T \mathbf{Y}}$$

$$[\mathbf{C}] = -\frac{\mathbf{Z}\mathbf{Z}^T}{\mathbf{Y}^T \mathbf{Z}}$$

$[\mathbf{A}_{i+1}] = [\mathbf{A}_i] + [\mathbf{B}] + [\mathbf{C}]$
$i \leftarrow i + 1$
Go To Step 2

In the above the matrices are enclosed by square brackets. The initial choice of the metric is a positive definite matrix. The *identity* matrix is a safe choice.

DFP.m: The DFP algorithm (A6.7) is coded in this m-file. Similar to the previous programs, for two variables the design changes are tracked on a background contour plot, Figure 6.6 . The default initial metric is the identity matrix generated using a MATLAB built-in function.

Usage: `DFP('Example6_1',[0.5 0.5],4,0.0001,0,1,20);` (6.12)

Output: The output from the above invocation (copied from the Command window) can be obtained by removing the semi-colon after the statement. The variable definitions match those defined in the algorithm. For Example 6.1

```
A =
 1          0
 0          1
```

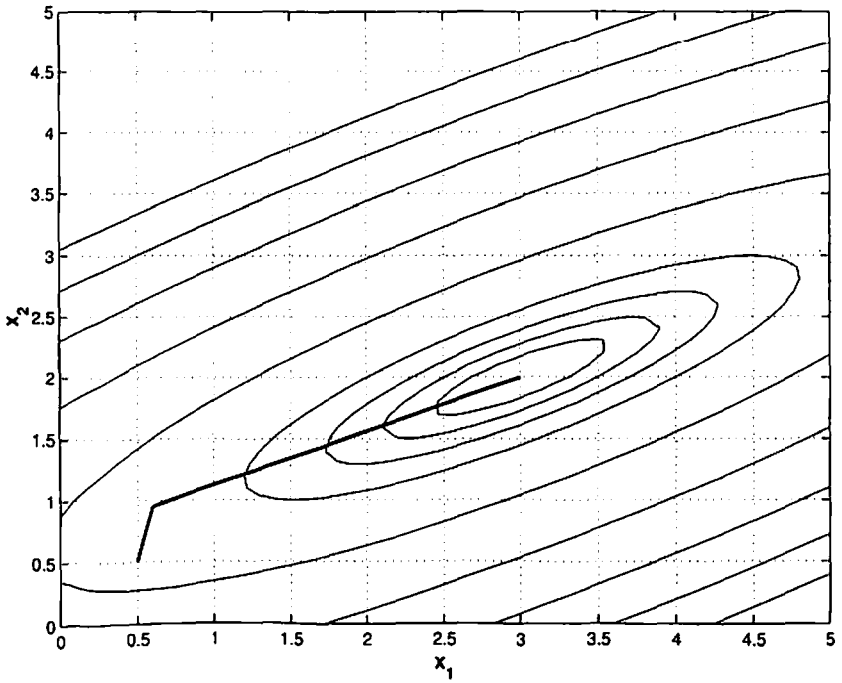

Figure 6.6 DFP Method: Example 6.1.


```
iteration number: 1
s =
  0.4990  2.2467
delx =
  0.0996
  0.4483
Y =
  -1.1458
   2.6153
Z =
  -1.1458
   2.6153
B =
  0.0094  0.0422
  0.0422  0.1899
C =
  -0.1610  0.3676
   0.3676 -0.8390
A =
  0.8483  0.4097
  0.4097  0.3509
iteration number: 2
s =
  1.2443  0.5446
The problem: Example6_1
The design vector, function value and KT value
during the iterations
 0.5000   0.5000      5.3125   5.2969
 0.5996   0.9483      4.7832   2.8411
 2.9996   1.9987      3.0000   0.0000
```

6.3.4 Broydon-Fletcher-Goldfarb-Shanno Method

If you were to program only one gradient-based method, then the Broydon-Fletcher-Goldfarb-Shanno (BFGS) [8] method would be the one. It is a quasi-Newton method and currently is the most popular of the Variable Metric methods. It enjoys the property of quadratic convergence and has robustness by carrying forward information from the previous iterations. The difference between the DFP and BFGS is the way the metric is updated. The former converges to the inverse of the Hessian, while the latter converges to the Hessian itself. In a sense the BFGS is more direct. The BFGS has replaced the Conjugate Gradient techniques as a workhorse in solving nonlinear equations.

For convergence the metric must be positive definite. An initial choice of positive definite matrix for the metric is usually sufficient to ensure this property for quadratic problems. The identity matrix is usually a default choice.

Algorithm: Broyden-Fletcher-Goldfarb-Shanno (BFGS) Method (A6.8)

Step 1. Choose $\mathbf{X}_1$, $[\mathbf{A}_1]$ (initial metric), N

$\varepsilon_1, \varepsilon_2, \varepsilon_3$: (tolerance for stopping criteria)

Set $i = 1$ (initialize iteration counter)

Step 2. The search direction is obtained as a solution to

$[\mathbf{A}_i]\mathbf{S}_i = -\nabla f(\mathbf{X}_i)$

$\mathbf{X}_{i+1} = \mathbf{X}_i + \alpha_i \mathbf{S}_i$; $\Delta\mathbf{X} = \alpha_i \mathbf{S}_i$

α_i is determined by Minimizing $f(\mathbf{X}_{i+1})$

Step 3. If $\nabla f(\mathbf{X}_{i+1})^T \nabla f(\mathbf{X}_{i+1}) \le \varepsilon_3$, converged

If $|f(\mathbf{X}_{i+1}) - f(\mathbf{X}_i)| \le \varepsilon_1$, stop (function not changing)

If $\Delta\mathbf{X}^T \Delta\mathbf{X} \le \varepsilon_2$; stop (design not changing)

If $i + 1 = N$, stop (iteration limit)

Else

$\mathbf{Y} = \nabla f(\mathbf{X}_{i+1}) - \nabla f(\mathbf{X}_i)$

$$[\mathbf{B}] = \frac{\mathbf{Y}\mathbf{Y}^T}{\mathbf{Y}^T\Delta\mathbf{X}}$$

$$[\mathbf{C}] = \frac{\nabla f(\mathbf{X}_i)\,\nabla f(\mathbf{X}_i)^T}{\nabla f(\mathbf{X}_i)^T\mathbf{S}_i}$$

$[\mathbf{A}_{i+1}] = [\mathbf{A}_i] + [\mathbf{B}] + [\mathbf{C}]$

$i \leftarrow i + 1$

Go To Step 2

Application of BFGS Method: The method is applied to Example 6.1. Only essential computations are shown.

Step1. $\mathbf{X}_1 = \begin{bmatrix}0.5\\0.5\end{bmatrix}$; $f(\mathbf{X}_1) = 5.3125$; $[\mathbf{A}_1] = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$; $\nabla f(\mathbf{X}_1) = \begin{bmatrix}-0.5\\-2.25\end{bmatrix}$

Step 2. $\mathbf{S}_1 = \begin{bmatrix}0.5\\2.25\end{bmatrix}$; $\alpha_1 = 0.1993$; $\mathbf{X}_2 = \begin{bmatrix}0.5996\\0.9484\end{bmatrix}$; $\nabla f(\mathbf{X}_2) = \begin{bmatrix}-1.6460\\0.3658\end{bmatrix}$

Step 3. $\mathbf{Y} = \begin{bmatrix}-1.1460\\2.6158\end{bmatrix}$; $\Delta\mathbf{X} = \begin{bmatrix}0.0996\\0.4484\end{bmatrix}$

$$[\mathbf{B}] = \frac{\begin{bmatrix}-1.1460\\2.6158\end{bmatrix}[-1.1460\ \ 2.6158]}{[-1.1460\ \ 2.6158]\begin{bmatrix}0.0996\\0.4484\end{bmatrix}} = \begin{bmatrix}1.2403 & -2.8312\\-2.8312 & 6.4625\end{bmatrix}$$

$$[\mathbf{C}] = \frac{\begin{bmatrix}-1.6460\\0.3658\end{bmatrix}[-1.6460\ \ 0.3658]}{[-1.6460\ \ 0.3658]\begin{bmatrix}0.5000\\2.2500\end{bmatrix}} = \begin{bmatrix}-0.0471 & -0.2118\\-0.2118 & -0.9529\end{bmatrix}$$

$$[\mathbf{A}_2] = [\mathbf{A}_1] + [\mathbf{B}] + [\mathbf{C}] = \begin{bmatrix}2.1933 & -3.0429\\-3.0429 & 6.5095\end{bmatrix}$$

$i = 2$; Go To Step 2

Step 2. $\mathbf{S}_2 = \begin{bmatrix}1.9135\\0.8383\end{bmatrix}$; $\alpha_2 = 1.2544$; $\mathbf{X}_3 = \begin{bmatrix}3\\2\end{bmatrix}$; $f(\mathbf{X}_3) = 3.0$; $\nabla f(\mathbf{X}_3) = \begin{bmatrix}0\\0\end{bmatrix}$

Converged

The translation of the BFGS algorithm into code is left as an exercise for the student. It is recommended that the calculation of the search direction in Step 2 be accomplished as a solution to a set of linear equations (as listed) rather than inverting the matrix A. One other thing to note for Example 6.1 is that if the polynomial approximation was used for the one-dimensional process, the calculation of α^* would have been exact (why?).

6.4 NUMERICAL TECHNIQUES—SECOND ORDER

Second-order methods for unconstrained optimization are not used because evaluation of the Hessian matrix during each iteration the Hessian matrix is considered to be computationally expensive. A second-order method with the property of quadratic convergence is very impressive. An n-variable problem can converge in *one* iteration. As mentioned before, for real design problems, where decisions are required to be made on the discrete nature of some of the variables, the existence of the first derivative, not to mention second derivatives, is questionable. Moreover, the quasi-Newton methods of the last two sections are able to effectively function as second-order methods as they approach the solution—*and they do not need the estimation of second derivatives*. One second-order method is presented here for the sake of completeness.

Indeed, there is only one basic second-order technique for unconstrained optimization. It is based on the extension of the Newton-Raphson technique to multivariable problem. Many different extensions are available, but in this text, a

direct extension recast in the general algorithmic structure (A6.1) is presented. Once again the general problem and specific example are as follows.

$$\text{Minimize} \quad f(\mathbf{X}); \quad [\mathbf{X}]_n \tag{6.1}$$

$$\text{Subject to:} \quad x_i^l \le x_i \le x_i^u; \quad i = 1, 2, \ldots, n \tag{6.2}$$

$$\text{Minimize} \quad f(\mathbf{X}) = f(x_1, x_2) = 3 + (x_1 - 1.5x_2)^2 + (x_2 - 2)^2 \tag{6.3}$$

$$\text{Subject to:} \quad 0 \le x_1 \le 5; \quad 0 \le x_2 \le 5 \tag{6.4}$$

Modified Newton's Method: The Newton-Raphson method, used for a single variable, solves the problem $\phi(x) = 0$. This equation can represent the FOC for an unconstrained problem in one variable (Chapter 5). The iterative change in the variable is computed through

$$\Delta x = -\frac{\phi(x)}{\phi'(x)} = -\frac{f'(x)}{f''(x)} \tag{6.13}$$

where f is the single-variable objective function. The multivariable extension to computing a similar change in the variable vector is

$$\Delta \mathbf{X} = -[\mathbf{H}]^{-1} \nabla f(\mathbf{X}) \tag{6.14}$$

The original Newton-Raphson is not known for its robustness or stability and Equation (6.14) shares the same disadvantage. To control the design variable changes and to bring it under the scheme of algorithm (A6.1) the left-hand side is defined as the search direction vector S followed by a standard one-dimensional stepsize computation. This is termed the *modified Newton* method. The complete algorithm is as follows.

Algorithm: Modified Newton Method (A6.9)

Step 1. Choose $\mathbf{X}_1$, N

$\varepsilon_1, \varepsilon_2, \varepsilon_3$: (tolerance for stopping criteria)

Set $i = 1$ (initialize iteration counter)

Step 2. The search direction is obtained as a solution to

$[\mathbf{H}(\mathbf{X}_i)]S_i = -\nabla f(\mathbf{X}_i)$; [H] is the Hessian

$\mathbf{X}_{i+1} = \mathbf{X}_i + \alpha_i \mathbf{S}_i$; $\Delta \mathbf{X} = \alpha_i \mathbf{S}_i$

α_i is determined by Minimizing $f(\mathbf{X}_{i+1})$

Step 3. If $\nabla f(\mathbf{X}_{i+1})^T \nabla f(\mathbf{X}_{i+1}) \le \varepsilon_3$, converged

If $|f(\mathbf{X}_{i+1}) - f(\mathbf{X}_i)| \le \varepsilon_1$, stop (function not changing)

If $\Delta \mathbf{X}^T \Delta \mathbf{X} \le \varepsilon_2$, stop (design not changing)

If $i + 1 = N$, stop (iteration limit)

Else

$i \leftarrow i + 1$

Go To Step 2

Application of Modified Newton Method: Algorithm (A6.9) is used to solve Example 6.1. Only essential computations are included:

Step 1. $\mathbf{X}_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$; $f(\mathbf{X}_1) = 5.3125$

Step 2. $\begin{bmatrix} 2.0000 & -3.0000 \\ -3.0000 & 6.5000 \end{bmatrix}$ $\mathbf{S}_1 = -\begin{bmatrix} -0.5000 \\ -2.2500 \end{bmatrix}$

$\mathbf{S}_1 = \begin{bmatrix} 2.5000 \\ 1.5000 \end{bmatrix}$; $\alpha_1 = 1.0$; $\mathbf{X}_2 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$; $f(\mathbf{X}_2) = 3.0$; converged

As expected only a single iteration was necessary to obtain the solution. Translation of the algorithm into MATLAB code is once again left as an exercise.

6.5 ADDITIONAL EXAMPLES

Three additional examples are presented in this section. With many of the numerical techniques already programmed, obtaining the solution is a matter of judicious application of the numerical procedures. This section also presents some creative use of the methods and explores nonquadratic problems. The first example is the Rosenbrock problem [9]. This example was among those created to challenge the numerical techniques for unconstrained minimization. It is also sometimes referred to as the *banana function*. If you have a copy of the *Optimization Toolbox* from MATLAB, you will see this as part of the toolbox demo. You will also note that the toolbox contains many of the techniques developed in this section. The second example is a solution to a nonlinear two-point boundary value problem that is due to the Navier-Stokes equations describing flow due to a spinning disk. The last is an unusual data fitting example using Bezier curves. Once again the Optimization Toolbox presents its version of the data fitting example when running *optdemo* at the MATLAB prompt.

6.5.1 Example 6.2—Rosenbrock Problem

The Rosenbrock problem is

$$\text{Minimize} \quad f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{6.15}$$

$$\text{Subject to:} \quad -2 \le x_1 \le 5; \quad -2 \le x_2 \le 5$$

The side constraints are used for drawing contours. The solution to this problem is

$$x_1^* = 1.0; \quad x_2^* = 1.0; \quad f^* = 0.0$$

The problem is notorious for requiring a large number of iterations for convergences. Changes in design are small as the solution is being approached. Here the Random Walk and Conjugate Gradient methods are used to examine the solution. Both these methods are changed to draw appropriate contour levels. To avoid clutter, disable printing text information on the lines representing design changes. Also, since these methods are run for 200 iterations, the final printing to the Command window is avoided. The trace of the design variables on the contour plot is provided as the output from the program, along with the final values. As the figure is being drawn, it is clear that the progress to the solution is slow.

Solution Using Random Walk Method: Figure 6.7 presents the result of executing **RandomWalk.m** for this example. The program invocation and output from the Command window is

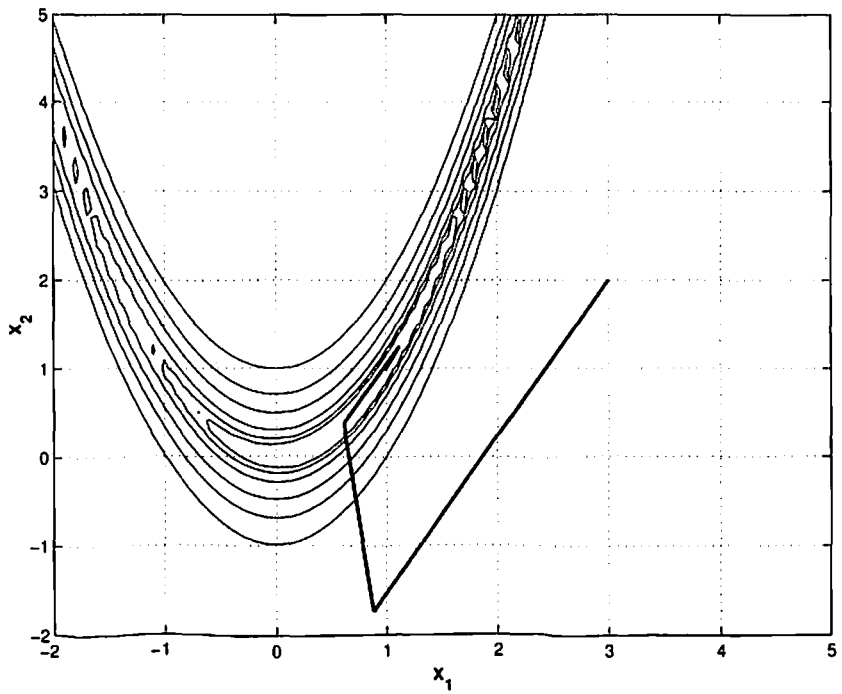

Figure 6.7 Random walk method: Example 6.2.


```
» RandomWalk('Example6_2',[3 2],200,1.0e-08,0,1,20)
The problem: Example6_2
ans =
 2.0000e+002 1.0000e+000 1.0000e+000 1.2327e-007
```

Since there are no stopping criteria built into the method (was suggested as an exercise), the method ran for 200 iterations (first value). The final value for the design vector and the objective function are very impressive:

$$\text{random walk}: x_1^* = 1.0; \quad x_2^* = 1.0; \quad f^* = 1.23e-07$$

Solution Using Conjugate Gradient Method: The invocation and output from the Command window is

```
» ConjugateGradient('Example6_2',[3 2],200,1.0e-08,0,1,20)
The problem: Example6_2
No. of iterations: 200
ans =
  9.6807e-001   9.3700e-001    1.0222e-003
```

The solution is

$$\text{conjugate gradient}: x_1^* = 0.96807; \quad x_2^* = 0.937; \quad f^* = 1.0222e-03$$

This appears no match for the Random Walk method for this investigation. Figure 6.8 tracks the design changes on the contour plot.

6.5.2 Example 6.3—Three-Dimensional Flow near a Rotating Disk

This example represents another of the exact solutions to the Navier-Stokes equations from fluid mechanics. The nonlinear two-point boundary value problem describes a viscous flow around a flat disk that rotates about an axis perpendicular to its plane with a constant angular velocity. After suitable redefinition [10] the mathematical description of the problem is the following set of coupled nonlinear equations with boundary conditions at two points:

$$2F + H' = 0 \tag{6.16a}$$

$$F^2 + F'H - G^2 - F'' = 0 \tag{6.16b}$$

$$2FG + HG' - G'' = 0 \tag{6.16c}$$

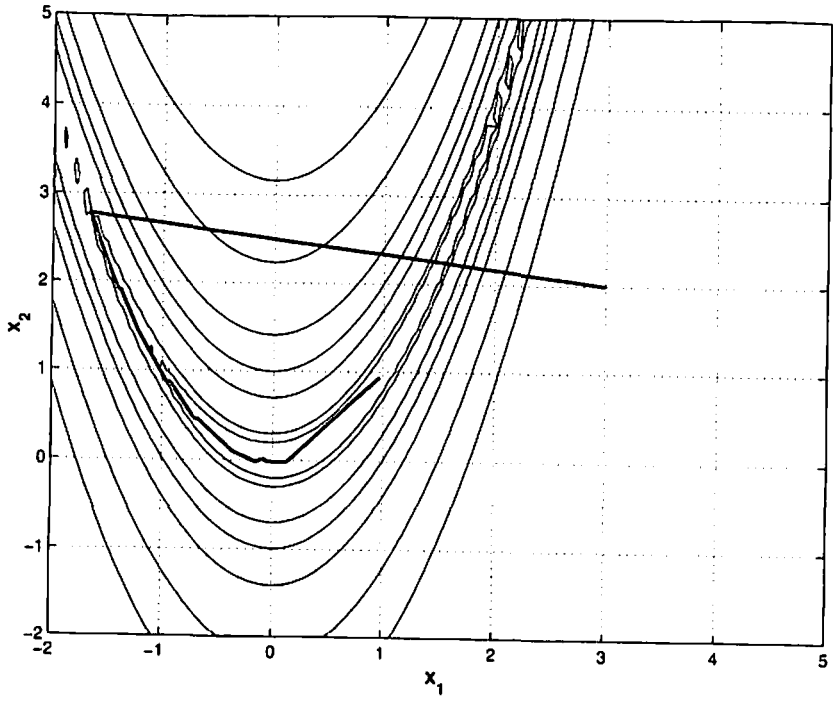

Figure 6.8 Conjugate gradient method: Example 6.2.

$$F(0) = 0;\quad G(0) = 1;\quad H(0) = 0 \tag{6.17a}$$

$$F(\infty) = 0;\quad G(\infty) = 0 \tag{6.17b}$$

Equations (6.16) are converted to state space form (see Chapter 5, Example 5.3). The design variables are the missing boundary conditions on $F'(0)$,$G'(0)$ and the final value of the independent variable (in lieu of ∞). The objective function is the squared error in the integrated value at the final condition.

The Optimization Problem

$$\text{Minimize}\quad f(x_1, x_2, x_3):\quad y_1(x_3)^2 + y_3(x_3)^2 \tag{6.18}$$

where the state variables are obtained from the solution of the following initial value problem:

$$\begin{bmatrix} y_1' \\ y_2' \\ y_3' \\ y_4' \\ y_5' \end{bmatrix} = \begin{bmatrix} y_2 \\ y_1^2 + y_2 y_5 - y_3^2 \\ y_4 \\ 2y_1 y_3 + y_4 y_5 \\ -2y_1 \end{bmatrix};\quad \begin{bmatrix} y_1(0) \\ y_2(0) \\ y_3(0) \\ y_4(0) \\ y_5(0) \end{bmatrix} = \begin{bmatrix} 0 \\ x_1 \\ 1 \\ x_2 \\ 0 \end{bmatrix} \tag{6.19}$$

The state equations (6.19) are integrated using MATLAB function *ode45*.

Solution to the Problem: The three-variable problem is solved using the DFP method. **Example6_3.m** returns the objective function. It calls the function **Ex6_3_state.m** which returns the system equations in state form. The call to the DFP method with the initial design vector of [1 −1 8] with 20 iterations of the DFP method, and a tolerance for the golden section of 0.0001 is shown below. Also in the call is the input to the upper bound calculation.

Usage: `DFP('Example6_3',[1 -1 8],20,0.0001,0,1,20)` (6.20)

The final value for the design after 20 iterations was

$$x_1^*(dfp) = 0.5101;\quad x_2^*(dfp) = -0.6157;\quad x_3^*(dfp) = 10.7627;\quad f^* = 5.7508\text{e-}006 \tag{6.21}$$

The tolerance for the golden section was only 0.0001. It is possible the objective cannot decrease until a smaller tolerance is specified.

The published values for the design variables are

$$x_1^* = 0.5101;\quad x_2^*(dfp) = -0.6159 \tag{6.22}$$

which is remarkably close to the values obtained by the DFP method. The value of x_3^* *is usually not specified* but is about 7. Since it is a substitute for infinity, the larger number should not be a problem.

Example 6.3 is not trivial problem. The integration is highly sensitive to the initial values. A reasonable starting point is essential to prevent the integration from generating NaN's (not a number). The second design variable must have a negative sign. The example illustrates the application of standard optimization technique to solve a nonlinear differential system. A similar application can be made to problems in system dynamics and optimal control. It is essential to understand that the algorithms outlined here are numerical tools that transcend any particular discipline.

There are numerical techniques that address two-point nonlinear boundary value problems more efficiently. However, computing resources, especially on a PC, are not a concern. Therefore, the procedure adopted in this example is an acceptable approach

for this class of problems. Even the Random Walk method is impressive in this instance.

6.5.3 Example 6.4—Fitting Bezier Parametric Curves

The example introduces a nontraditional and an unusual curve fitting procedure. Given a set of data points a *Bezier curve of user-chosen order* is fit to the data. Bezier curves are parametric curves that are a special case of uniform B-splines. Bezier parametrization is based on the *Bernstein basis* functions. Any point P on a two-dimensional Bezier curve (any parametric curve) is actually obtained as $P(x(v), y(v))$, where $0 \le v \le 1$. The actual relations are

$$P(v) = [x \;\; y] = \sum_{i=0}^{n} B_i J_{n,i}(v), \quad 0 \le v \le 1 \tag{6.23}$$

$$J_{n,i}(v) = \binom{n}{i} v^i (1 - v)^{n-i}$$

where B_i are the vertices of the polygon that determines the curve. B_i represent a pair of values in two-dimensional space. The order of the curve is n—the highest power in the basis functions. The actual computations of the points on the curve are easier using matrix algebra and are well explained and documented in Reference 11. Figure 6.9 represents a cubic Bezier curve. The following are some of the most useful properties of the curve as observed in the figure:

- The curve is completely defined by the polygon obtained by joining the vertices in order.
- The degree of the polynomial defining the curve is one less than the number of vertices of the polygon.
- The first and last points of the curve are coincident with the first and last vertex. The remaining vertex points do not typically lie on the curve.
- The slopes at the ends of the curve have the same direction as the sides of the polygon
- The curve is contained within the convex hull of the polygon.

In this example the convenient matrix representations for the curve are used. They are not discussed here, however, but are available in the code. For this example, the design variables are the inside polygon vertices. For $n = 5$, there are 6 vertices. The first and last vertex are known from the data points. This leaves 4 vertices or 8 design variables for $n = 5$.

Example6_4.m: This m-file is a stand-alone program to run the example in this section. Typing Example 6_4 at the prompt should start the program. Actually the

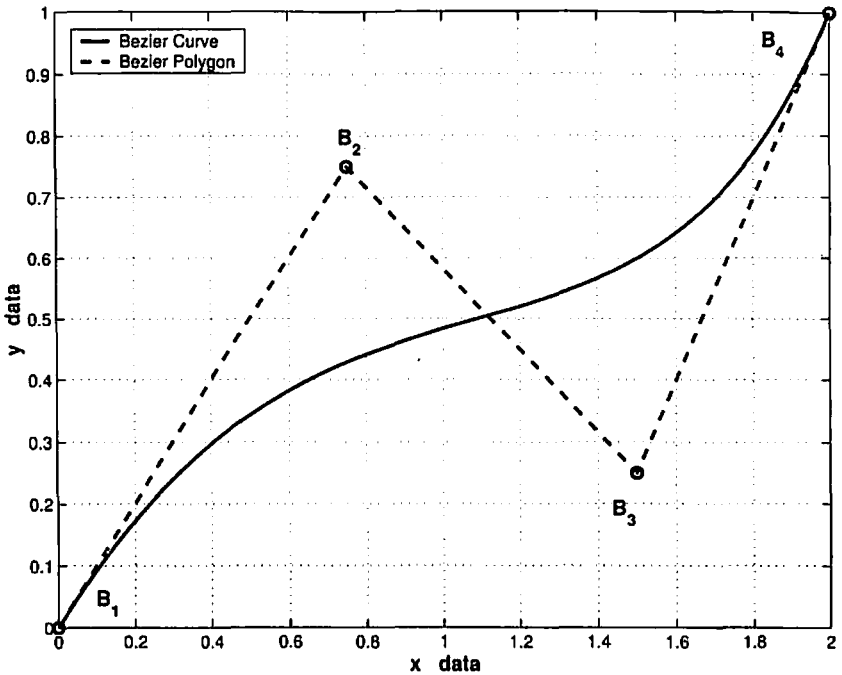

Figure 6.9 A cubic Bezier curve.

example is generic as most decisions are based on the text file that contains the data to be fitted. It calls on many m-files. The Bezier calculations are fairly modular. This program is not trivial. A more important reason for its inclusion is that, if understood, it signals a reasonable maturity with MATLAB programming. More importantly, it exemplifies the way to exploit the immense power of built-in MATLAB support. The author encourages the user to exert some effort tracing the progress of the calculations through the code. A lot of program control is based on the user's choice concerning the order of the Bezier curve. Also the initial values for the design variables *are generated automatically,* based on the data to be fitted. The function calling sequence used in this example can be better understood by the following structure:

Example6_5_3.m
Calls uigetfile to pick up the (*.txt) file that contains two-column data that will be fit (**Ex6_5_3.txt**)
Throws up an inputdialog box to obtain the order of fit from the user (**n = 5**)
Calls **coeff.m** with the order information
Calls **combination.m** – calculation of combination for Bernstein basis
Calls **Factorial.m** – calculates factorial of an integer

Calls **Curve_fit.m** which handles the optimization
Calls **DFP.m** (the objective function is computed in **Bez_Sq_Err.m**)
Calls **Bez_Sq_Err.m** – calculation of Objective function
Calls **gradfun.m** – calculation of derivatives
Calls **Gold_Section_nVar.m** - 1 D stepsize
Calls **UpperBound_nVar.m** – bracket the minimum

Since there are eight design variables, all of the information except for iteration number, convergence criteria, and the objective function is suppressed from the Command window to prevent scrolling and loss of comprehension.

The significant elements in this code are as follows:

- Picks up the data file through a file selection box.
- Uses an input dialog box to query the user on the order of the curve.
- The order determines the size of the coefficient matrix A and it is automatically generated.
- The number of design variables is automatically determined. Once again it is based on the order of the curve.
- The initial values for the design are automatically generated based on the data to be fit.
- The DFP method is used to find the optimum.
- The objective function is computed in the generic file **Bezier_Sq_Err.m**. The data specific to the example are passed through the MATLAB global statement.
- The original data set and the Bezier curve fitting the data are displayed at the end.
- The objective function and the convergence data for each iteration are printed to the Command window.

Objective Function: The objective function is the least squared error over all of the data points between the actual data and the fitted Bezier curve. It is expressed as

$$\text{Minimize: } f(\mathbf{X}): \sum_{k=1}^{nData}\left[y_{data} - y_{B_n}\right]^2 \tag{6.24}$$

The following information is copied from the Command window (the file is **Ex6_5_3.txt** and the order is 5):

```
» Example6_4
iteration number:      1
gradient length squared:        5.0001
objective function value:       0.5302
iteration number:      2
gradient length squared:        5.6890
objective function value:       0.2497
iteration number:      3
gradient length squared:        0.0546
objective function value:       0.1830
iteration number:      4
gradient length squared:        0.1322
objective function value:       0.1605
iteration number:      5
gradient length squared:        0.0573
objective function value:       0.1467
iteration number:      6
gradient length squared:        0.0374
objective function value:       0.1437
iteration number:      7
gradient length squared:        0.0071
objective function value:       0.1433
iteration number:      8
gradient length squared:        0.2793
objective function value:       0.1415
iteration number:      9
gradient length squared:        0.4337
objective function value:       0.1401
```


From the data above it appears that the error is around 0.14 and is difficult to reduce further. It also appears the reduction in the objective is accompanied by the increase in the gradient, after a certain number of iterations. It is moving away from satisfying the FOC. The best value for the FOC is about 0.007 and starts to worsen with further iterations. It is quite possible that numerical inaccuracies may be a problem as there are extensive computations involved in each iteration. It is also likely that the numerical gradient computation may need to be refined. Currently the gradients are computed using forward finite difference with a step of 0.001. These are parameters that need to be explored in optimization as there is no universal set of values for all classes of problems.

Figure 6.10 illustrates the original data and the Bezier curve after the last iteration. The polygons are not shown as our effort is to generate the curve. The fit is definitely acceptable and there is probably little the Bezier curve can do about a couple of points that are not very smoothly located. It is probably these points that do not allow the objective function to decrease further without increasing the gradient. This concludes another innovative use of unconstrained optimization. The author has used these curves to optimally design airfoils for low-speed flight.

Note on FOC for Unconstrained Minimization: This example has alerted us to the question of FOC $\nabla f = 0$. If analytical computation of the derivative is possible for a problem, then FOC has the likelihood of being met. In design problems like Example 6.4, where the elements of the gradient vector have the form

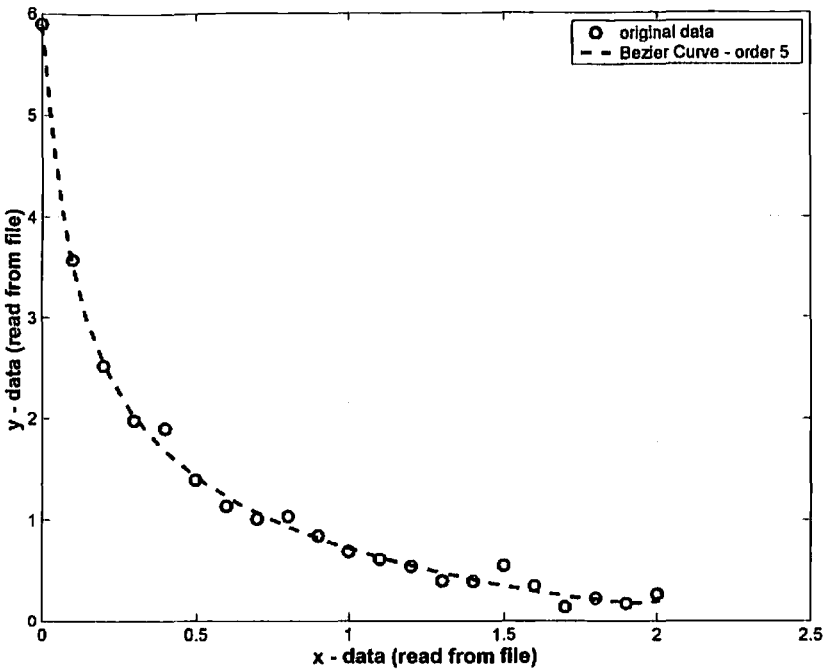

Figure 6.10 Original data and Bezier curve.

$$\frac{\partial\left\{\sum_{k=1}^{nData}\left[y_{data}-y_{B_n}\right]^2\right\}}{\partial x_i}$$

and which are best computed numerically, some specifications are needed regarding derivative computations if they are to be used for enforcing the necessary conditions.

REFERENCES

1. Fletcher, R., *Practical Methods for Optimization*, Vol.1, Wiley, New York, 1980.
2. Brent, R. P., *Algorithms for Minimization without Derivatives*, Prentice–Hall, Englewood Cliffs, NJ, 1973.
3. Powell, M. J. D., *Nonlinear Optimization*, Academic Press, New York, 1981.
4. Fox, R. L., *Optimization Methods for Engineering Design*, Addison–Wesley, Reading, MA, 1971.

5. Fletcher, R., and Reeves, R. M., Function Minimization by Conjugate Gradients, *The Computer Journal*, Vol. 7, pp. 149–180, 1964.
6. Davidon, W. C., Variable Metric Methods for Minimization, U.S. Atomic Energy Commission Research and Development Report No. ANL – 5990, Argonne National Laboratory, 1959.
7. Huang, H. Y., Unified Approach to Quadratically Convergent Algorithms for Function Minimization, *Journal of Optimization Theory and Applications*, Vol. 5, pp. 405–423, 1970.
8. Vanderplaats, G. N., *Numerical Optimization Techniques for Engineering Design*, McGraw–Hill, New York, 1984.
9. Hock, W., and Schittkowski, K., *Test Examples for Non Linear Programming Codes*, Lecture Notes in Economic and Mathematical Systems, 187, Springer-Verlag, Berlin, 1980.
10. Schlichting, H., *Boundary Layer Theory*, McGraw–Hill, New York, 1979.
11. Rogers, G. F., and Adams, J. A., *Mathematical Elements for Computer Graphics*, 2nd ed., McGraw–Hill, New York, 1990.

PROBLEMS

6.1 Apply the Random Walk method to Example 5.2.

6.2 Solve using the Random Walk method:

Minimize $f(x_1,x_2) = x_1^4 - 2x_1^2x_2 + x_1^2 + x_2^2 - 2x_1 + 4$

6.3 Apply the Pattern Search method to Example 5.2.

6.4 Solve using the Pattern Search method:

Minimize $f(x_1,x_2) = x_1^4 - 2x_1^2x_2 + x_1^2 + x_2^2 - 2x_1 + 4$

6.5 Modify the Pattern Search so that program control parameters for contour plotting, golden section, and upper bound calculation can be set by the user. Include the prompt and allow for default values in case the user decides not to take advantage of it.

6.6 Translate the Powell method into working MATLAB code. Verify the solution to Examples 6.1 and 5.2. Start from several points and verify that the number of cycles to converge is the same.

6.7 Verify that the search directions are conjugate with respect to the Hessian matrix for Examples 6.1 and 5.2.

6.8 Solve using Powell's method and that verify any two search directions are conjugate with respect to the Hessian matrix at the solution:

Minimize $f(x_1,x_2) = x_1^4 - 2x_1^2x_2 + x_1^2 + x_2^2 - 2x_1 + 4$

6.9 Solve Example 5.2 using the Steepest Descent, Conjugate Gradient, DFP, and BFGS methods.

6.10 Solve using the Steepest Descent, Conjugate Gradient, DFP, and BFGS methods the problem

Minimize $f(x_1,x_2) = x_1^4 - 2x_1^2x_2 + x_1^2 + x_2^2 - 2x_1 + 4$

6.11 Verify the values in Section 6.3.3 using a calculator. Verify if the matrix A at the solution is the inverse of the Hessian.

6.12 Verify if the matrix A at the solution is the inverse of the Hessian for DFP method for

Minimize $f(x_1,x_2) = x_1^4 - 2x_1^2x_2 + x_1^2 + x_2^2 - 2x_1 + 4$

6.13 Develop the BFGS method into MATLAB code and verify the calculations in Section 6.3.4.

6.14 Develop the Modified Newton method into MATLAB code and apply it to Example 5.2.

6.15 How will you incorporate the side constraints into the code for all of the various methods in this section? Implement them in the numerical procedures.

6.16 Solve Examples 6.2, 6.3, and 6.4 by one other method of the section.

6.17 Identify and solve a system dynamics problem using any method of this section.

6.18 Identify and solve your own curve fit problem.

7

NUMERICAL TECHNIQUES FOR CONSTRAINED OPTIMIZATION

This chapter explores algorithms/methods that handle the general optimization problem. Both equality and inequality constraints are included. For an engineering problem this will involve bringing several nonlinear relations into the design space. This is certain to increase the degree of difficulty in obtaining the solution. For the designer there is an additional burden of being more attentive to the design changes and the corresponding numbers to coax the solution if the mathematical definition of the problem is particularly severe. In all of these problems there are two outcomes that the algorithms seek to accomplish. The first is to ensure that the design is feasible (satisfies all constraints) and the second that it is optimal (satisfies the Kuhn-Tucker conditions). While the focus is on determining the solution, in times of difficulty, it is essential to remember that *feasibility* is more important that *optimality*. Also unstated in the rest of the chapter is that the optimal solution must be feasible. Two distinct approaches will be used to handle the constrained optimization problem. The first approach is termed the *indirect* approach and solves the problem by transforming it into an unconstrained problem. The second approach is to handle the constraints without transformation—the *direct* approach.

The indirect approach is an expression of incremental development of the subject to take advantage of the current methods. For example, it leverages the DFP method to handle constrained optimal problems. Two indirect methods are presented, the Exterior Penalty Function (EPF) method, and the Augmented Lagrange Multiplier (ALM) method. The direct approach handles the constraints and the objective together without any transformation. Four methods are presented. The methods in this book are Sequential Linear Programming (SLP), Sequential Quadratic Programming (SQP), Generalized Reduced Gradient Method (GRG), and Sequential Gradient Restoration Algorithm (SGRA).

7.1 PROBLEM DEFINITION

The standard format of the nonlinear programming problem (NLP) is reproduced here for convenience:

$$\text{Minimize} \quad f(x_1, x_2, \ldots, x_n) \tag{7.1}$$

$$\text{Subject to:} \quad h_k(x_1, x_2, \ldots, x_n) = 0, \quad k = 1, 2, \ldots, l \tag{7.2}$$

$$g_j(x_1, x_2, \ldots, x_n) \le 0, \quad j = 1, 2, \ldots, m \tag{7.3}$$

$$x_i^l \le x_i \le x_i^u \quad i = 1, 2, \ldots, n \tag{7.4}$$

In vector notation

$$\text{Minimize} \quad f(\mathbf{X}), \quad [\mathbf{X}]_n \tag{7.5}$$

$$\text{Subject to:} \quad [\mathbf{h}(\mathbf{X})]_l = 0 \tag{7.6}$$

$$[\mathbf{g}(\mathbf{X})]_m \le 0 \tag{7.7}$$

$$\mathbf{X}^{\text{low}} \le \mathbf{X} \le \mathbf{X}^{\text{up}} \tag{7.8}$$

For this chapter the following indices are reserved: n, number of variables; l, number of equality constraints; m, number of inequality constraints. Many of the algorithms and methods present a reasonable level of difficulty. A two-variable problem (for graphical illustration), with a single equality and inequality constraint is used to illustrate the various algorithms. All of the functions are nonlinear although the constraint functions have simple geometric shapes.

7.1.1 Problem Formulation—Example 7.1

Example 7.1 is a simple mathematical formulation. It does not represent any engineering problem. It is constructed to have a solution of $x_1^* = 1$ and $x_2^* = 1$ for the constrained problem and $x_1^* = 0.8520$ and $x_2^* = 0.8520$ for the unconstrained problem. The two constraint functions have simple nonlinearity. The equality constraint is a circle while the inequality constraint is an ellipse.

$$\text{Minimize} \quad f(x_1, x_2): \; x_1^4 - 2x_1^2 x_2 + x_1^2 + x_1 x_2^2 - 2x_1 + 4 \tag{7.9}$$

$$\text{Subject to:} \quad h(x_1, x_2): \; x_1^2 + x_2^2 - 2 = 0 \tag{7.10a}$$

$$g(x_1, x_2): \; 0.25\, x_1^2 + 0.75\, x_2^2 - 1 \le 0 \tag{7.10b}$$

$$0 \le x_1 \le 5; \;\; 0 \le x_2 \le 5 \tag{7.10c}$$

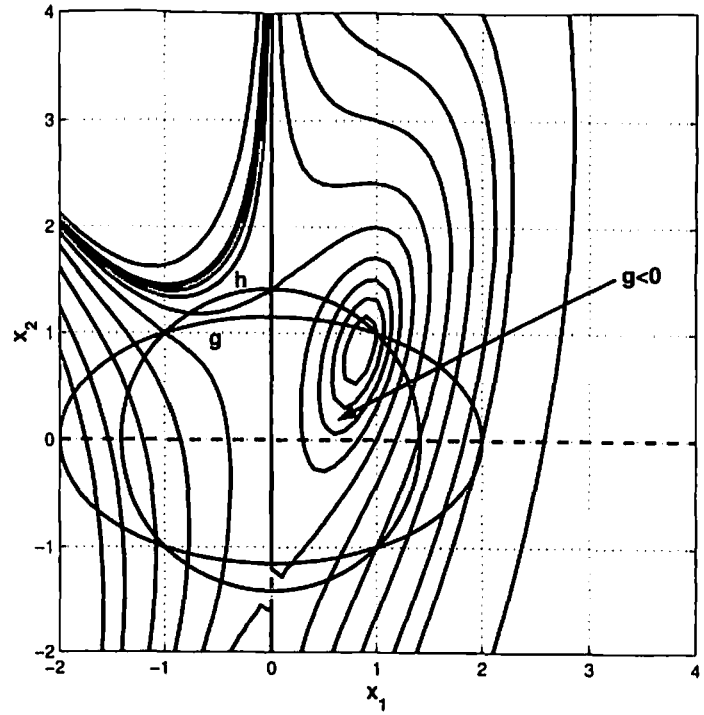

Figure 7.1 Graphical solution for Example 7.1.

Figure 7.1 displays the graphical solution to the problem. It is clear from the figure that the solution is at $x_1^* = 1$, $x_2^* = 1$ (why?). The value of the function is 3. A simple approach to the solution of the problem is to use Equation (7.10a) to eliminate one of the variables from the problem thereby reducing it to a single-variable constrained problem. This is not done here but it is a good exercise. **Fig7_1.m**[1] will create the figure.

7.1.2 Necessary Conditions

The necessary condition—or the Kuhn–Tucker (KT) conditions for the problem—was developed in Chapter 4. It is based on the minimization of the Lagrangian function (F). The conditions are

Minimize

$$F(x_1 \ldots x_n, \lambda_1 \ldots \lambda_l, \beta_1 \ldots \beta_m) = f(x_1, \ldots x_n) + \lambda_1 h_1 + \cdots + \lambda_l h_l + \beta_1 g_1 + \cdots + \beta_m g_m \tag{7.11}$$

[1]Files to be downloaded from the web site are indicated by boldface sans serif type.

subject to the constraints. There are $n + l + m$ unknowns in the Lagrangian. The same number of equations is required to solve the problem. These are provided by the FOC or Kuhn-Tucker conditions. Here n equations are obtained as

$$\frac{\partial F}{\partial x_i} = \frac{\partial f}{\partial x_i} + \lambda_1 \frac{\partial h_1}{\partial x_i} + \cdots + \lambda_l \frac{\partial h_l}{\partial x_i} + \beta_1 \frac{\partial g_1}{\partial x_i} + \cdots + \beta_m \frac{\partial g_m}{\partial x_i} = 0; \quad i = 1, 2, \ldots, n \tag{7.12}$$

l equations are obtained directly through the equality constraints

$$h_k(x_1, x_2, \ldots, x_n) = 0; \quad k = 1, 2, \ldots, l \tag{7.13}$$

m equations are applied through the 2^m cases

$$\begin{aligned} \beta_j g_j = 0 \rightarrow \text{if } \beta_j = 0 \quad &\text{then} \quad g_j < 0 \\ \text{if } g_j = 0 \quad &\text{then} \quad \beta_j > 0 \end{aligned} \tag{7.14}$$

The KT conditions apply only if the points are *regular* (see Example 4.3).

Application of the KT Conditions: The Lagrangian

$$\begin{aligned} F(x_1, x_2, \lambda, \beta): \quad & x_1^4 - 2x_1^2 x_2 + x_1^2 + x_1 x_2^2 - 2x_1 + 4 + \\ & \lambda(x_1^2 + x_2^2 - 2) + \beta(0.25\, x_1^2 + 0.75\, x_2^2 - 1) \end{aligned} \tag{7.15}$$

Three equations are set up through

$$\frac{\partial F}{\partial x_1} = 4\, x_1^3 - 4\, x_1 x_2 + 2\, x_1 + x_2^2 - 2 + 2\lambda\, x_1 + 0.5\, \beta\, x_1 = 0 \tag{7.16a}$$

$$\frac{\partial F}{\partial x_2} = -2\, x_1^2 + 2\, x_1\, x_2 + 2\, \lambda\, x_2 + 1.5\, \beta\, x_2 = 0 \tag{7.16b}$$

$$h = x_1^2 + x_2^2 - 2 = 0 \tag{7.16c}$$

The fourth equation is developed through

Case a: $\beta = 0$ With this information Equations (7.16) become

$$4\, x_1^3 - 4\, x_1\, x_2 + 2\, x_1 + x_2^2 - 2 + 2\, \lambda\, x_1 = 0 \tag{7.17a}$$

$$-2\, x_1^2 + 2\, x_1\, x_2 + 2\, \lambda\, x_2 = 0 \tag{7.17b}$$

$$x_1^2 + x_2^2 - 2 = 0 \tag{7.17c}$$

Equation set (7.17) is solved symbolically in MATLAB. There are two solutions that satisfy the side constraints (7.10c)

$$x_1^* = 0; \quad x_2^* = 1.4142; \quad \lambda^* = 0$$
$$x_1^* = 0.9275; \quad x_2^* = 1.0676; \quad \lambda^* = -0.1217$$

Both of these violate the inequality constraint. In Figure 7.1, the first solution definitely does but the second is a little difficult to see. Zooming should help. Case a does not provide a solution.

Case b: $g = 0$ This establishes four equations as

$$4\, x_1^3 - 4\, x_1 x_2 + 2\, x_1 - 2 + x_2^2 + 2\, \lambda\, x_1 + 0.5\, \beta\, x_1 = 0 \tag{7.18a}$$

$$-2\, x_1^2 + 2\, x_1\, x_2 + 2\, \lambda\, x_2 + 1.5\, \beta\, x_2 = 0 \tag{7.18b}$$

$$x_1^2 + x_2^2 - 2 = 0 \tag{7.18c}$$

$$0.25\, x_1^2 + 0.75\, x_2^2 - 1 = 0 \tag{7.18d}$$

Equations (7.18) are solved symbolically and the only solution satisfying the side constraints is

$$x_1^* = 1.0; \quad x_2^* = 1.0; \quad \lambda^* = -0.5000; \quad \beta^* = 0$$

which was expected (for the design variables not the multipliers). The value of $\beta^* = 0$ is not expected as this is an active constraint. The KT conditions can be considered to be weakly satisfied as β is not strictly positive. As in Chapter 6, wherever possible, the KT conditions are used to check for convergence.

7.1.3 Elements of a Numerical Technique

There is a distinct difference in the numerical techniques with incorporation of the constraints, even if the idea of using a search direction vector is still useful. The major differences in the algorithms of the previous chapter was the way the search directions were established. To understand the necessary changes, the general approach for unconstrained optimization as seen in the previous chapter was:

General Algorithm (A6.1)

Step 1. Choose $\mathbf{X}_0$

Step 2. For each iteration i

Determine search direction vector $\mathbf{S}_i$

(It would be nice if the objective decreased along this direction)

Step 3. Calculate $\Delta\mathbf{X}_i = \alpha_i \mathbf{S}_i$

Note: $\Delta\mathbf{X}_i$ is now a function of the scalar α_i as $\mathbf{S}_i$ is known from Step 2

α_i is called the step size as it establishes the length of $\Delta\mathbf{X}_i$

α_i is determined by Minimizing $f(\mathbf{X}_{i+1})$, where

$\mathbf{X}_{i+1} = \mathbf{X}_i + \Delta\mathbf{X}_i$

As the above steps are reviewed, in Step 2 a dilemma needs to be resolved: Should the search direction $\mathbf{S}_i$ decrease the objective function *or* should it attempt to satisfy the constraints? (It is assumed that it cannot do both though very often it will.) Since feasibility is more important than optimality, it is easier to identify the latter as a significant influence in determining the direction. This is more difficult than it sounds as each constraint can have its own favorite direction at the current point. Such occasions call for a trade-off and to see if it is possible to negotiate a decrease in the objective function as well. It is clear that Step 2 will involve investment of effort and some sophistication.

Step 3 is not far behind in the requirement for special handling. Once the direction is established (Step 2), what kind of stepsize will be acceptable? Several results are possible. First, the objective function can be decreased along the direction. Second, current active constraints (including the equality constraints) can become inactive, violated, or can still preserve the active state. Third, current inactive constraints can undergo a similar change of state. Fourth, probably most important, current violated constraints can become active or inactive. It is difficult to encapsulate this discussion in a generic algorithm. Some algorithms combine Steps 2 and 3 into a single step. Some divide the iteration into a feasibility component and an optimality component. In essence the generic algorithm for this chapter, which is not particularly useful, is:

Generic Algorithm (A7.1)

Step 1. Choose $\mathbf{X}_0$

Step 2. For each iteration i

Determine search direction vector $\mathbf{S}_i$

Step 3. Calculate $\Delta\mathbf{X}_i = \alpha_i \mathbf{S}_i$

$\mathbf{X}_{i+1} = \mathbf{X}_i + \Delta\mathbf{X}_i$

7.2 INDIRECT METHODS FOR CONSTRAINED OPTIMIZATION

These methods were developed to take advantage of codes that solve unconstrained optimization problems. They are also referred to as Sequential Unconstrained Minimization Techniques (SUMT) [1]. The idea behind the approach is to repeatedly call the *unconstrained optimization algorithm* using the solution of the previous

iteration. The unconstrained algorithm itself executes many iterations. This would require a robust unconstrained minimizer to handle a large class of probems. The BFGS method is robust and impressive over a large class of problems.

A preprocessing task involves tranforming the constrained problem into an unconstrained problem. This is largely accomplished by augmenting the objective function with additional functions that reflect the violation of the constraints very significantly. These functions are referred to as the *penalty functions*. There was significant activity in this area at one time which led to entire families of different types of penalty function methods [2]. The first of these was the Exterior Penalty Function (EPF) method. The first version of ANSYS [3] that incorporated optimization in its finite element program relied on the EPF. The EPF (presented in the next section) had several shortcomings. To address those the Interior Penalty Function (IPF) methods were developed leading to the Variable Penalty Function (VPF) methods. In this text, only the EPF is addressed largely due to academic interest. In view of the excellent performance of the direct methods, these methods will probably not be used today for continuous problems. They are once again important in *global optimization* techniques for constrained problems. The second method presented in this section, the Augmented Lagrange Method (ALM), is the best of the SUMT. Its exceedingly simple implementation, its quality of solution, and its ability to generate information on the Lagrange multipliers allow it to seriously challenge the direct techniques.

7.2.1 Exterior Penalty Function (EPF) Method

The transformation of the optimization problem (7.1)–(7.4) to an unconstrained problem is made possible through a penalty function formulation. The transformed unconstrained problem is:

$$\text{Minimize} \quad F(\mathbf{X}, r_h, r_g) = f(\mathbf{X}) + \mathbf{P}(\mathbf{X}, r_h, r_g) \tag{7.19}$$

$$x_i^l \le x_i \le x_i^u \qquad i = 1, 2, \ldots, n \tag{7.4}$$

where $\mathbf{P}(\mathbf{X}, r_h, r_g)$ is the penalty function. r_h and r_g are penalty constants (also called *multipliers*.)

The penalty function is expressed as

$$\mathbf{P}(\mathbf{X}, r_h, r_g) = r_h \left[\sum_{k=1}^{l} h_k(\mathbf{X})^2 \right] + r_g \left[\sum_{j=1}^{m} (\max\{0, g_j(\mathbf{X})\})^2 \right] \tag{7.20}$$

In Equation (7.20), if the equality constraints are not zero, their value gets squared and multiplied by the penalty multiplier and then gets added to the objective function. If the inequality constraint is in violation, it too gets squared and added to the objective function after being amplified by the penalty multipliers. In a sense, if the constraints

are not satisfied, then they are penalized, hence the function's name. It can be shown that the transformed unconstrained problem solves the original constrained problem as the multipliers r_h, r_g approach ∞. In order for $\mathbf{P}$ to remain finite at these values of the multipliers (needed for a valid solution), the constraints must be satisfied. In computer implementation this limit is replaced by a large value instead of ∞. Another facet of computer implementation of this method is that a large value of the multipliers at the first iteration is bound to create numerical difficulties. These multipliers are started with small values and updated geometrically with each iteration. The unconstrained technique, for example DFP, will solve Equation (7.19) for a known value of the multipliers. The solution returned from the DFP can be considered as a function of the multiplier and can be thought of as

$$\mathbf{X}^* = \mathbf{X}^*(r_h, r_g) \tag{7.21}$$

The SUMT iteration involves updating the multipliers and the initial design vector and calling the unconstrained minimizer again. In the algorithm it is assumed that the DFP method is used (although any method from Chapter 6 can be used, BFGS is recommended).

Algorithm: Exterior Penalty Function (EPF) Method (A7.2)

Step 1. Choose $\mathbf{X}^1$, N_s (no. of SUMT iterations), N_u (no. of DFP iterations)
ε_i's (for convergence and stopping)
r_h^1, r_g^1 (initial penalty multipliers)
c_h, c_g (scaling value for multipliers)
$q = 1$ (SUMT iteration counter)

Step 2. Call DFP to minimize $F(\mathbf{X}^q, r_h^q, r_g^q)$
Output: $\mathbf{X}^{q*}$

Step 3. Convergence for EPF
If $h_k = 0$, for $k = 1, 2, \ldots, l$;
If $g_j \le 0$, for $j = 1, 2, \ldots, m$;
If all side constraints are satisfied
Then Converged, Stop
Stopping Criteria:
$\Delta F = F_q - F_{q-1}$, $\Delta \mathbf{X} = \mathbf{X}^{q*} - \mathbf{X}^{(q-1)*}$
If $(\Delta F)^2 \le \varepsilon_1$: Stop (function not changing)
Else If $\Delta \mathbf{X}^T \Delta \mathbf{X} \le \varepsilon_1$: Stop (design not changing)
Else If $q = N_s$: Stop (maximum iterations reached)
Continue
$q \leftarrow q + 1$
$r_h^q \leftarrow r_h^{q*} C_h$; $r_g^q \leftarrow r_g^{q*} C_g$
$\mathbf{X}^q \leftarrow \mathbf{X}^{q*}$
Go to Step 2

The EPF is very sensitive to the starting value of the multipliers and to the scaling factors as well. Different problems respond favorably to different values of the multipliers. It is recommended that the initial values of the multipliers be chosen as the ratio of the objective function to the corresponding term in the penalty function at the initial design. This ensures that both the objective function and the constraints are equally important in determining the changes in the design for the succeeding iteration.

One reason for the term *Exterior Penalty* is that at the end of each SUMT iteration the design will be *infeasible* (until the solution is obtained). This implies that the method determines design values that are approaching the feasible region from the outside. This is a serious drawback if the method fails prematurely, as it will often do. The information generated so far is valueless as the designs were never feasible. As seen in the example below, the EPF severely increases the nonlinearity of the problem creating conditions for the method to fail. It is expected that the increase in nonlinearity is balanced by a closer starting value for the design, as each SUMT iteration starts closer to the solution than the previous one.

In the following the EPF is applied to Example 7.1 through a series of calculations rather than through the translation of the algorithm into MATLAB code. There are a couple of changes with respect to Algorithm (A7.1). To resolve the penalty function with respect to the inequality constraint, the constraint is assumed to always be in violation so that the return from the max function is the constraint function itself. This will drive the inequality constraint to be active which we know to be true for this example. Numerical implementation as outlined in the algorithm should allow determination of inactive constraints. Instead of numerical implementation of the unconstrained problem, an analytical solution is determined using MATLAB symbolic computation. Example 7.1 is reproduced for convenience.

$$\text{Minimize} \quad f(x_1, x_2): \quad x_1^4 - 2x_1^2 x_2 + x_1^2 + x_1 x_2^2 - 2x_1 + 4 \tag{7.9}$$

$$\text{Subject to:} \quad h(x_1, x_2): \quad x_1^2 + x_2^2 - 2 = 0 \tag{7.10a}$$

$$g(x_1, x_2): \quad 0.25x_1^2 + 0.75x_2^2 - 1 \le 0 \tag{7.10b}$$

$$0 \le x_1 \le 5; \quad 0 \le x_2 \le 5 \tag{7.10c}$$

Figure 7.2 is the contour plot of the transformed unconstrained function for values of $r_h = 1$ and $r_g = 1$. The increase in nonlinearity is readily apparent. Figure 7.3 is the plot for $r_h = 5$ and $r_g = 5$. Handling these functions numerically even with the BFGS will be a considerable challenge. Both the figures suggest several points that satisfy first order conditions. Their closeness makes it difficult for any numerical technique to find the optimum. It is clear that the EPF severely increases the nonlinearity of the problem. **Sec7_2_1_plot.m** contains the code that generates the plot. Since the code uses symbolic manipulation, actually drawing the plot takes time. Evaluating the data numerically will make a big difference.

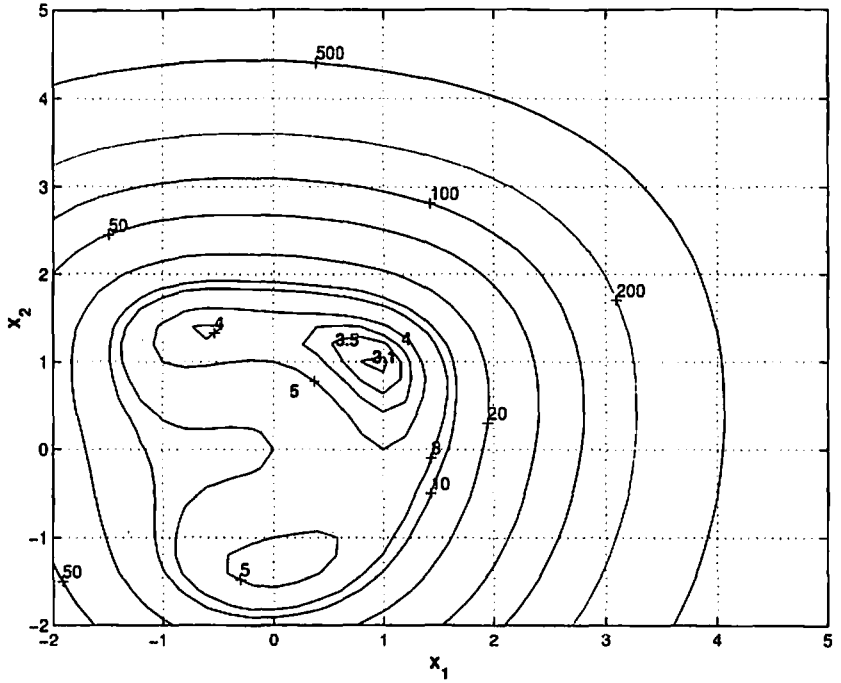

Figure 7.2 Exterior Penalty Function Method: Example 7.1 ($r_h = 1$, $r_g = 1$).

Application of EPF Algorithm (A7.2): The corresponding unconstrained function for this problem is

$$F(x_1, x_2, r_h, r_g) = (x_1^4 - 2x_1^2x_2 + x_1^2 + x_1x_2^2 - 2x_1 + 4) + r_h(x_1^2 + x_2^2 - 2)^2 + r_g(0.25x_1^2 + 0.75x_2^2 - 1)^2 \quad (7.22)$$

Sec7_2_1_calc.m is an m-file that will calculate the solution and the values of the function, for Example 7.1, for a predetermined set of values of the penalty multipliers. It requires two inputs from the user at different stages. The first input is the values for the multipliers for which the solution should be obtained. The list of solution (there are nine for this problem) is displayed in the Command window. The user finds the solution that satisfies the side constraints (usually one) which must be entered at the prompt (note: it must be entered as a vector). The program then prints out the values of the various functions involved in the example. The following is posted from the Command window for both of the penalty multipliers set to 25.

```
» Sec7_2_1_calc
enter value for rh [default = 1] : 25
```


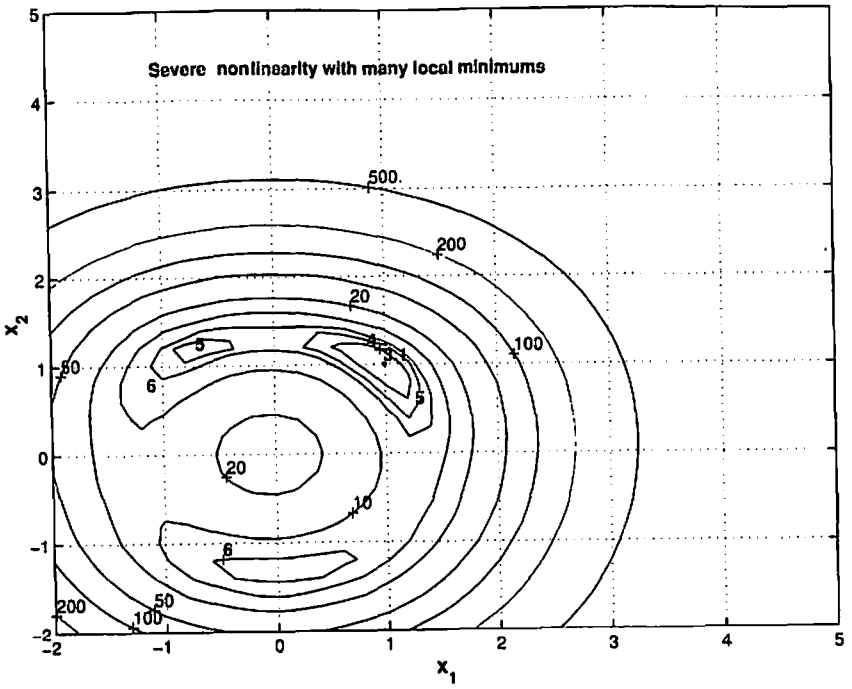

Figure 7.3 Exterior Penalty Function Method: Example 7.1 ($r_h = 5$, $r_g = 5$).

```
enter value for rg [default = 1] : 25
ans =
 -1.4125   -0.1015
 -0.8692    1.0980
 -0.6263   -1.2107
 -0.0278   -1.3266
 -0.0124    1.3267
 -0.0090   -0.0000
  0.7893   -1.1267
  0.9775    1.0165
  1.4184   -0.1318
Input the design vector chosen for evaluation
[0.9775 1.0165]
The design vector [ 0.9775 1.0165 ]
objective function: 2.9810
equality constraint: -0.0112
inequality constraint: 0.0138
```

In the above run, the equality and the inequality constraint are not satisfied (as expected). Table 7.1 documents the results of the application of the EPF method to Example 7.1 through **Sec7_2_1_calc.m**. The scaling factor is 5 for both multipliers. A glance at Table 7.1 clearly illustrates the characteristics of the EPF method . As the values of the multipliers increase:

- The design approaches the optimal value.
- The constraint violations decrease.
- The solution is being approached from outside the feasible region.

An important note needs to be made. The analytical computation using symbolic computation appeared easy. The same cannot be said for the numerical computation, especially looking at Figure 7.3.

7.2.2 Augmented Lagrange Multiplier (ALM) Method

This is the most robust of the penalty function methods. More importantly it also provides information on the Lagrange multipliers at the solution. This is achieved by not solving for the multipliers but merely updating them during successive SUMT iterations [2,4]. It overcomes many of the difficulties associated with the penalty function formulation without any significant overhead.

Transformation to the Unconstrained Problem: The general optimization problem (7.1)–(7.4) is transformed as in the method of Lagrange multipliers

Minimize

$$F(\mathbf{X}, \boldsymbol{\lambda}, \boldsymbol{\beta}, r_h, r_g): \quad f(\mathbf{X}) + r_h \sum_{k=1}^{l} h_k(\mathbf{X})^2 + r_g \sum_{j=1}^{m} \left(\max\left[g_j(\mathbf{X}), -\frac{\beta_j}{2r_g} \right] \right)^2 + \sum_{k=1}^{l} \lambda_k h_k(\mathbf{X}) + \sum_{j=1}^{m} \beta_j \left(\max\left[g_j(\mathbf{X}), -\frac{\beta_j}{2r_g} \right] \right) \tag{7.23}$$

$$x_i^l \le x_i \le x_i^u \quad i = 1, 2, \ldots, n \tag{7.4}$$

Table 7.1 Exterior Penalty Function Method: Example 7.1

Iteration	r_h	r_g	x_1	x_2	f	h	g
1	1.0	1.0	0.9228	1.0391	2.9578	–0.0687	0.0227
2	5.0	5.0	0.9464	1.0364	2.9651	–0.0302	0.0295
3	25.0	25.0	0.9775	1.0165	2.9810	–0.0112	0.0138
4	125	125	0.9942	1.0044	2.9944	–0.0027	0.0037
5	625	625	0.9988	1.0009	2.9988	–5.9775e-004	7.5097e-004

Here $\boldsymbol{\lambda}$ is the multiplier vector tied to the equality constraints, $\boldsymbol{\beta}$ is the multiplier vector associated with the inequality constraints, and r_h and r_g are the penalty multipliers used similar to the EPF method. F is solved as an unconstrained function for predetermined values of $\boldsymbol{\lambda}$, $\boldsymbol{\beta}$, r_h, and r_g. Therefore, the solution for each SUMT iteration is

$$\mathbf{X}^* = \mathbf{X}^*(\boldsymbol{\lambda}, \boldsymbol{\beta}, r_h, r_g)$$

At the end of the SUMT iteration the values of the multipliers and penalty constants are updated. The latter are usually geometrically scaled but unlike EPF do not have to be driven to ∞ for convergence.

Algorithm: Augmented Lagrange Multiplier (ALM) Method (A7.3)

Step 1. Choose $\mathbf{X}^1$, N_s (no. of SUMT iterations),
N_u (no. of DFP iterations)
ε_i's (for convergence and stopping)
r_h^1, r_g^1 (initial penalty multipliers)
c_h, c_g (scaling value for multipliers)
$r_h^{\max}$, $r_g^{\max}$ (maximum value for multipliers)
$\boldsymbol{\lambda}^1$, $\boldsymbol{\beta}^1$ (initial multiplier vectors)
$q = 1$ (SUMT iteration counter)

Step 2. Call DFP to minimize $F(\mathbf{X}^q, \boldsymbol{\lambda}^q, \boldsymbol{\beta}^q, r_h^q, r_g^q)$
Output: $\mathbf{X}^{q*}$

Step 3. Convergence for ALM
If $h_k = 0$, for $k = 1, 2, \ldots, l$;
If $g_j \le 0$, for $j = 1, 2, \ldots, m$;
(If $\beta_j > 0$ for $g_j = 0$)
(If $\nabla f + \Sigma \lambda_k \nabla h_k + \Sigma \beta_j \nabla g_j = 0$)
If all side constraints are satisfied
Then Converged, Stop
Stopping Criteria:
$\Delta F = F_q - F_{q-1}$, $\Delta \mathbf{X} = \mathbf{X}^{q*} - \mathbf{X}^{(q-1)*}$
If $(\Delta F)^2 \le \varepsilon_1$: Stop (function not changing)
Else If $\Delta \mathbf{X}^T \Delta \mathbf{X} \le \varepsilon_1$: Stop (design not changing)
Else If $q = N_s$: Stop (maximum iterations reached)
Continue
$q \leftarrow q + 1$
$\boldsymbol{\lambda}^q \leftarrow \boldsymbol{\lambda}^q + 2\, r_h\, \mathbf{h}(\mathbf{X}^{q*})$
$\boldsymbol{\beta}^q \leftarrow \boldsymbol{\beta}^q + 2\, r_g\, (\max[\mathbf{g}(\mathbf{X}^{q*}), -\boldsymbol{\beta}^q / 2r_g])$
$r_h^q \leftarrow r_h^q C_h$; $r_g^q \leftarrow r_g^q C_g$

$\mathbf{X}^q \leftarrow \mathbf{X}^{q*}$

Go to Step 2

The ALM algorithm (A7.3) is available as a stand-alone program in **ALM.m**. The program is long which is not unexpected. These algorithms are not trivial. To translate algorithm (A7.3) into working code is a tremendous accomplishment, particularly if your exposure to MATLAB occurred through this book. The reader should walk through the code line by line. The organization and structure are kept simple and straightforward. Comments have been liberally used as an encouragement to understand the code.

ALM.m: The code gathers input from the user through several prompts in the Command window. The prompts and the final iteration are copied below. The significant features of the code are:

- The starting design vector is the first input to the program. The number of design variables is automatically extracted from this information.
- The lower and upper bounds on the design variables (side constraints) are obtained through user prompts (though they are not used in the program).
- Number of equality constraints and corresponding initial multipliers (λ) are prompted.
- Number of inequality constraints and corresponding initial multipliers (β) are prompted.
- Values at each SUMT iteration are printed in the Command window. Warnings generated by MATLAB are switched off.
- The **DFP.m** is used for unconstrained optimization. This program uses **GoldSection_nVar.m**, **UpperBound_nVar.m**, and **gradfunction.m**.
- The program expects the following files to be available in the path:

Objective function	**Ofun.m**
Equality constraints	**Hfun.m**
Inequality constraints	**Gfun.m**
Unconstrained function	**FALM.m**

- The program uses global statements to communicate multiplier values.
- The initial penalty multipliers are computed automatically.
- Several parameters are coded into the program, especially those needed for the golden section and the upper bound calculation. These can be changed by the user, if necessary.

Application to Example 7.1: The following represents the initial prompts seen in the Command window and the values for the starting iterations. Other iterations are similarly displayed. Also included below is a consolidated printing

of values for all of the SUMT iterations at the termination of the program. The initial design vector is [3 2]. Initial $\lambda = 1$ and $\beta = 1$.

```
» ALM
Input the starting design vector
This is mandatory as there is no default vector setup
The length of your vector indicates the number of
unknowns (n)
Please enter it now and hit return : for example [ 1 2
3 ...]
[3 2]

 The initial design vector:
   3      2

Input the minimum values for the design vector. .
These are input as a vector. The default values are
3 * start value below the start value unless it is zero.
In that case it is -5:
[0 0]

The minimum values for the design variables are : 0 0

Input the maximum values for the design vector.
These are input as a vector. The default values are
3 * start value above the start value.
If start value is 0 then it is +5:
[5 5]

The maximum values for the design variables are : 5 5

Number of equality constraints [0] : 1
Initial values for lambda : 1

Number of inequality constraints [0] : 1
Initial values for beta : 1

ALM iteration number: 0
Design Vector (X) : 3 2
Objective function : 64
Square Error in constraints(h, g) : 1.2100e+002
1.8063e+001
Lagrange Multipliers (lambda beta): 1 1
Penalty Multipliers (rh rg): 5.2893e-001 3.5433e+000
```

```
(other iteration omitted)
----------------------
X not changing : 0.000E+000 reached in 5 iterations

 The values for x and f and g and h are :
   3.0000e+000   2.0000e+000   6.4000e+001   1.8063e+001
1.2100e+002
  7.9603e-001 7.1223e-001 2.9443e+000 0 7.3799e-001
   9.9937e-001   1.0092e+000   2.9995e+000   1.8223e-004
2.9434e-004
  9.9607e-001 1.0013e+000 2.9962e+000 0 2.7319e-005
   9.9940e-001   1.0002e+000   2.9994e+000   2.5872e-011
6.2772e-007
   9.9940e-001   1.0002e+000   2.9994e+000   2.5872e-011
6.2772e-007

 The values for lamda and beta are :
 1.0000e+000 1.0000e+000
 -9.0876e-001 0
 -1.6493e-001 9.5663e-001
 9.1192e-002 9.5616e-001
 -7.6430e-002 9.9221e-001
```

The program took five iterations (fifth was not necessary) to find the solution as:

$$x_1^* = 0.9994;\ x_2^* = 1.0002;\ f^* = 2.9994;\ h^* = 0.0000;\ g^* = 0.0007 \quad (7.24)$$

which is very close to the actual solution. The values for the multipliers are

$$\lambda^* = -0.0763;\ \beta^* = 0.9922 \quad (7.25)$$

This satisfies the KT conditions. However, in Section 7.1.2 the values of the multipliers, obtained using symbolic computation, are quite different. It is difficult to explain the discrepancy except to say that the symbolic calculation failed to discover this particular solution which is indeed possible. The values in (7.25) are more acceptable than those in Section 7.1.2 considering the KT conditions.

The ALM method, as demonstrated above, is an effective and useful method in spite of belonging to the class of indirect methods. It has several significant advantages over the other penalty function formulations [2]. Some of them are listed below:

- The method is not sensitive to the initial penalty multipliers or their scaling strategy.
- The method does not require the penalty multipliers to be driven to extreme values to establish convergence.
- The equality and active constraint can be satisfied precisely.

- The starting design can be feasible or infeasible.
- The initial choices for the multipliers can be relaxed.
- There is only a modest increase in the calculations.
- At convergence the Lagrange multipliers will be driven to their optimum values. This allows verification of KT conditions through the sign of the multipliers associated with the active inequality constraints.

7.3 DIRECT METHODS FOR CONSTRAINED OPTIMIZATION

The direct methods include both the objective and the constraints to search for the optimal solution. While the methods do not involve conversion to a different class of problems (like Section 7.2), most of them are based on *linearization* of the functions about the current design point. Linearization is based on expansion of the function about the current variable values using the Taylor series (Chapter 4).

Linearization: The Taylor series for a *two-variable* expanded function $f(x, y)$, quadratically about the current point (x_p, y_p) is expressed as:

$$f(x_p + \Delta x, y_p + \Delta y) = f(x_p, y_p) + \left[\left.\frac{\partial f}{\partial x}\right|_{(x_p,y_p)} \Delta x + \left.\frac{\partial f}{\partial y}\right|_{(x_p,y_p)} \Delta y \right] + \frac{1}{2}\left[\left.\frac{\partial^2 f}{\partial x^2}\right|_{(x_p,y_p)} (\Delta x)^2 + 2 \left.\frac{\partial^2 f}{\partial x \partial y}\right|_{(x_p,y_p)} \Delta x \Delta y + \left.\frac{\partial^2 f}{\partial y^2}\right|_{(x_p,y_p)} (\Delta y)^2 \right] \quad (7.26)$$

If the displacements are organized as a column vector $[\Delta x \quad \Delta y]^T$, the expansion in Equation (7.26) can be expressed in a condensed manner as

$$f(x_p + \Delta x, y_p + \Delta y) = f(x_p, y_p) + \nabla f|^T_{(x_p,y_p)} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} + \frac{1}{2} [\Delta x \quad \Delta y]^T [\mathbf{H}(x_p, y_p)] \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \quad (7.27)$$

For n variables, with X_p the current point and ΔX the displacement vector,

$$f(\mathbf{X}_p + \Delta \mathbf{X}) = f(\mathbf{X}_p) + \nabla f(\mathbf{X}_p)^T \Delta \mathbf{X} + \frac{1}{2} \Delta \mathbf{X}^T \mathbf{H}(\mathbf{X}_p) \Delta \mathbf{X} \quad (7.28)$$

Equation (7.28) can be written in terms of the difference in function values as

$$\Delta f = f(\mathbf{X}_p + \Delta \mathbf{X}) - f(\mathbf{X}_p) = \nabla f(\mathbf{X}_p)^T \Delta \mathbf{X} + \frac{1}{2} \Delta \mathbf{X}^T \mathbf{H}(\mathbf{X}_p) \Delta \mathbf{X} \quad (7.29)$$

$$\Delta f = \delta f + \delta^2 f$$

where $\delta f = \nabla f^T \Delta \mathbf{X}$ is termed the *first variation*. $\delta^2 f$ is the *second variation* and is given by the second term in the above equation. In linearization of the function f about the current value of design $\mathbf{X}_p$, only the first variation is used. The neighboring value of the function can be expressed as

$$\tilde{f}(\mathbf{X}_p) = f(\mathbf{X}_p) + \nabla f(\mathbf{X}_p)^T \Delta \mathbf{X} \tag{7.30}$$

All of the functions in the problem can be linearized similarly. It is essential to understand the difference between $f(\mathbf{X})$ and $\tilde{f}(\mathbf{X}_p)$. This is illustrated in Figure 7.4 using the objective function of Example 7.1 expanded about the current design $x_1 = 3$, $x_2 = 2$. The curved lines $f(\mathbf{X})$ are the contours of the original function. The straight lines $\tilde{f}(\mathbf{X}_p)$ are the contours of the linearized function (lines of constant value of the function) obtained through the following:

Original function: $f(x_1, x_2)$: $x_1^4 - 2x_1^2 x_2 + x_1^2 + x_1 x_2^2 - 2x_1 + 4$

Linearized function: $\tilde{f}(\Delta x_1, \Delta x_2) = 64 + 92\,\Delta x_1 - 6\,\Delta x_2$

where the coefficients in the second expression includes the evaluation of the function and gradient at $x_1 = 3, x_2 = 2$ and substituted in Equation (7.30). In Figure 7.4 several contours are shown although those of interest are in the neighborhood of the current design. Any nonlinear function can be expanded in the same manner. If another point was chosen, then the slopes of the lines would be different.

The quadratic expansion of the function can be obtained by using Equation (7.28). The expanded curves will be nonlinear (quadratic). They appear as ellipses in Figure 7.5 where several contours of the objective function of Example 7.1 are expanded quadratically about $x_1 = 3, x_2 = 2$. It is important to recognize that the contours would be different if another point were chosen for expansion.

Four direct methods are discussed The first is Sequential Linear Programming (SLP), where the solution is obtained by successively solving the corresponding linearized optimization problem. The second, Sequential Quadratic Programming (SQP), uses the quadratic expansion for the objective function. Like the SUMT, the current solution provides the starting values for the next iteration. The third, the Generalized Reduced Gradient Method (GRG), develops a sophisticated search direction and follows it with an elaborate one-dimensional process. The fourth, the Sequential Gradient Restoration Algorithm (SGRA), uses a two-cycle approach working on feasibility and optimality alternately to find the optimum. There are several other methods due to many researchers but they differ from those listed in small details. To keep the length of this chapter reasonable, they have not been included. With the code and other procedures available from this section, it should not be difficult to program the additional techniques that are worth exploring for particular classes of problems.

Except for special classes of problems in the following two chapters, this chapter is the principal reason for the book. It provides a resolution of the complete optimization problem as postulated. For a long time now there have not been any new or greatly improved numerical techniques. There is a lot of work on global optimization.

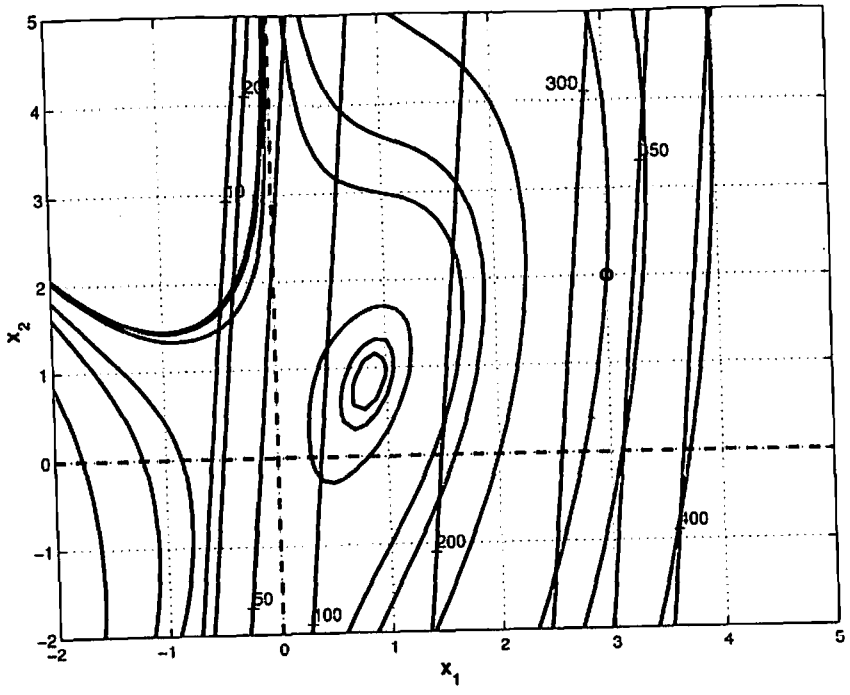

Figure 7.4 Illustration of linearization.

Current interest in traditional design optimization (this chapter), is directed toward actual design applications and migration from the mainframe and UNIX environment to the PC (like this book). For completeness, the standard format of the NLP is reproduced here for convenience:

$$\text{Minimize} \quad f(x_1, x_2, \ldots, x_n) \tag{7.1}$$

$$\text{Subject to:} \quad h_k(x_1, x_2, \ldots, x_n) = 0, \quad k = 1, 2, \ldots, l \tag{7.2}$$

$$g_j(x_1, x_2, \ldots, x_n) \le 0, \quad j = 1, 2, \ldots, m \tag{7.3}$$

$$x_i^l \le x_i \le x_i^u, \quad i = 1, 2, \ldots, n \tag{7.4}$$

In vector notation

$$\text{Minimize} \quad f(\mathbf{X}), \quad [\mathbf{X}]_n \tag{7.5}$$

$$\text{Subject to:} \quad [h(\mathbf{X})]_l = 0 \tag{7.6}$$

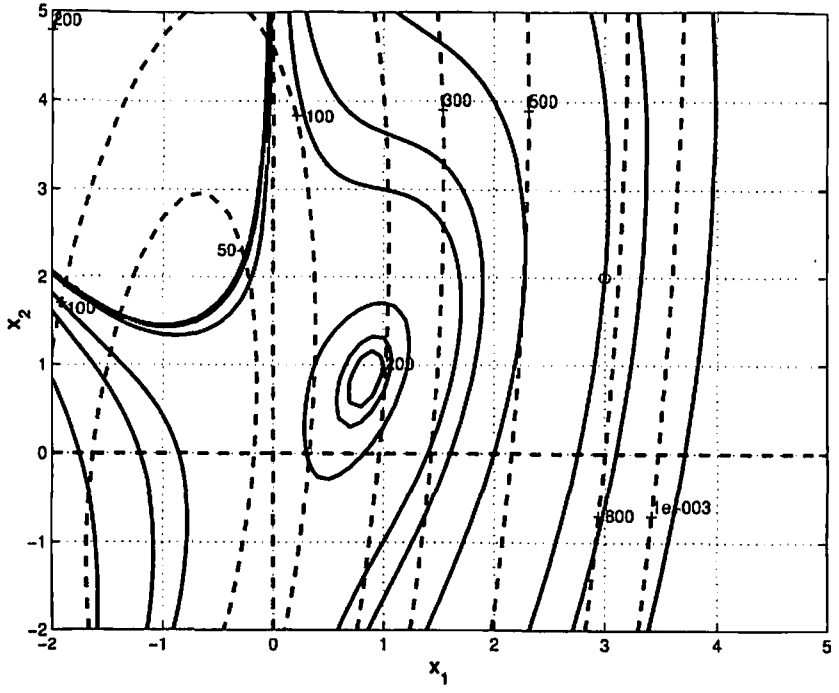

Figure 7.5 Illustration of quadratic expansion.

$$[g(\mathbf{X})]_m \le 0 \tag{7.7}$$

$$\mathbf{X}^{\text{low}} \le \mathbf{X} \le \mathbf{X}^{\text{up}} \tag{7.8}$$

Example 7.1 will be used to illustrate the various algorithms.

$$\text{Minimize} \quad f(x_1, x_2): \quad x_1^4 - 2x_1^2x_2 + x_1^2 + x_1x_2^2 - 2x_1 + 4 \tag{7.9}$$

$$\text{Subject to:} \quad h(x_1, x_2): \quad x_1^2 + x_2^2 - 2 = 0 \tag{7.10a}$$

$$g(x_1, x_2): \quad 0.25x_1^2 + 0.75x_2^2 - 1 \le 0 \tag{7.10b}$$

$$0 \le x_1 \le 5; \quad 0 \le x_2 \le 5 \tag{7.10c}$$

7.3.1 Sequential Linear Programming (SLP)

In the SLP [5] all of the functions are expanded linearly. If $\mathbf{X}_i$ is considered the current design vector, then the linearized optimal problem can be set up as

$$\text{Minimize:} \quad \tilde{f}(\Delta\mathbf{X}) = f(\mathbf{X}_i) + \nabla f(\mathbf{X}_i)^{\mathrm{T}}\,\Delta\mathbf{X} \tag{7.31}$$

$$\text{Subject to:} \quad \tilde{h}_k(\Delta\mathbf{X}): \; h_k(\mathbf{X}_i) + \nabla h_k^{\mathrm{T}}(\mathbf{X}_i)\Delta\mathbf{X} = 0; \quad k = 1, 2, \ldots, l \tag{7.32}$$

$$\tilde{g}_j(\Delta\mathbf{X}): \; g_j(\mathbf{X}_i) + \nabla g_j(\mathbf{X}_i)^{\mathrm{T}}\,\Delta\mathbf{X} \le 0; \quad j = 1, 2, \ldots, m \tag{7.33}$$

$$\Delta x_i^{\text{low}} \le \Delta x_i \le \Delta x_i^{\text{up}}; \quad i = 1, 2, \ldots, n \tag{7.34}$$

Equations (7.31)–(7.34) represent a Linear Programming (LP) problem (Chapter 3). All of the functions in Equations (7.31)–(7.34), except for $\Delta\mathbf{X}$, have numerical values after substituting a numerical vector for $\mathbf{X}_i$. Assuming an LP program code is available, it can be called repeatedly after the design is updated as

$$\mathbf{X}_{i+1} = \mathbf{X}_i + \Delta\mathbf{X}$$

In order to include a search direction and stepsize calculation, the $\Delta\mathbf{X}$ in Equations (7.31)–(7.33) is considered as a solution for $\mathbf{S}$ (a search direction at the current design). Stepsize computation strategy is no longer simple. If there are violated constraints, alpha attempts to reduce this violation. If the current solution is feasible, the stepsize will attempt to reduce the function without causing the constraints to be excessively violated. Such strategies are largely implemented at the discretion of the algorithm or code developer.

Algorithm: Sequential Linear Programming (SLP) (A7.4)

Step 1. Choose $\mathbf{X}^1$, N_s (no. of iterations),
ε_i's (for convergence and stopping)
$q = 1$ (iteration counter)

Step 2. Call LP to optimize (7.31)–(7.34)
Output: $\mathbf{S}$
Use a constrained α^* calculation ($\mathbf{X}_i = \mathbf{X}^q$)
$\Delta\mathbf{X} = \alpha^*\,\mathbf{S}$
$\mathbf{X}^{q+1} = \mathbf{X}^q + \Delta\mathbf{X}$

Step 3. Convergence for SLP
If $h_k = 0$, for $k = 1, 2, \ldots, l$;
If $g_j \le 0$, for $j = 1, 2, \ldots, m$;
If all side constraints are satisfied
If KT conditions satisfied
Then Converged, Stop
Stopping Criteria:
$\Delta\mathbf{X} = \mathbf{X}^{q+1} - \mathbf{X}^q$
If $\Delta\mathbf{X}^{\mathrm{T}}\Delta\mathbf{X} \le \varepsilon_1$: Stop (design not changing)
If $q = N_s$: Stop (maximum iterations reached)

Continue

$q \leftarrow q + 1$

Go to Step 2

The SLP is applied to Example 7.1 below. In Chapter 3 no explicit program for solving the LP program was developed although a command-by-command walk through was introduced. A program for applying the Simplex method, **psimplex.m** is included on the website (without any comment or discussion). Using instructions from Chapter 3, adding some program control, and organizing some bookkeeping can provide a MATLAB program for LP problems. Alternately, MATLAB's *linprog* can be used in a loop to help translate the SLP algorithm to code. In this section the linearized subproblem in each iteration is graphically presented. The area of interest can be zoomed and the solution read from the plot. This is treated as $\Delta\mathbf{X}$ (instead of using it as S). The design is updated and another plot can be obtained by running the m-file for a new value of the design vector. **Sec7_3_1_plot.m** assists in displaying the linearized functions from which solution can be obtained. The equality constraint is the green line. The red is inequality constraint. The design vector is an input to the program. The code uses symbolic calculation to solve and display the subproblem.

Application of SLP

Step 1. $\mathbf{X}^1 = [\,3\;2\,]$

Step 2. Linearized subproblem

$$\text{Minimize}\quad \tilde{f}(\Delta\mathbf{X}) = 64 + [92\;\;-6]\begin{bmatrix}\Delta x_1\\ \Delta x_2\end{bmatrix}$$

$$\text{Subject to: } \tilde{h}(\Delta\mathbf{X}):\ 11 + [6\;\;4]\begin{bmatrix}\Delta x_1\\ \Delta x_2\end{bmatrix} = 0$$

$$\tilde{g}(\Delta\mathbf{X}):\ 4.25 + [1.5\;\;3]\begin{bmatrix}\Delta x_1\\ \Delta x_2\end{bmatrix} \le 0$$

From Figure 7.6 (zoomed) the solution is $\delta x_1 = \Delta x_1 = -1.335$, $\delta x_2 = \Delta x_2 = -0.75$. For the next iteration $\mathbf{X}^2 = [\,1.665, 1.25]$. Repeat Step 2.

Step 2. Linearized subproblem

$$\text{Minimize } \tilde{f}(\Delta\mathbf{X}) = 6.7985 + [13.0305\;\;-1.382]\begin{bmatrix}\Delta x_1\\ \Delta x_2\end{bmatrix}$$

$$\text{Subject to: } \tilde{h}(\Delta\mathbf{X}):\ 2.3347 + [3.33\;\;2.5]\begin{bmatrix}\Delta x_1\\ \Delta x_2\end{bmatrix} = 0$$

$$\tilde{g}(\Delta\mathbf{X}):\ 0.8649 + [0.8325\;\;1.875]\begin{bmatrix}\Delta x_1\\ \Delta x_2\end{bmatrix} \le 0$$

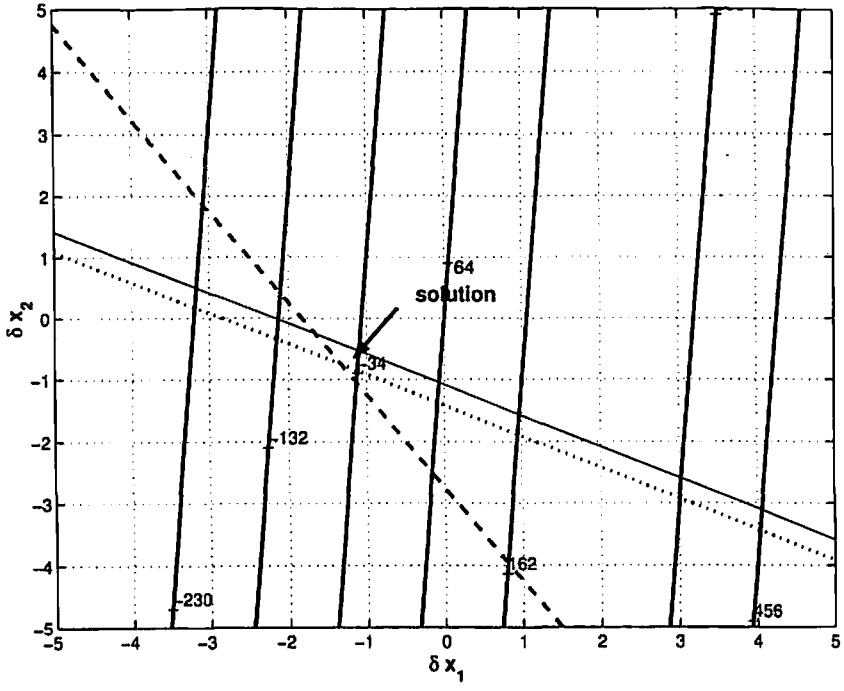

Figure 7.6 Example 7.1 SLP iteration 1.

From Figure 7.7 (zoomed) the solution is $\Delta x_1 = -0.54$, $\Delta x_2 = -0.23$. For the next iteration $\mathbf{X}^3 = [1.125, 1.02]$. Repeat Step 2.

Step 2. Linearized subproblem

$$\text{Minimize } \tilde{f}(\Delta\mathbf{X}) = 3.206 + [2.3957\;\;-0.2363]\begin{bmatrix}\Delta x_1\\ \Delta x_2\end{bmatrix}$$

$$\text{Subject to: } \tilde{h}(\Delta\mathbf{X}):\ 0.3060 + [2.25\;\;2.04]\begin{bmatrix}\Delta x_1\\ \Delta x_2\end{bmatrix} = 0$$

$$\tilde{g}(\Delta\mathbf{X}):\ 0.067 + [0.5625\;\;1.53]\begin{bmatrix}\Delta x_1\\ \Delta x_2\end{bmatrix} \le 0$$

From Figure 7.8 (zoomed) the solution is $\Delta x_1 = -0.125$, $\Delta x_2 = -0.02$. For the next iteration $\mathbf{X}^4 = [\,1.0\;\;1.0\,]$. Since this is the known solution, further iterations are stopped.

The SLP took three iterations to find the solution. This is very impressive. It is true only for those cases where the number of equality plus active inequality constraints is

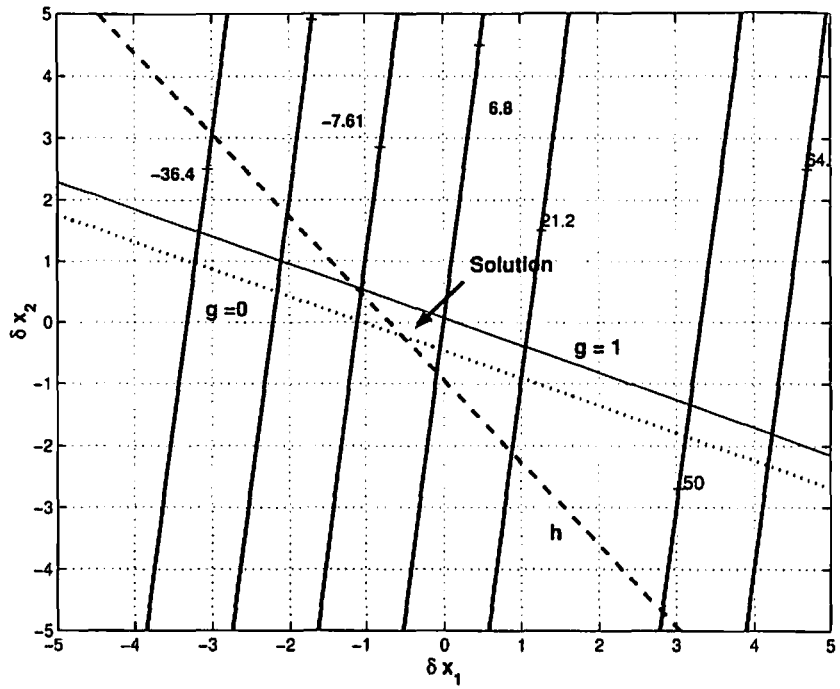

Figure 7.7 Example 7.1 SLP iteration 2.

the same as the number of design variables—a *full complement*. The main disadvantage of the method appears if there is no full complement. In that case the side constraints (7.34) are critical to the determination of the solution as they develop into active constraints. In these situations the limits on the side constraints are called the *move limits*. In case of active side constraints the move limits establish the values of the design changes. If the design changes are considered as the search direction (see algorithm), then both *value* and *direction* are affected. For these changes to be small as the solution is approached, the move limits *have to be adjusted (lowered) with every iteration*. The strategy for this adjustment may influence the solution.

Example 7.1 without the inequality constraint, but with the equality constraint provides an illustration of the move limits. Figure 7.9 represents the linearization about the point [3, 2]. The move limits in the figure (box) represent the side constraint limits

$$-2 \le \Delta x_1 \le 2; \qquad -2 \le \Delta x_2 \le 2$$

It is clear from Figure 7.9 that without the move limits the solution is unbounded, which is not very helpful. Using the solution obtained in the figure, the next iteration

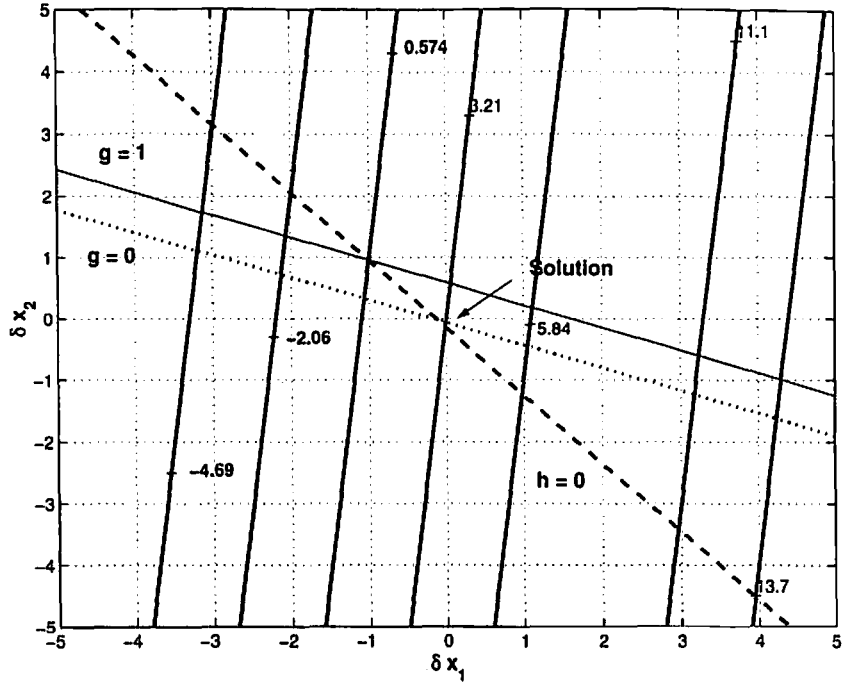

Figure 7.8 Example 7.1 SLP iteration 3.

can be executed. Once again the move limits will influence the solution. If the move limits are left at the same value, the solution will always be on the square with no possibility of convergence. These limits must be lowered with each iteration. Usually they are done geometrically through a scaling factor. This implies the convergence will depend on the strategy for changing the move limits, not an appealing situation. The SLP is not one of the popular methods. The move limits are avoided with the next method.

7.3.2 Sequential Quadratic Programming (SQP)

The lack of robustness in the SLP due to the need for move limits can be countered by including an element of nonlinearity in the problem. There are several ways this can be achieved. One way is to require that the search directions (remember in actual implementation $\Delta \mathbf{X}$ is $\mathbf{S}$) in Equations (7.31)–(7.34) be limited by the unit circle which can be described the constraint:

$$S^T S \le 1 \tag{7.35}$$

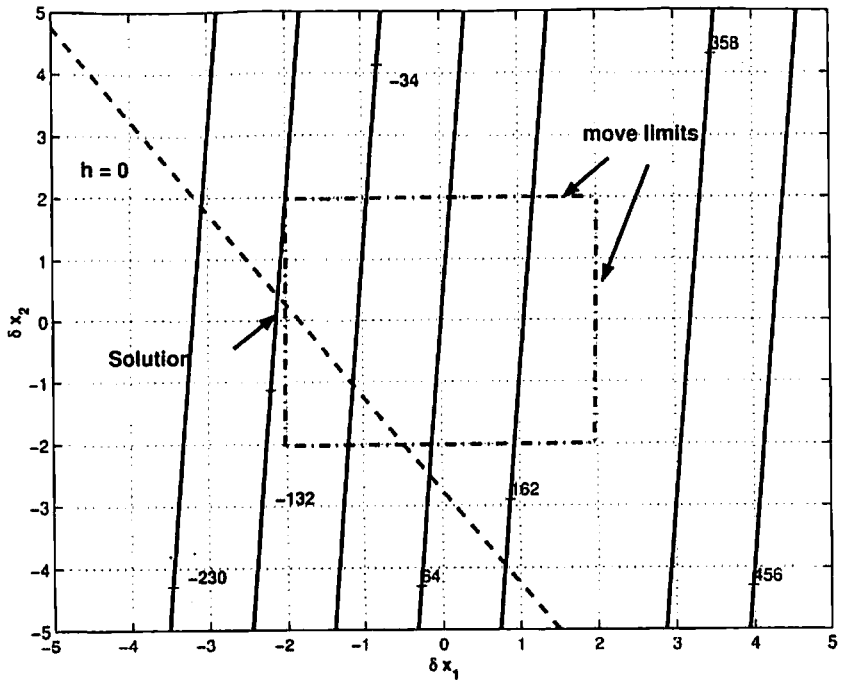

Figure 7.9 Illustration of move limits.

The stepsize determination should then account for the actual change in the design vector. The problem expressed by Equations (7.31)–(7.35) is a quadratic programming (QP) subproblem. An iterative solution of this subproblem can be one form of the SQP method. This method is popular and is used widely. While the QP subprogram is a simple nonlinear problem with linear constraints, it is still a multivariable nonlinear problem. Fortunately, QP problems appear in many disciplines and there are efficient procedures and algorithms to address these problems [6–10]. One of these methods is an extension of the Simplex method of Linear Programming.

The solution to the QP is well documented. It would require much discussion and therefore is not introduced here. Readers are referred to the various references. This is also the primary reason why the QP method is available in the Optimization Toolbox in MATLAB. The formal methods for QP problems are based on efficiency in the various stages of the implementation. An inefficient but successful solution to the problem can be obtained by using the ALM of the previous section to solve the QP subproblem. After all, computing resources are not a constraint today. In lieu of formal discussion, an intuitive understanding of the numerical method is attempted in this book. The translation of SQP into code is not trivial and the method is usually the basis of commercial software. For example, SQP is the only algorithm for constrained

optimum that MATLAB implements (as of this writing). Having access to the Optimization Toolbox allows NLP to be solved using SQP. A detailed discussion of the solution technique is also available [11].

The QP subproblem employed in this subsection is based on expanding the objective function quadratically about the current design. The constraints are expanded linearly as in SLP [8]. This is called *sequential quadratic programming* (SQP).

$$\text{Minimize} \quad \tilde{f}(\Delta\mathbf{X}) = f(\mathbf{X}_i) + \nabla f(\mathbf{X}_i)^{\mathrm{T}} \Delta\mathbf{X} + \frac{1}{2} \Delta\mathbf{X}^{\mathrm{T}} \nabla^2 f(\mathbf{X}_i) \Delta\mathbf{X} \tag{7.36}$$

$$\text{Subject to:} \quad \tilde{h}_k(\Delta\mathbf{X}): h_k(\mathbf{X}_i) + \nabla h_k^{\mathrm{T}}(\mathbf{X}_i)\Delta\mathbf{X} = 0; \quad k = 1, 2, \ldots, l \tag{7.32}$$

$$\tilde{g}_j(\Delta\mathbf{X}): g_j(\mathbf{X}_i) + \nabla g_j(\mathbf{X}_i)^{\mathrm{T}} \Delta\mathbf{X} \le 0; \quad j = 1, 2, \ldots, m \tag{7.33}$$

$$\Delta x_i^{\text{low}} \le \Delta x_i \le \Delta x_i^{\text{up}}; \quad i = 1, 2, \ldots, n \tag{7.34}$$

In Equation (7.36) $\nabla^2 f(\mathbf{X}_i)$ is the Hessian matrix. In actual implementation the real Hessian will not be used. Instead a metric [H] that is updated with each iteration is used. This is based on the success of the Variable Metric Methods (VMM) of the previous chapter. Several researchers [6,9] have shown that the BFGS update for the Hessian provides an efficient implementation of the SQP method. This QP is well posed and convex and should yield a solution. Solution to this subproblem is at the heart of the SQP method. A considerable amount of research has been invested in developing efficient techniques to handle this subproblem. In a formal implementation of the SQP the $\Delta\mathbf{X}$ in Equations (7.36) and (7.32)–(7.34) must be replaced by the search direction S. The QP for S also modifies the constraint equations so that a feasible direction can be found with respect to the current active constraints. Figure 7.10 illustrates the concern about moving linearly to a point on an active constraint. $\mathbf{X}^0$, the current design is on an active constraint. If the search direction $\mathbf{S}^1$ follows the linearized function at $\mathbf{X}^0$ (the tangent to the function at that point), any stepsize, along the search direction, however small, will cause the constraint to be violated. In order to determine a neighboring point that will still satisfy the constraints, a search direction slightly less than the tangent is used. This introduces a lack of consistency as the deviation from tangency becomes a matter of individual experience and practice. Experiments have suggested that 90–95% of the tangent is a useful figure although making it as close to 100% is recommended [2]. The search direction finding QP subproblem is

$$\text{Minimize} \quad \tilde{f}(\mathbf{S}) = f(\mathbf{X}_i) + \nabla f(\mathbf{X}_i)^{\mathrm{T}} \mathbf{S}^{\mathrm{T}}[\mathbf{H}]\mathbf{S} \tag{7.37}$$

$$\text{Subject to:} \quad \tilde{h}_k(\mathbf{S}): ch_k(\mathbf{X}_i) + \nabla h_k^{\mathrm{T}}(\mathbf{X}_i)\mathbf{S} = 0; \quad k = 1, 2, \ldots, l \tag{7.38}$$

$$\tilde{g}_j(\mathbf{S}): cg_j(\mathbf{X}_i) + \nabla g_j(\mathbf{X}_i)^{\mathrm{T}} \mathbf{S} \le 0; \quad j = 1, 2, \ldots, m \tag{7.39}$$

$$s_i^{\text{low}} \le s_i \le s_i^{\text{up}}; \quad i = 1, 2, \ldots, n \tag{7.40}$$

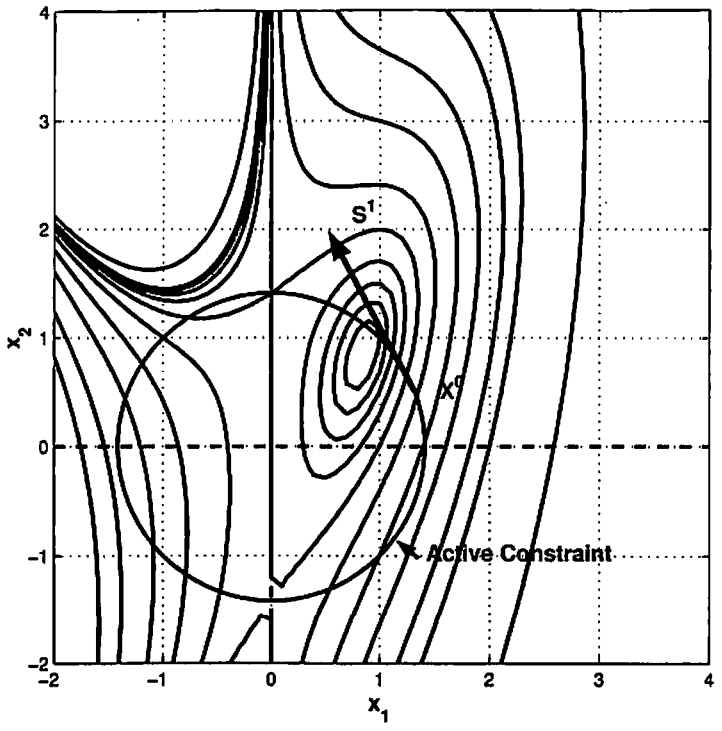

Figure 7.10 Moving linear to an active constraint.

The side constraints are not necessary as this is a well-posed problem. The factor c is the adjustment for moving tangent to the constraint. The suggested value for c is 0.9–0.95. For the inequality constraint, the value of c is 1 if the constraint is satisfied (should be able to move tangent to the constraint).

Stepsize Calculation: Once the search direction has been determined, it is necessary to calculate the stepsize. Since the methods of this chapter have seen an enormous increase in the number of calculations, the stepsize calculations are based on simultaneously decreasing the objective as well as improving the constraint satisfaction. In order to accomplish this goal a suitable function must be developed and the stepsize determination based on minimizing this unconstrained function. Several authors refer to it as the *descent function*. The description of this function varies from researcher to researcher. Some define this as the Lagrangian function in which case the Lagrange multipliers must be determined. Many others include only the constraints with a penalty multiplier and adopt the EPF form for the function. Still others include only the maximum violated constraint. Two forms of the function are listed below:

$$\mathbf{X}^i = \mathbf{X}^{i-1} + \alpha \mathbf{S} \tag{7.41}$$

$$\text{Minimize:} \quad \phi(\mathbf{X}^i) = f(\mathbf{X}^i) + r\sum_{k=1}^{l} h_k(\mathbf{X}^i)^2 + r\sum_{j=1}^{m} \max[g_j(\mathbf{X}^i), 0]^2 \tag{7.42}$$

$$\text{Minimize:} \quad \phi(\mathbf{X}^i) = f(\mathbf{X}^i) + \sum_{k=1}^{l} |\lambda_k h_k(\mathbf{X}^i)| + \sum_{j=1}^{m} \beta_j \max[g_j(\mathbf{X}^i), 0] \tag{7.43}$$

Equation (7.42) or (7.43) can be used to obtain the value g stepsize α. The function in Equation (7.43) requires an important observation. Should the Lagrange multipliers be calculated for each value of α? Imagine using the golden section to solve Equation (7.43) with the multipliers calculated from the satisfaction of the Kuhn-Tucker conditions for each new value of X. Several updating strategies for the multipliers in Equation (7.43) can be found in the listed references. A simple one is to hold the values of the multipiers constant at the previous iteration value and change them at the end of the iteration.

Calculating the Multipliers: The popularity of the SQP method is that it attempts to update the Hessian based on the Lagrangian of the problem. Calculating the multipliers is necessary to verify the Kuhn-Tucker conditions at the solution. Calculating the multipliers, for the current value of the design vector, is necessary for additional operations, including setting up the descent function in Equation (7.43), as well as updating the metric replacement for the Hessian (follows next). The calculation uses the FOC based on the Lagrangian. The values of the gradients of the various functions are computed at the current design. The multipliers are then obtained as a solution to a set of linear equations. The solutions to QP and SQP problems usually take advantage of the *active constraint set*. This includes all equality constraints and only those inequality constraints that are active. This implies that SQP iterations *start at a feasible point*. The multipliers corresponding to inactive constraints are set to zero. The remaining multipliers can be established considering a set of an appropriate number of linear independent equations from Equation (7.12):

$$\frac{\partial F}{\partial x_i} = \frac{\partial f}{\partial x_i} + \lambda_1 \frac{\partial h_i}{\partial x_i} + \cdots + \lambda_l \frac{\partial h_l}{\partial x_i} + \beta_1 \frac{\partial g_1}{\partial x_i} + \cdots + \beta_m \frac{\partial g_m}{\partial x_i} = 0; \quad i = 1, 2, \ldots, n \tag{7.12}$$

Replacing Hessian with BFGS Update: In Equation (7.37), calculating the actual Hessian is strongly discouraged. Since the BFGS method converges to the Hessian, the BFGS metric is considered to be an excellent replacement of the Hessian [9]. The metric is based on the Lagrangian of the function [12]:

$$\mathbf{p} = \nabla f(\mathbf{X}^{q+1}) + \sum_{k=1}^{l} \lambda_k \nabla h_k(\mathbf{X}^{q+1}) + \sum_{j=1 \forall j \in J}^{m} \beta_j \nabla g_j(\mathbf{X}^{q+1})$$

$$-\nabla f(\mathbf{X}^{q}) + \sum_{k=1}^{l} \lambda_k \nabla h_k(\mathbf{X}^{q}) + \sum_{j=1 \forall j \in J}^{m} \beta_j \nabla g_j(\mathbf{X}^{q}) \tag{7.44}$$

$$\mathbf{y} = \mathbf{X}^{q+1} - \mathbf{X}^{q}$$

$$[\mathbf{H}^{q+1}] = [\mathbf{H}^{q}] + \frac{\mathbf{p}\mathbf{p}^{\mathrm{T}}}{\mathbf{p}^{\mathrm{T}}\mathbf{y}} - \frac{[\mathbf{H}^{q}]^{\mathrm{T}}[\mathbf{H}^{q}]}{\mathbf{y}^{\mathrm{T}}[\mathbf{H}^{q}]\mathbf{y}}$$

$j \in J$ implies the set of active inequality constraints. It is recommended that the metric be kept positive definite as much as possible even if it is likely that it will not be positive definite at the solution. One way of ensuring this is to verify that

$$\mathbf{p}^{\mathrm{T}}\mathbf{y} > 0$$

for each iteration.

Algorithm: Sequential Quadratic Programming (SQP) (A7.5)

Step 1. Choose $\mathbf{X}^1$, N_s (no. of iterations),
ε_i's (for convergence and stopping)
$q = 1$ (iteration counter)

Step 2. Call QP to optimize (7.37)–(7.40) ($\mathbf{X}_I = \mathbf{X}^q$)
Output: $\mathbf{S}$
Use a constrained α^* calculation
$\Delta\mathbf{X} = \alpha^* \mathbf{S}$
$\mathbf{X}^{q+1} = \mathbf{X}^q + \Delta\mathbf{X}$

Step 3. Convergence for SQP
If $h_k = 0$, for $k = 1, 2, \ldots, l$;
If $g_j \le 0$, for $j = 1, 2, \ldots, m$;
If KT conditions are satisfied
Then Converged, Stop
Stopping Criteria:
$\Delta\mathbf{X} = \mathbf{X}^{q+1} - \mathbf{X}^q$
If $\Delta\mathbf{X}^{\mathrm{T}}\Delta\mathbf{X} \le \varepsilon_1$: Stop (design not changing)
If $q = N_s$: Stop (maximum iterations reached)
Continue
Update Metric [H]
$q \leftarrow q + 1$
Go to Step 2

Application of SQP: The SQP technique is applied to Example 7.1. There are several ways to accomplish this. The application can use graphical support as shown with the SLP (this is an exercise). The application can use the QP program from MATLAB (also an exercise). In this book the QP problem is solved by applying the KT conditions—which is the feature of this subsection. After all for this specific example the functions are analytically defined and their structure is simple. We can harness the symbolic operations in MATLAB to calculate the derivatives and the actual Hessian. As in the case of SLP, the actual change in ΔX is computed [Equations (7.36), (7.32)–(7.34)] rather than the search direction S for the QP subproblem.

Sec7_3_2.m applies the SQP method symbolically to Example 7.1. It identifies the solution using the Lagrange method (Chapter 5). It prompts for the current design vector. It solves the QP analytically applying the KT conditions and providing the results for the two cases. Case a corresponds to $\beta = 0$ (g must be less than zero). Case b corresponds to $g = 0$ ($\beta > 0$). The output from running the program is recorded below for several iterations in this implementation of SQP. Analytical solution with user intervention is used. **Sec7_3_2.m** must be run for each iteration. You should be able to copy and paste design vector in the command window.

Iteration 1: $\mathbf{X}^1 = [3\ \ 2]$

$$f = 64; \quad \nabla f = \begin{bmatrix} 92 \\ -6 \end{bmatrix}; \quad [\mathbf{H}] = \begin{bmatrix} 102 & -8 \\ -8 & 6 \end{bmatrix}$$

$$h = 11; \quad \nabla h = \begin{bmatrix} 6 \\ 4 \end{bmatrix}; \quad g = 4.25; \quad \nabla g = \begin{bmatrix} 1.5 \\ 3 \end{bmatrix}$$

The solution to the QP problem is Case a:

$$\Delta x_1 = -1.0591; \quad \Delta x_2 = -1.1613$$

$$\tilde{h} = 0; \quad \tilde{g} = -0.8225;$$

$$\tilde{x}_1 = 1.9409; \quad \tilde{x}_2 = 0.8387$$

Iteration 2: $\mathbf{X}^2 = [1.9409 \quad 0.8387]$

$$f = 13.123; \quad \nabla f = \begin{bmatrix} 25.32 \\ -4.2785 \end{bmatrix}; \quad [\mathbf{H}] = \begin{bmatrix} 43.85 & -6.0862 \\ -6.0862 & 3.8818 \end{bmatrix}$$

$$h = 2.4705; \quad \nabla h = \begin{bmatrix} 3.8818 \\ 1.6774 \end{bmatrix}; \quad g = 0.4693; \quad \nabla g = \begin{bmatrix} 0.9704 \\ 1.2580 \end{bmatrix}$$

The solution to the QP problem is Case a:

$$\Delta x_1 = -0.6186; \quad \Delta x_2 = -0.0041$$

$$\tilde{h} = 0; \quad \tilde{g} = -1.8281;$$

$$\bar{x}_1 = 1.3222; \quad \bar{x}_2 = 0.7975$$

Iteration 3: $\mathbf{X}^3 = [1.3222 \quad 0.7975]$

$$f = 4.2126; \quad \nabla f = \begin{bmatrix} 6.3087 \\ -1.3875 \end{bmatrix}; \quad [\mathbf{H}] = \begin{bmatrix} 19.789 & -3.6938 \\ -3.6938 & 2.6444 \end{bmatrix}$$

$$h = 0.3843; \quad \nabla h = \begin{bmatrix} 2.6444 \\ 1.5951 \end{bmatrix}; \quad g = -0.0858; \quad \nabla g = \begin{bmatrix} 0.6611 \\ 1.1963 \end{bmatrix}$$

The solution to the QP problem is Case a:

$$\Delta x_1 = -0.2685; \quad \Delta x_2 = +0.2042$$

$$\bar{h} = 0; \quad \bar{g} = -0.0190;$$

$$\bar{x}_1 = 1.0537; \quad \bar{x}_2 = 1.0018$$

Iteration 4: $\mathbf{X}^4 = [1.0537 \quad 1.0018]$

$$f = 3.0685; \quad \nabla f = \begin{bmatrix} 1.5682 \\ -1.0937 \end{bmatrix}; \quad [\mathbf{H}] = \begin{bmatrix} 11.316 & -2.2112 \\ -2.2112 & 2.1074 \end{bmatrix}$$

$$h = 0.1138; \quad \nabla h = \begin{bmatrix} 2.1074 \\ 2.0036 \end{bmatrix}; \quad g = -0.0302; \quad \nabla g = \begin{bmatrix} -0.1093 \\ 1.5027 \end{bmatrix}$$

The solution to the QP problem is Case b:

$$\Delta x_1 = -0.0523; \quad \Delta x_2 = -0.0017; \quad \beta = 0.9275$$

$$\bar{h} = 0; \quad \bar{g} = 0;$$

$$\bar{x}_1 = 1.0014; \quad \bar{x}_2 = 1.0000;$$

$$f = 3.0014; \quad h = 0.0027; \quad g = 0.0006$$

The solution is almost converged and further iterations are left to the reader. The next iteration should produce values for convergence.

The SQP here takes five iterations to converge. This was based on computing the actual Hessian. The SLP took four. The ALM took five. For Example 7.1 the SQP does not appear to have any significant advantage, which is unexpected. The ALM starts to appear quite attractive. The SQP in **Sec7_3_2.m** is implemented analytically based on KT conditions. It is very different than Algorithm (A7.5) which represents a numerical implementation.

7.3.3 Generalized Reduced Gradient (GRG) Method

The GRG method is another popular technique for constrained optimization. The original method, the Reduced Gradient Method [13], has seen several different customizations due to several researchers [14]. Like the SQP this is an active set method—*deals with active inequalities*—hence the method includes equality constraints only. The inequality constraints are transformed to equality constraints using a linear slack variable of the type used in LP problems. The general optimization problem is restructured as

$$\text{Minimize} \quad f(x_1, x_2, \ldots, x_n) \tag{7.45}$$

$$\text{Subject to:} \quad h_k(x_1, x_2, \ldots, x_n) = 0, \quad k = 1, 2, \ldots, l \tag{7.46}$$

$$g_j(x_1, x_2, \ldots, x_n) + x_{n+j} = 0, \quad j = 1, 2, \ldots, m \tag{7.47}$$

$$x_i^l \le x_i \le x_i^u \quad i = 1, 2, \ldots, n \tag{7.48}$$

$$x_{n+j} \ge 0 \quad j = 1, 2, \ldots, m \tag{7.49}$$

The total number of variables is $n + m$. The active set strategy assumes that it is possible to define or choose a subset of $(n - l)$ independent variables from the set of $(l + m)$ equality constraints. Once these variables are established, the dependent variables $(l + m)$ can be recovered through the constraints. The number of independent variables is based on the original design variables and the original equality constraints of the problem. The following development is adapted from Reference 2.

The set of the design variables [X] is partitioned into the independent set [Z] and dependent set [Y].

$$\mathbf{X} = \begin{bmatrix} \mathbf{Z} \\ \mathbf{Y} \end{bmatrix}$$

The constraints are accumulated as

$$\mathbf{H} = \begin{bmatrix} \mathbf{h} \\ \mathbf{g} \end{bmatrix}$$

Relations (7.45)–(7.49) can be recast as

$$\text{Minimize} \quad f(\mathbf{Z}, \mathbf{Y}) \tag{7.50}$$

$$\text{Subject to:} \quad \mathbf{H}(\mathbf{Z}, \mathbf{Y}) = 0 \tag{7.51}$$

$$z_i^l \le z_i \le z_i^u \quad i = 1, 2, \ldots, n - l \tag{7.52}$$

A similar set of side constraints for $\mathbf{Y}$ can be expressed. In actual implementation the inactive inequality constraints are not included in the problem. Also, the slack variables can belong to the independent set. In GRG, the linearized objective is minimized subject to the linearized constraints about the current design $\mathbf{X}^i$.

$$\text{Minimize } \tilde{f}(\Delta\mathbf{Z}, \Delta\mathbf{Y}) = f(\mathbf{X}^i) + \nabla_Z f(\mathbf{X}^i)^T \Delta\mathbf{Z} + \nabla_Y f(\mathbf{X}^i)^T \Delta\mathbf{Y} \tag{7.53}$$

$$\text{Subject to: } \tilde{H}_k(\Delta\mathbf{Z}, \Delta\mathbf{Y}) = \mathbf{H}_k(\mathbf{X}^i) + \nabla_Z \mathbf{H}_k(\mathbf{X}^i)^T \Delta\mathbf{Z} + \nabla_Y \mathbf{H}_k(\mathbf{X}^i)^T \Delta\mathbf{Y} = 0 \tag{7.54}$$

If the GRG began at a feasible point, the $\mathbf{H}(\mathbf{X}^i) = 0$, the linearized constraints are also zero, then the linearized equations in $[\mathbf{H}]$ determine a set of linear equations in $\Delta\mathbf{Z}$ and $\Delta\mathbf{Z}$. Considering the vector $[\mathbf{H}]$ the linear equation can be written as

$$[\mathbf{A}]\Delta\mathbf{Z} + [\mathbf{B}]\Delta\mathbf{Y} = 0 \tag{7.55}$$

where
$$[\mathbf{A}] = \begin{bmatrix} \nabla_Z \mathbf{H}_1^T \\ \nabla_Z \mathbf{H}_2^T \\ \vdots \\ \nabla_Z \mathbf{H}_{l+m}^T \end{bmatrix}; \quad [\mathbf{B}] = \begin{bmatrix} \nabla_Y \mathbf{H}_1^T \\ \nabla_Y \mathbf{H}_2^T \\ \vdots \\ \nabla_Y \mathbf{H}_{l+m}^T \end{bmatrix}$$

From Equation (7.55) the change in the dependent variable $\Delta\mathbf{Y}$ can be expressed in terms of the changes in the independent variables as

$$\Delta\mathbf{Y} = -[\mathbf{B}]^{-1} [\mathbf{A}] \Delta\mathbf{Z} \tag{7.56}$$

Substituting Equation (7.56) into Equation (7.53) reduces it to an unconstrained problem in $\Delta\mathbf{Z}$:

$$\begin{aligned} \tilde{f}(\Delta\mathbf{Z}) &= f(\mathbf{X}^i) + \nabla_Z f(\mathbf{X}^i)^T \Delta\mathbf{Z} - \nabla_Y f(\mathbf{X}^i)^T [\mathbf{B}]^{-1} [\mathbf{A}] \Delta\mathbf{Z} \\ &= f(\mathbf{X}^i) + \left\{ \nabla_Z f(\mathbf{X}^i)^T - \nabla_Y f(\mathbf{X}^i)^T [\mathbf{B}]^{-1} [\mathbf{A}] \right\} \Delta\mathbf{Z} \qquad (7.57) \\ &= f(\mathbf{X}^i) + \left\{ \nabla_Z f(\mathbf{X}^i) - \left([\mathbf{B}]^{-1} [\mathbf{A}] \right)^T \nabla_Y f(\mathbf{X}^i) \right\}^T \Delta\mathbf{Z} \\ &= f(\mathbf{X}^i) + [\mathbf{G}_R]^T \Delta\mathbf{Z} \end{aligned}$$

$[\mathbf{G}_R]$ is the *reduced gradient* of the function $f(\mathbf{X})$. It provides the *search direction*. The stepsize α is found by requiring that the changes in $\Delta\mathbf{Y}$, corresponding to the changes in $\Delta\mathbf{Z}$ (α S), determine a feasible design. This is implemented through Newton's method as described below. Since the functions are linear, there is no quadratic

convergence associated with this choice. The iterations start with a feasible current design vector $[\mathbf{X}^i] = [\mathbf{Z}^{iT}\ \mathbf{Y}^{iT}]$

Stepsize Computation: Algorithm (GRG)

Step 1. Find $\mathbf{G}_R$, $\mathbf{S} = -\mathbf{G}_R$; Select α

Step 2. $q = 1$

$\Delta\mathbf{Z} = \alpha\mathbf{S}$; $\mathbf{Z} = \mathbf{Z}^i + \Delta\mathbf{Z}$;

$\Delta\mathbf{Y}^q = -[\mathbf{B}]^{-1}[\mathbf{A}]\Delta\mathbf{Z}$

Step 3. $\mathbf{Y}^{q+1} = \mathbf{Y}^q + \Delta\mathbf{Y}^q$

$$\mathbf{X}^{i+1} = \begin{bmatrix} \mathbf{Z} \\ \mathbf{Y}^{q+1} \end{bmatrix}$$

If $[\mathbf{H}(\mathbf{X}^{i+1})] = 0$; Stop, Converged

Else $q \leftarrow q + 1$

$\Delta\mathbf{Y}^q = [\mathbf{B}]^{-1} [-\mathbf{H}(\mathbf{X}^{i+1})]$

Go To Step 3

In the above algorithm, neither $\mathbf{Z}$, $[\mathbf{A}]$, $[\mathbf{B}]$, nor α is changed. The singularity of $[\mathbf{B}]$ is a concern and can be controlled by the selection of independent variables. It appears that $\Delta\mathbf{Z}$ is responsible for decrease in the function while $\Delta\mathbf{Y}$ is responsible for feasibility. The above algorithm is usually executed for two values of α. A quadratic interpolation between the three points ($\alpha = 0$ is available from the previous iteration) is used to determine the optimum stepsize. One final iteration is performed for this optimum stepsize.

Algorithm: Generalized Reduced Gradient (GRG) (A7.6)

Step 1. Choose $\mathbf{X}^1$ (must be feasible)

N_s (no. of iterations),

ε_i's (for convergence and stopping)

$p = 1$ (iteration counter)

Step 2. Identify $\mathbf{Z}$, $\mathbf{Y}$

Calculate $[\mathbf{A}]$, $[\mathbf{B}]$

Calculate $[\mathbf{G}_R]$

Calculate Optimum Stepsize α^* (see algorithm)

Calculate $\mathbf{X}^{p+1}$

Step 3. Convergence for GRG

If $h_k = 0$, for $k = 1, 2, \ldots, l$;

If $g_j \le 0$, for $j = 1, 2, \ldots, m$;

If KT conditions are satisfied

Then Converged, Stop

Stopping Criteria:

$\Delta \mathbf{X} = \mathbf{X}^{p+1} - \mathbf{X}^{p}$

If $\Delta \mathbf{X}^{\mathrm{T}} \Delta \mathbf{X} \leq \varepsilon_1$: Stop (design not changing)

If $p = N_s$: Stop (maximum iterations reached)

Continue

$p \leftarrow p + 1$

Go to Step 2

Application of the GRG Method: The GRG algorithm is applied to Example 7.1. Once again the illustration is through a set of calculations generated using MATLAB. For a start Example 7.1 is rewritten using a slack variable for the inequality constraint.

$$\begin{aligned} \text{Minimize} \quad & f(x_1, x_2, x_3): \quad x_1^4 - 2x_1^2 x_2 + x_1^2 + x_1 x_2^2 - 2x_1 + 4 \\ \text{Subject to:} \quad & H_1(x_1, x_2, x_3): \quad x_1^2 + x_2^2 - 2 = 0 \\ & H_2(x_1, x_2): \quad 0.25x_1^2 + 0.75x_2^2 - 1 + x_3 = 0 \\ & 0 \leq x_1 \leq 5; \quad 0 \leq x_2 \leq 5; \quad x_3 \geq 0 \end{aligned}$$

For this problem $n = 2$, $l = 1$, $m = 1$, $n + m = 3$, $n - l = 1$, $l + m = 2$.

Example 7.1 is not a good problem to illustrate the working of the GRG method because the only feasible point where g is active is also the solution. This makes $\mathbf{Z}$, and therefore $\mathbf{G}_R$, a scalar. The example is retained because of uniformity. In the implementation below, $\mathbf{Z} = [x_1]$ and $\mathbf{Y} = [x_2 \quad x_3]^{\mathrm{T}}$.

Sec7_3_3.m: This m-file handles Example 7.1 through the GRG method symbolically. It requests a starting design which should be feasible. It sets up the $\mathbf{Z}$ and $\mathbf{Y}$ automatically and computes [**A**] and [**B**]. Experimenting with the code indicates that not all feasible points provide a nonsingular [**B**]. It then prompts the user for two values of α which it uses to perform a quadratic interpolation. It computes an optimum stepsize and completes the iteration for $\Delta \mathbf{Y}$. This completes one iteration of the GRG. Experimenting with the code illustrates that the choices of α need some thought for the iteration to succeed. The following is transcribed from the Command window for two iterations. Note the prompts for alpha:

Iteration 1

```
» Sec7_3_3
Input the feasible design chosen for evaluation.
[1.2 0.74833]          % This is X1

The design vector [ 1.2000 0.7483 0.2200]

function and constraints(f h1 h2):
 3.6304e+000 -2.2111e-006 0
```



```
A =
 2.4000e+000
 6.0000e-001
B =
 1.4967e+000 0
 1.1225e+000 1.0000e+000
S =
 -6.0183e+000    % (Reduced Gradient)

Input the stepsize for evaluation.
          % (Prompt for first stepsize)
0.1
alpha =
 1.0000e-001

No. of iterations and constraint error: 29
 1.0000e-001 8.2187e-009

design vector: 5.9817e-001 1.2815e+000 -3.2110e-001

function and constraints (f h1 h2)
 3.3548e+000 7.2526e-005 5.4394e-005
S =
 -6.0183e+000

Input the stepsize for evaluation.
                % (Prompt for second stepsize)
0.05
alpha =
 5.0000e-002

No. of iterations and constraint error: 12
 5.0000e-002 6.2635e-009

design vector: 8.9909e-001 1.0916e+000 -9.5823e-002

function and constraints (f h1 h2)
 2.9702e+000 -6.3314e-005 -4.7485e-005
alpha =
 5.6595e-002      % (quadratically interpolated α)
S =
 -6.0183e+000

No. of iterations and constraint error: 14
 5.6595e-002 4.3237e-009

design vector: 8.5940e-001 1.1231e+000 -1.3072e-001

function and constraints (f h1 h2)
 2.9903e+000 -5.2604e-005 -3.9453e-005
```

This completes one iteration. This is far from satisfactory as x_3, the slack variable, is *negative*. The stepsize and design change calculation must be made sensitive to the side constraints. In actual implementation of the GRG, since $g(\mathbf{X})$ was *inactive* at the start, it would not have been included in the calculations at all. Using the solution to start the next additional iterations can be obtained leading to the solution.

Iteration 2

Rerun **Sec7_3_3.m** with $\mathbf{X}^2$ = [8.5940e-001 1.1231e+000]

The rest of the iterations are left as an exercise for the reader. To complete **Sec7_3_3.m** the calculations for KT conditions must be included. In the above, since $g(\mathbf{X})$ is in violation, the KT conditions were not computed. This is currently one of the more robust methods. For this particular example the progress of solution is quite sensitive to the stepsize chosen for interpolation.

7.3.4 Sequential Gradient Restoration Algorithm (SGRA)

The SGRA is a two-phase approach to the solution of NLP problems. Starting at a feasible design, the Gradient phase decreases the value of the objective function while satisfying the linearized active constraints. This will cause constraint violation for nonlinear active constraints. The Restoration phase brings back the design to feasibility which may establish a new and different set of active constraints. This cycle of two phases is repeated until the optimum is found [15, 16]. The method incorporates a descent property with each cycle. Like other algorithms in this section, it uses an active constraint strategy. In comparison the GRG combines the two phases together. The method has been favorably compared to GRG and the Gradient Projection method [17].

The SGRA as originally introduced in the listed references assumes the inequality constraints of the form

$$g_j(\mathbf{X}) \geq 0; \quad j = 1, 2, \ldots, m$$

To be consistent with the discussion in the book it is changed to the standard form of NLP problems as defined in this text. In the following development of the algorithm, only the salient features are indicated. They are also transcribed consistent with the other algorithms in this section. It is recommended that the reader review the references for a detailed description. The general problem is

$$\text{Minimize} \quad f(\mathbf{X}), \quad [\mathbf{X}]_n \tag{7.5}$$

$$\text{Subject to:} \quad [h(\mathbf{X})]_l = 0 \tag{7.6}$$

$$[g(\mathbf{X})]_m \leq 0 \tag{7.7}$$

$$\mathbf{X}^{\text{low}} \leq \mathbf{X} \leq \mathbf{X}^{\text{up}} \tag{7.8}$$

In the SGRA only active inequality constraints are of interest. Equality constraints are always active. Active inequality constraints also include violated constraints. If ν indicates the set of active constraints, then Equations (7.6) and (7.7) are combined into a vector of active constraints (ϕ):

$$\phi(\mathbf{X}) = \begin{bmatrix} \mathbf{h}(\mathbf{X}) \\ \mathbf{g}(\mathbf{X}) \geq 0 \end{bmatrix}_\nu \tag{7.58}$$

The number of active inequality constraints is $\nu - l$. The Lagrangian for the problem can be expressed in terms of the active constraints alone [since the multipliers for the $g_j(\mathbf{X}) < 0$ will be set to zero as part of KT conditions]. The KT conditions are then expressed as

$$\nabla_X F(\mathbf{X}, \lambda^\nu) = \nabla_X f(\mathbf{X}) + [\nabla_X \phi]\lambda^\nu \tag{7.59a}$$

$$\phi(\mathbf{X}) = 0 \tag{7.59b}$$

$$\lambda_{\nu-l} \geq 0 \tag{7.59c}$$

where $[\nabla_X \phi] = [\nabla_X \phi_1 \quad \nabla_X \phi_2 \quad \ldots \quad \nabla_X \phi_\nu]$; $\lambda^\nu = [\lambda_1 \quad \lambda_2 \quad \ldots \quad \lambda_\nu]^T$

Gradient Phase: Given a feasible design $\mathbf{X}^i$, find the neighboring point

$$\tilde{\mathbf{X}}_g = \mathbf{X}^i + \Delta\mathbf{X}$$

such that $\delta f < 0$ and $\delta\phi = 0$. Imposing a quadratic constraint on the displacement $\Delta\mathbf{X}$ [16], the problem can be set up as an optimization subproblem whose KT conditions determine

$$\Delta\mathbf{X} = -a\nabla F_x(\mathbf{X}^i, \lambda^\nu) = a\mathbf{S} \tag{7.60}$$

The interpretation of the search direction $\mathbf{S}$ is used to connect to the other algorithms in this book. It is not in the original development. The search direction is opposite to the gradient of the Lagrangian, which is a novel feature. To compute the Lagrangian, the Lagrange multipliers have to be calculated. This is done by solving a system of ν linear equations

$$[\nabla_X \phi(\mathbf{X}^i)]^T \nabla_X f(\mathbf{X}^i) + [\nabla_X \phi(\mathbf{X}^i)]^T [\nabla_X \phi(\mathbf{X}^i)]\lambda^\nu = 0 \tag{7.61}$$

Stepsize for Gradient Phase: The stepsize a calculation is based on driving the optimality conditions in Equation (7.59a) to zero. Therefore, if

$$\tilde{\mathbf{X}}_g = \mathbf{X}^i - \alpha \nabla F_X(\mathbf{X}^i, \lambda^\nu)$$

then optimum α^* is found by cubic interpolation while trying to satisfy

$$\nabla_X F(\tilde{\mathbf{X}}_g, \lambda^\nu)^T \nabla_X F(\mathbf{X}^i, \lambda^\nu) = 0 \tag{7.62}$$

Care must be taken that this stepsize does not cause significant constraint violation. This is enforced by capping the squared error in the constraints by a suitable upper bound which is set up as

$$\phi(\tilde{\mathbf{X}}_g)^T \phi(\tilde{\mathbf{X}}_g) \le P_{\max} \tag{7.63}$$

Reference 16 suggests that P_{max} is related to another performance index Q, which is the error in the optimality conditions:

$$Q = \nabla F_X(\mathbf{X}^i, \lambda^\nu)^T \nabla F_X(\mathbf{X}^i, \lambda^\nu) \tag{7.64}$$

Restoration Phase: It is expected that at the end of the gradient phase the function will have decreased but there would be some constraint dissatisfaction (assuming there was at least one nonlinear active constraint at the beginning of the gradient phase). The restoration phase establishes a neighboring feasible solution. It does this by ensuring that the linearized constraints are feasible. Prior to this, the active constraint set has to be updated ($\bar{\nu}$) since previously feasible constraints could have become infeasible (and previously infeasible constraints could have become feasible) with the design change caused by the gradient phase. The design vector and the design changes for this phase can be written as

$$\tilde{\mathbf{X}}_r = \tilde{\mathbf{X}}_g + \Delta \mathbf{X}_r \tag{7.65}$$

The design change for this phase $\Delta \mathbf{X}_r$ is obtained as a least square error in the design changes subject to the satisfaction of the linear constraints. Setting up an NLP subproblem this calculation for the design changes is

$$\Delta \mathbf{X}_r = -\nabla_X \phi(\tilde{\mathbf{X}}_g) \sigma^{\bar{\nu}} \tag{7.66}$$

Here $\sigma^{\bar{\nu}}$ is the $\bar{\nu}$ vector Lagrange multiplier of the quadratic subproblem. The values for the multipliers are established through the linear equations

$$\mu \phi(\tilde{\mathbf{X}}_g) - \nabla_X \phi(\tilde{\mathbf{X}}_g)^T \nabla_X \phi(\tilde{\mathbf{X}}_g) \sigma^{\bar{\nu}} = 0 \tag{7.67}$$

The factor μ is a user-controlled parameter to discourage large design changes.

The Restoration phase is iteratively applied until

$$\phi(\tilde{\mathbf{X}}_r)^T \phi(\tilde{\mathbf{X}}_r) \le \varepsilon_1 \tag{7.68}$$

where ε_1 is a small number.

At the conclusion of the restoration phase the constraints are feasible and the next cycle of Gradient–Restoration phase can be applied again. A summary of the above steps is captured in the algorithm given below.

Algorithm: Sequential Gradient Restoration Algorithm (SGRA) (A7.7)

Step 1. Choose $\mathbf{X}^1$ (must be feasible)
  N_s (no. of iterations),
  ε_i's (for convergence and stopping)
  $p = 1$ (iteration counter)

Step 2. Execute Gradient Phase
  Calculate Stepsize using cubic interpolation
  Calculate $\mathbf{X}_g^{p+1}$

Step 3. Execute Restoration Phase
  $\mathbf{X}^{p+1}$

Step 4. Convergence for SGRA
  If $h_k = 0$, for $k = 1, 2, \ldots, l$;
    If $g_j \le 0$, for $j = 1, 2, \ldots, m$;
      If KT conditions are satisfied
      Then Converged, Stop
  Stopping Criteria:
    $\Delta \mathbf{X} = \mathbf{X}^{p+1} - \mathbf{X}^p$
    If $\Delta \mathbf{X}^T \Delta \mathbf{X} \le \varepsilon_1$: Stop (design not changing)
  If $p = N_s$: Stop (maximum iterations reached)
    Continue
    $p \leftarrow p + 1$
    Go to Step 2

Application of the SGRA (A7.7): Example 7.1 is used to test the application of the SGRA. The m-file **Sec7_3_4.m** contains the code that applies the SGRA through a combination of symbolic and numeric calculations. As in the previous examples it is customized for two variables, one equality and one inequality constraint. Unlike the GRG implementation it incorporates the *active constraint strategy*. It executes one cycle of the SGRA which has a single iteration in the Gradient phase and several iterations in the Restoration phase. In the gradient phase the cubic interpolation of the original technique is replaced by the quadratic interpolation used everywhere in the book. The starting input is a feasible design vector. In the gradient phase the method

requests a value for the stepsize to scan for the optimum α^* using quadratic interpolation. This is important for successive cycles start at the optimum value of the stepsize for the previous cycle. The value of μ in Equation (7.67) is set at 0.5. Printing is controlled through the semicolon and the reader is encouraged to follow details by removing it.

In the following the program is run with the starting design used for the GRG method. The output from the Command window is pasted below with some bold annotations to draw attention.

```
» Sec7_3_4
Input the feasible design chosen for evaluation.
[1.2 0.74833]    (feasible design vector input)

The design vector [ 1.2000 0.7483]

function and constraints(f h g):
 3.6304e+000 -2.2111e-006 -2.2000e-001
**********************
 Gradient Phase
**********************
The Lagrange multipliers
 -1.0812e+000

Input the stepsize for evaluation.
1                (alpha used to start scanning process)

alpha for quadratic interpolation
 0 1.2500e-001 2.5000e-001
          (alpha that brackets the minimum)
function values for quadratic interpolation
 1.0142e+001 6.6291e+000 1.1300e+001
          (corresponding function value)
optimum alpha: 1.1615e-001

The design vector [ 1.0043 1.0622]

function and constraints(f h1 h2):
 3.0077e+000 1.3681e-001 9.8328e-002
          (both constraints violated)
The performance indices Q, P :
 6.3181e+000 1.8718e-002
    (P measures the squared error in the constraints)
**********************
 Restoration Phase
**********************
 Number of Restoration iterations: 21

The design vector [ 1.0000 1.0000]
```



```
function and constraints(f h1 h2):
 3.0000e+000 6.7129e-005 4.6741e-005
          (Constraints satisfied - g active)
The performance index P :
 6.6911e-009
```

The solution has converged in *one iteration*. The SGRA is of the same caliber as the GRG (some comparisons show it to be better). It is difficult to compare different algorithms because of different coding structures and nonstandard implementation details. Nevertheless the SGRA has considerable merit. It is definitely better than the GRG as far as Example 7.1 is concerned (Section 7.3.3). This comparison may not be fair since the active constraint strategy was not used for the GRG even though it is part of the GRG. This demonstrates the need for such a strategy in all algorithms that deal with constrained optimization.

7.4 ADDITIONAL EXAMPLES

Three additional examples are introduced here to illustrate and expand the use of these techniques. During the early development of these techniques there was a significant emphasis on efficiency, robustness, and keeping computational resources as low as possible. Most of the techniques were developed on mainframes using FORTRAN. Today the search for global solutions (that are largely heuristic), coupled with simple calculations repeated almost endlessly, performed on desktop PCs with incredible computing power, through software systems that provide a wide range of resources has shifted the focus to experimentation and creativity. The author feels such an effort must be coupled with knowledge and insight gained through a wide selection of applications and understanding the progress of numerical calculation. In spite of all the exposure so far in the text, for many real design problems the solution is not automatic. Dealing with nonlinear equations is a challenge and often surprising. Trying to understand why the technique does not work is as much a learning experience as a routine solution to the problem. Since these examples use the methods of the previous section with different functions, all of the code developed before is revised with and the appropriate changes and renamed. These modifications require that few lines be changed from the original code. This allows the user to make incremental changes as needed. This is not an efficient approach but capitalizes on the unlimited hard drive space available today. Students are urged to understand the algorithm, the flow of the code, and the result being sought.

7.4.1 Example 7.2—Flagpole Problem

This is the same problem which was solved graphically in Chapter 2, analytically in Chapter 4, and now considered for numerical solution. In this subsection the scaled version of the problem is solved. From Chapter 4, Example 4.3, it is reintroduced as

Example 7.2

$$\text{Minimize} \quad f = x_1^2 - x_2^2 \tag{7.69}$$

$$g_1: \ 2.5509x_1^2 + 0.1361x_1 - 33.148(x_1^4 - x_2^4) \le 0 \tag{7.70a}$$

$$g_2: \ (0.0488 + 1.4619x_1)(x_1^2 + x_1x_2 + x_2^2) - 166.5614(x_1^4 - x_2^4) \le 0 \tag{7.70b}$$

$$g_3: \ 1.0868x_1 + 0.3482 - 28.4641(x_1^4 - x_2^4) \le 0 \tag{7.70c}$$

$$g_4: \ x_2 - x_1 + 0.001 \le 0 \tag{7.70d}$$

$$0.02 \le x_1 \le 1.0; \quad 0.02 \le x_2 \le 1.0 \tag{7.71}$$

Insight about Numerical Solution: In Chapter 4 it was investigated that the solution does not satisfy the KT conditions because the solution is not a regular point. It was ascertained that the gradients of the two active constraints that determine the solution (g_1, g_3) are nearly parallel in the neighborhood of the solution. How will this affect the numerical search for the solution?

Typically what a numerical technique does is pick a useful direction and see how best it can move along it. Suppose the technique decides to move along the gradient of g_3 to develop it into an active constraint. There is obviously going to be a problem in pushing g_1 to become active. Pushing a constraint to be active is a basic way to reduce the objective function in well-formulated NLP problem. Therefore, most algorithms (unless they are trained to compromise in some significant way) will underachieve by only pushing one of the constraints to be active and ensuring the other is feasible.

ALM_7_4_1.m: This m-file uses the ALM method to solve the problem. In order to preserve the old ALM.m to reflect the discussion in Section 7.2.2 a copy of the file is made. **Ofun_741.m**, **Hfun_741.m**, **Gfun_741.m**, and **FALM_741.m** reflect the functions from Example 7.2. Also, **ALM_7_4_1.m** is adjusted to avoid handling equality constraints and provide some printing support. Otherwise it is essentially the same. Once again the output from the Command window is pasted below and annotated to draw attention to some issues. Only the initial prompts with the inputs, the initial iteration, and the consolidated information after the final iteration are copied below.

```
» ALM_7_4_1                (invoke program)
Input the starting design vector
This is mandatory as there is no default vector setup
```



```
The length of your vector indicates the number of
unknowns (n)
Please enter it now and hit return : for example [ 1
2 3 ...]
[0.75 0.55]                (initial design vector)

 The initial design vector:
 7.5000e-001 5.5000e-001

Input the minimum values for the design vector.
These are input as a vector. The default values are
3 * start value below the start value unless it is
zero.
In that case it is -5:
[0.02 0.02]   ( input lower bound : not currently used)

The minimum values for the design variables are :
2.0000e-002 2.0000e-002

Input the maximum values for the design vector.
These are input as a vector. The default values are
3 * start value above the start value.
If start value is 0 then it is +5:
[1 1]      (input upper bound -currently not used)

The maximum values for the design variables are : 1 1

Number of equality constraints [0] :
          (default value of 0 is ok)
Number of inequality constraints [0] : 4
Initial values for beta : [1 1 1 1]
(input number of inequality constraints and initial
beta)

ALM iteration number: 0
Design Vector (X) : 7.5000e-001 5.5000e-001
Objective function : 2.6000e-001
(In the following violated constraints are shown in
bold)
Inequality constraints : -5.9180e+000 -3.5997e+001
5.2383e+000 -1.9900e-001
Square Error in constraints(h, g) : 0 0
Lagrange Multipliers (lambda beta): 0 1 1 1 1
Penalty Multipliers (rh rg): 1 1
```

```
ALM iteration number: 4
Design Vector (X) : 6.4726e-001 6.1013e-001
Objective function: 4.6695e-002
Error in constraints (h, g) : 0 2.1366e-009
Lagrange Multipliers (lambda beta): 0 0 4.8227e-002
0
Penalty Multipliers (rh rg): 1000 1000


 The values for x and f and g are :
 7.5000e-001 5.5000e-001 2.6000e-001 0
 6.5647e-001 6.0173e-001 6.8871e-002 0
 6.4726e-001 6.1017e-001 4.6638e-002 1.5525e-006
 6.4726e-001 6.1013e-001 4.6695e-002 2.1366e-009
 6.4726e-001 6.1013e-001 4.6695e-002 2.1366e-009
(Note design and function are not changing hence
stopped)
 The values for beta are :
 1.0000e+000 0 0 0
 1.0000e+000 0 0 0
 1.0000e+000 1.4062e-002 3.8983e-002 4.8227e-002
 1.0000e+000 8.9252e-001 1.7073e-001 0
(Each column represents values of beta for an
iteration - beta for constraints 1,2 and 4 are zero and
it can be seen below they are feasible but inactive.
Only the third constraint is active)
 Inequality constraints : -6.7858e-002 -4.9734e+000
 4.6223e-005 -3.6136e-002
 (Note third constraint is active. The first one is
almost active)
```

The analytical solution for this problem is

$$x_1^* = 0.6774; \quad x_2^* = 0.6445; \quad f^* = 4.3577\text{E}-02;$$
$$g_1^* - 0.0; \quad g_2^* = -4.9837; \quad g_3^* = 0.0; \quad g_4^* = -0.3196\text{E}-02$$

In comparison, the ALM has done a decent job of getting close to the solution before stalling. Some of this can be traced to the discussion prior to running the program.

7.4.2 Example 7.3—Beam Design

This section requires the use of the Optimization Toolbox from MATLAB. More specifically, it uses the QP function to solve the QP subproblem in the SQP method. The problem is solved using the direct procedure illustrated through the example in

Section 7.3.2. This section combines the *symbolic* capability of MATLAB with the resources available in the Optimization Toolbox to implement an original interpretation of the SQP. The example is the design of an I-beam for use in a particular structural problem. This is a traditional mechanical/civil engineering structural optimization problem. It was introduced in Chapter 1 for illustration. It also appears in Chapters 8 and 10 for illustration of additional issues. In the example here, the I-beam will carry a weight at its center. The centric loading and the design variables for the problem are defined in Figure 8.7 (Chapter 8). The figure is not reproduced here but x_1 is the depth of the beam, x_2 is the flange width, x_3 is the web thickness, and x_4 is the flange thickness. Other parameters (g, L, W, E, σ_y, τ_y) are not redefined here.

The mathematical model is

$$\text{Minimize} \quad f(\mathbf{X}) = \rho L A_c$$

$$\text{Subject to:} \quad g_1(\mathbf{X}): WL^3 - 12EI \le 0.25 \quad \text{(deflection constraint)}$$

$$g_2(\mathbf{X}): WLx_1 - 8I\sigma_y \le 0 \quad \text{(normal stress constraint)}$$

$$g_3(\mathbf{X}): WQ_c - 2Ix_3\tau_y \le 0 \quad \text{(shear stress constraint)}$$

Some geometric constraints to relate the various design variables are

$$g_4(\mathbf{X}): x_1 - 3x_2 \le 0$$
$$g_5(\mathbf{X}): 2x_2 - x_1 \le 0$$
$$g_6(\mathbf{X}): x_3 - 1.5x_4 \le 0$$
$$g_7(\mathbf{X}): 0.5x_4 - x_3 \le 0$$

The side constraints on the design variables are

$$3 \le x_1 \le 20, \quad 2 \le x_2 \le 15, \quad 0.125 \le x_3 \le 0.75, \quad 0.25 \le x_4 \le 1.25$$

The following relations define the cross-sectional area (A_c), moment of inertia (I), and first moment of area about the centroid (Q_c) in terms of the design variables:

$$A_c = 2x_2x_4 + x_1x_3 - 2x_3x_4$$

$$I = \frac{x_2x_1^3}{12} - \frac{(x_2 - x_4)(x_1 - 2x_3)^3}{12}$$

$$Q_c = 0.5x_2x_4(x_1 - x_4) + 0.5x_3(x_1 - x_4)^2$$

The design parameters for the problem are $W = 1.4 \times 2000$ (1.4 is the factor of safety), $L = 96$, $E = 29$ E+06, $\rho = 0.284$, $\sigma_y = 36$ E+03, $\tau_y = 21$ E+03.

Ex_743.m: This is the m-file that will implement the solution through the SQP. While this is a traditional design problem, it is also not trivial. It is rare to find a published detailed procedure for solution. The solution is usually obtained numerically. In any event, analytical application of the KT conditions is usually not contemplated because of 2^7 cases (for the 7 constraints) even if many of those cases are trivial. Inclusion of the side constraints will provide an additional 8 linear constraints (a grand total of 15 constraints) of the form (illustrated for x_1 only as)

$$g_8(\mathbf{X}):\ -x_1 + 3 \le 0$$

...

$$g_{12}(\mathbf{X}):\ x_1 - 20 \le 0$$

...

The reader is strongly recommended to review the code in the m-file for the following reasons:

- The power of symbolic manipulation is exploited further using vector definitions to make the implementation fairly easy. The symbolic calculations are used to generate the data for the QP subproblem.
- The symbolic calculations are effortlessly integrated with numerical computations (the solution to the QP is numerical).
- The implementation of the SQP is straightforward and direct (and simple).
- The solution is significantly better than the numerical solution by MATLAB's own SQP program (Chapter 10).

The program requires the starting guess for the design vector. The functions for the example are hard coded. The final solution is printed to the Command window:

Initial Guess:	10.000	5.000	0.500	1.000
Final Design:	5.657	2.000	0.125	0.250
Objective:	44.838			

All constraints are satisfied:

Active constraints: 1 (deflection), 7, 9 (x_2—lower bound), 10 (x_3—lower bound), 11 (x_4—lower bound)

The reader is encouraged to explore why the solution obtained here is significantly better than the one using the SQP from the toolbox, granted that the implementations are not the same. Some thought should be given to the performance of quasi-Newton (metric) methods and their adaptation to constrained techniques if the Hessian is not positive definite, which in this case it is.

7.4.3 Example 7.4—Optimal Control

This section introduces another class of problems that can be addressed by design optimization techniques. It is the basic trajectory determination problem in optimal control. Given two points in a plane, find a path that will be traversed between them using a steering vector of constant magnitude. The path has to be traversed in unit time. The problem can be described using Figure 7.11. If the problem can be described in the physical plane, the steering vector considered as velocity, then the kinematic equations for a particle moving on the path can be described by the following differential equations (design optimization prefers to deal with algebraic functions), where x, y locate the problem in physical space:

$$\dot{x} = V\cos\theta(t);\quad 0 \le t \le 1$$
$$\dot{y} = V\sin\theta(t);\quad 0 \le t \le 1$$

The initial and final conditions are (these were the values chosen for the example)

$$x(0) = 0;\quad y(0) = 0$$
$$x(1) = 3;\quad y(1) = 2$$

Here, $\theta(t)$ is regarded as a control variable. It is a continuous function of t between $0 \le t \le 1$ that will transfer the particle from the initial point to the final point. In contrast, design optimization (i.e., this book) determines scalar values of the design variables.

These types of problems belong to the area of optimal control. There are special numerical techniques that are available to solve these categories of problems. One approach to these problems is to use design optimization techniques. This is aided by using a parametric representation for the control function. Many different parametric representations for the control have been used, such as cubic splines and Hermite polynomials. In this study a Bezier curve is used to represent the control. The differential equations are then numerically integrated to establish the trajectory.

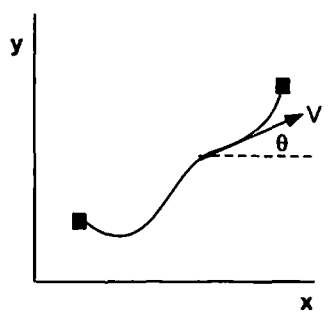

Figure 7.11 Description of trajectory control.

Solution Technique (adopted in this section)

- Use a Bezier curve to parametrize the control function.
- Integrate the differential equations forward using a Runge-Kutta method (ode45.m)
- Use an unconstrained technique (fminunc.m) minimizer to drive the square of the error in the final conditions to zero (least squared error).

This example needs the MATLAB toolbox to execute fminunc.m. Alternately, the unconstrained minimizer can be among those previously developed. This translates into an unconstrained minimum problem, which therefore should have been handled in Chapter 6. It is included here because the differential equations are actually considered *differential constraints*. The objective function therefore is

$$\text{Minimize} \quad f(\mathbf{X}) = [x_{\text{final}} - x(1)]^2 + [y_{\text{final}} - y(1)]^2$$

Ex744.m will execute the example and produce plots of the control and the trajectory. It uses two standard MATLAB functions mentioned before (ode45.m and fminunc.m). The Bezier curve generation requires **coeff.m, Combination.m, Factorial.m**. The objective function is constructed in **obj_optcont.m**. It requires the use of ode45.m,

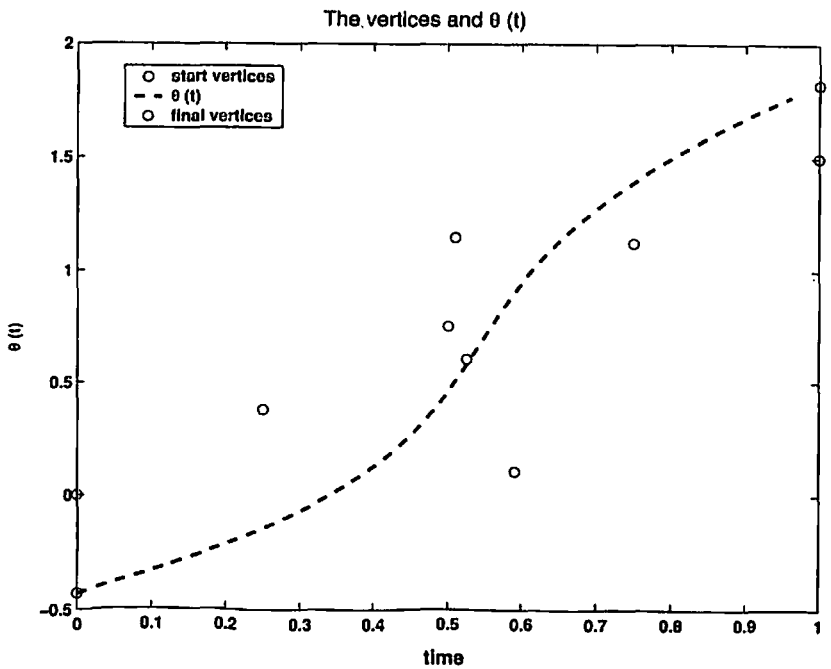

Figure 7.12 Control function $\theta(t)$.

which requires **brachi.m** to return the state space definition of the problem. **brachi.m** also includes the Bezier interpretation code.

Figure 7.12 describes the Bezier vertices of the control function at the start and the final iterations. The final continuous control is also plotted on the figure. The start curve is a straight line through the vertices and is not shown. Figure 7.13 shows the start trajectory and the final trajectory. It is clear from the figure that the initial and the final conditions are satisfied for the final trajectory. Since the equations were integrated, the differential constraints are therefore completely satisfied.

The final trajectory shown is only feasible—it is one of many—that can pass between the two points. This opens the possibility that a further optimization can be performed with respect to the trajectory. For example, find the shortest path in the time provided. Note it is not the straight path since for a given velocity magnitude the destination may be reached sooner than prescribed. The classical problem in optimal control is the *brachistochrone* problem. It is the minimum time for the particle to travel between two prescribed points dynamically (obeying principles of dynamics). The reader is encouraged to expand the technique and the code to explore these additional interesting problems.

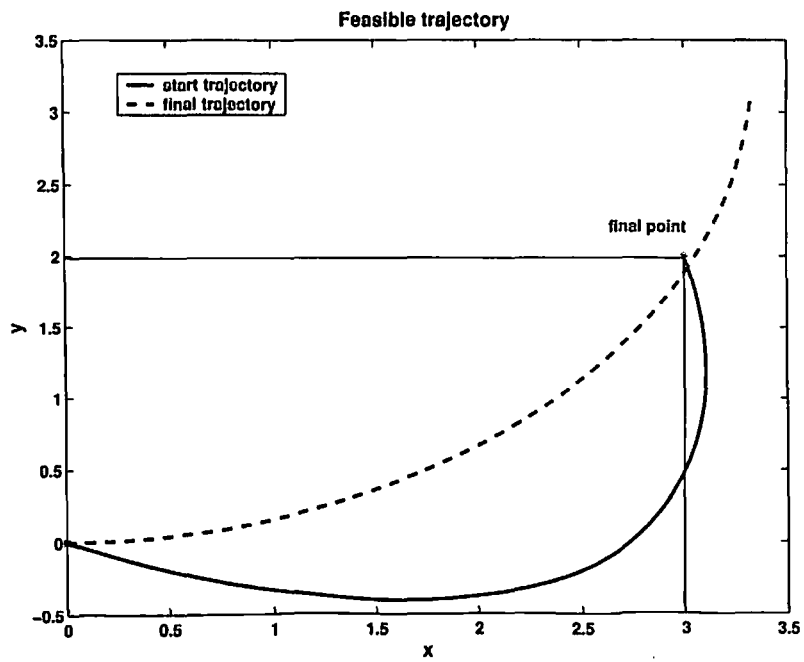

Figure 7.13 Trajectories—start and final.

REFERENCES

1. Fiacco, A. V., and McCormick, G. P., *Nonlinear Programming, Sequential Unconstrained Minimization Techniques*, Wiley, New York, 1968.
2. Vanderplaats, G. N., *Numerical Optimization Techniques for Engineering Design*, McGraw–Hill New York, 1984.
3. ANSYS Software, Ansys Inc., Pittsburgh, PA.
4. Bertekas, D. P., *Constrained Optimization and Lagrange Methods*, Academic Press, New York, 1982.
5. Kelley, J. E., The Cutting Plane Method, *Journal of SIAM*, Vol. 8, pp. 702–712, 1960.
6. Gill, P. E., Murray, W., and Wright, M. H., *Practical Optimization*, Academic Press, New York, 1981.
7. Boot, J. C. G., Quadratic Programming, *Studies in Mathematical and Managerial Economics*, H. Theil (ed.), Vol. 2, North–Holland, Amsterdam, 1964.
8. Biggs, M. C., Constrained Minimization Using Recursive Quadratic Programming: Some Alternate Subproblem Formulations, *Towards Global Optimization*, L. C. W. Dixon and G. P. Szego (eds.), pp. 341–349, North–Holland, Amsterdam, 1975.
9. Powell, M. J. D., A Fast Algorithm for Nonlinear Constrained Optimization Calculations, No. DAMPTP77/NA 2, University of Cambridge, England, 1977.
10. Han, S. P., A Globally Convergent Method for Nonlinear Programming, *Journal of Optimization Theory and Applications*, Vol. 22, p. 297, 1977.
11. Arora, J. S., *Introduction to Optimum Design*, McGraw–Hill, New York, 1989.
12. Branch, M. A., and Grace, A., *Optimization Toolbox*, User's Guide, MathWorks Inc., 1996.
13. Wolfe, P., Methods of Nonlinear Programming, *Recent Advances in Mathematical Programming*, R. L. Graves and P. Wolfe (eds.), McGraw–Hill, New York, 1963.
14. Gabriele, G. A., and Ragsdell, K. M., The Generalized Gradient Method: A Reliable Tool for Optimal Design, *ASME Journal of Engineering and Industry, Series B*, Vol. 99, May 1977.
15. Miele, A., Huang, H. Y., and Heideman, J. C., Sequential Gradient Restoration Algorithm for the Minimization of Constrained Functions—Ordinary and Conjugate Gradient Versions, *Journal of Optimization Theory and Applications*, Vol. 4, No. 4, 1969.
16. Levy, A. V., and Gomez S., Sequential Gradient-Restoration Algorithm for the Optimization of a Nonlinear Constrained Function, *Journal of the Astronautical Sciences*, Special Issue on Numerical Methods in Optimization, Dedicated to Angelo Miele, K. H. Well (sp. ed.), Vol. XXX, No. 2, 1982.
17. Rosen, J. B., The Gradient Projection Method for Nonlinear Programming, Part II: Nonlinear Constraints, *SIAM Journal of Applied Mathematics*, Vol. 9, No. 4, 1961.

PROBLEMS

7.1 Modify **Sec7_2_1_plot.m** to review contour values and plot automatically selected contours.

7.2 Modify **Sec7_2_1_calc.m** to automatically calculate the initial multipliers, loop over penalty multipliers, and to automatically pick the design vector that satisfies the side constraints.

7.3 (uses Optimization Toolbox) Develop the SLP by using the LP solver from the Optimization Toolbox and apply to Example 7.1.

7.4 Solve Example 7.1 by solving the QP problem graphically.

7.5 Develop the code to find a feasible solution from a given starting vector for constrained problems.

7.6 Develop a program to calculate the multipliers for the NLP problem at a given design.

7.7 Develop a program for a constrained stepsize calculation procedure.

7.8 (uses Optimization Toolbox) Use the QP program from the toolbox and develop your own SQP implementation.

7.9 (uses Optimization Toolbox) Use the SQP program from MATLAB to solve Example 7.1.

7.10 Develop **Sec7_3_2.m** to automatically execute several iterations to convergence.

7.11 Implement a new version of SQP where the Hessian matrix is maintained as the identity matrix for all iterations.

7.12 Finish solving Example 7.1 using the GRG method.

7.13 Solve Example 7.1 with $\mathbf{Z} = [x_2]$ and $\mathbf{Y} = [x_1 \quad x_3]^T$.

7.14 Modify the GRG code to include consideration of active constraints and compare the performance.

7.15 Build in KT condition calculation into the SGRA.

7.16 Solve the Brachistochrone optimal control problem.

8

DISCRETE OPTIMIZATION

This chapter introduces some concepts, methods, and algorithms associated with discrete optimization. It is not possible to compress the subject of Discrete Optimization to a single chapter as the topic itself is capable of spawning several courses. Discrete optimization is very different, difficult, diverse, and continues to develop even today. These problems are largely combinatorial and are computationally more time intensive than the corresponding continuous problems. The problems addressed by the discrete optimization research community are mainly in the area of operations research characterized usually by linear models. Engineering optimization mostly incorporates nonlinear relations. From a real perspective, discrete design variables are fundamental in engineering optimization. In the beam design problem in Chapter 1, a practical solution should involve identifying an "off-the-shelf" beam as the rolling mill will probably make it prohibitively expensive for a limited production of the unique beam that was identified as the solution to the continuous optimization problem. In the problem regarding the number of different placement machines the solution was expected to be integers (*discrete value*). Similarly, choice of diameters, lengths, washers, valves, men, components, stock sizes, and so on, are usually guided by avoiding high procurement costs associated with nonstandard components unless cost recovery is possible because of large volumes. Continuous optimization can be justified in *one-of-a-kind* design or if the item is to be completely manufactured in-house (no off-the-shelf component is necessary).

Practical engineering design requires that some design variables belong to an ordered set of values—*discrete variables*. This makes it a discrete optimization

problem. However, there are very few engineering problems that are currently solved as a discrete optimization problem. First, discrete optimization algorithms are difficult to apply. Second, they are time consuming. Third, most discrete algorithms and code address linear mathematical models. Discrete optimization in engineering will necessarily involve adaptations from the currently available techniques used by the decision-making community. Such adaptations are uncommon. It is also rare to find any book on design optimization that addresses or includes the subject of nonlinear discrete optimization, notwithstanding the fact that it is enormous in extent. A modest effort is being made in this book to acquaint the reader with the subject area.

The typical approach for incorporating discrete design variables in engineering is to solve the corresponding continuous optimization problem and adjust the optimal design to the nearest discrete values (this is similar to the rounding process to arrive at an *integer* number). According to Fletcher [1], there is no guarantee that this process is correct or that a good solution can be obtained this way. Very often this *rounding* may result in infeasible designs. Given that the alternative is to solve the discrete optimization problem itself, a systematic approach to this rounding process has become acceptable. A rounding process that is based on maintaining a feasible design is available in Reference 2.

This chapter and the next will differ significantly from the previous chapters in both their content and organization. Their primary focus is presenting new ideas and content rather than developing a technique or assisting in code development. Discrete problems require significantly different concepts than the traditional continuous mathematical models of the earlier chapters. Simple examples are used to bring out the difference. For example, derivatives, gradients, and Hessian do not apply to discrete problems. By extension, search directions and one-dimensional stepsize computation lose their relevance. While there are several excellent references on discrete optimization, almost all of them deal with linear mathematical models. There are few, if any, about nonlinear discrete optimization applied to engineering problems.

Discrete optimization problems (DP) are implied when the design variables are not continuous. Extending the concept of the side constraints, these design variables must be chosen from an ordered set of values, each design variable with its own distinct set, with no constraints on the spacing of values within the set. If the set is made up only of integers, then the problem is characterized as an *integer optimization* problem. If the mathematical model is linear, such a problem is identified as an Integer Programming (IP) problem as compared to the Linear Programming (LP) problem in Chapter 3. A large class of IP problems require the design variables (decision variables) to have values of either "0" or "1". These are further classified as Zero–One IP problems (ZIP—this book only). All IP problems can be reduced to Zero–One (0–1) IP problems by representing each integer by its binary equivalent, and associating a 0–1 design variable to each binary value location. For example:

integer value of 8 => binary value of 1000

Therefore, if the design variable x is restricted to integers between $0 \le x \le 16$, then x can be replaced by five 0–1 design variables $[y_1, y_2, y_3, y_4, y_5]$ from which x can be assembled as

$$x = y_1 (2^4) + y_2 (2^3) + y_3 (2^2) + y_4 (2^1) + y_5 (2^0)$$

Such a transformation is not recommended if there is a large number of integer variables. Engineering design problems can be expected to contain both continuous and discrete variables. These are termed Mixed Programming (MP) problems if the mathematical model is linear. In this book the classification is extended to nonlinear problems too.

In the interest of manageability, only three methods are presented in this chapter. These methods are only representative and in no way address the topic of discrete optimization sufficiently, let alone completely. The third one has seen limited use, but with computational resources not being a hurdle today, it is definitely attractive. The selection does not represent any priority, though the methods are among those that have been applied often in engineering. The methods of the next chapter have evolved through their initial application to discrete optimization. They could also belong to this chapter but have been kept separate because they are the driving force in the search for global optimum today. The methods of this chapter are (1) Exhaustive Enumeration, (2) Branch and Bound (partial or selective enumeration), and (3) Dynamic Programming.

8.1 CONCEPTS IN DISCRETE PROGRAMMING

In the opening discussion some definitions of discrete programming problems, regarding the nature of discrete optimization were introduced. In this section the concepts are detailed using a simple unconstrained minimization example. As a result, the optimality issues do not intrude on presenting the important ideas in discrete optimization including the treatment of finding a continuous solution followed by adjusting the discrete design variables to neighboring discrete values.

Example 8.1 Minimize the objective function given below, where x_1 is a continuous variable and x_2, x_3 are discrete variables. x_2 must have a value from the set [0.5 1.5 2.5 3.5] and x_3 must have a value from the set [0.22 0.75 1.73 2.24 2.78].

$$\text{Minimize } f(x_1, x_2, x_3) = (x_1 - 2)^2 + (x_1 - x_2)^2 + (x_1 - x_3)^2 + (x_2 - x_3)^2 \quad (8.1)$$

The side constraints on the design variables can be set up as

$$x_1 \in R \quad (8.2a)$$

$$x_2 \in [0.5 \quad 1.5 \quad 2.5 \quad 3.5] \in \mathbf{X}_{2d} \quad (8.2b)$$

$$x_3 \in [0.22 \quad 0.75 \quad 1.73 \quad 2.24 \quad 2.78] \in \mathbf{X}_{3d} \quad (8.2c)$$

The symbol $\in$ identifies that the variable on the left can have one of the values on the right side. In Equation (8.2a), R stands for a real value. This represents a standard use of the symbol $\in$ and is an effective way of expressing the idea that the discrete values can only be selected from a group of values. This is a mixed programming (MP) problem.

8.1.1 Problem Relaxation

If this were a unconstrained problem in continuous variables, the solution could be obtained by setting $\nabla f = 0$ and solving the three equations for the values of x_1, x_2, and x_3. Equations 8.2b and 8.2c has to be overlooked. Alternately, for a continuous problem, inspection of the objective function yields the following solution:

$$x_1^* = 2; \quad x_2^* = 2; \quad x_3^* = 2; \quad f^* = 0 \quad (8.3)$$

For the original problem $\partial f/\partial x_1$ is defined, but not $\partial f/\partial x_2$ or $\partial f/\partial x_3$, since x_2 and x_3 are discrete values. Derivatives are defined by taking the limit of the ratio of change of objective function f to the change in the value of the design variable, as the change in the variable approaches zero. The value of the objective function f is only defined at selected combination of x_2, x_3 in Equation (8.2) and is not defined elsewhere. Small/infinitesimal changes in the discrete design variables and therefore in the objective function are not defined in Example 8.1. Derivatives with respect to the discrete variables do not exist. This conclusion is of major significance as it makes this book's previous body of work of limited use in the pursuit of the solution to the DP. To recollect, the necessary and sufficient conditions, which drove the algorithms, were based on the gradients and their derivatives. Two areas in the previous chapters escape with a limited impact of this statement. They are LP and zero-order methods for numerical solution to unconstrained nonlinear optimization. Both of them play a significant role in discrete optimization problems. It is therefore possible to conclude that solution to Example 8.1 as established in Equation (8.3) is incorrect (the side constraints are in violation).

In DP the solution in Equation (8.3) represents the solution to a *relaxation* problem. Problem relaxation can take several forms. Mostly it is applied to the relaxation or the *weakening* of the constraints or the objective functions. There are no explicit constraints in Example 8.1. In this instance, the relaxation refers to the removal of the restriction of *discreteness* of the variables. This is identified as *continuous* relaxation. The relaxed problem or relaxation has several advantages [3].

- If a constraint relaxation is infeasible, so is the problem/model it relaxes.
- Constraint relaxation expands the set of feasible solutions. The relaxed optimal value must improve or equal the best feasible solution to the original problem/model.

- The optimal value of any relaxed model provides a lower bound on the optimal solution if it is a minimization problem. Similarly it establishes an upper bound for maximization problems.
- Many relaxation models produce optimal solutions that are easily rounded to good feasible solutions of the original problem/model. This appears to drive discrete optimization for engineering design problems.

The values established in Equation (8.3) are a solution to the *continuous* relaxation of Example 8.1. It was easy to obtain this solution while a technique for the solution to the discrete unconstrained optimization is still unknown—the Kuhn-Tucker conditions are no longer useful. The solution in Equation (8.3) is not acceptable—x_2^* and x_3^* are not elements of the permissible set (8.2). The value of $f^* = 0$ will be better than the best discrete solution—*lower bound on the solution* to the original problem.

8.1.2 Discrete Optimal Solution

A standard approach to solving discrete optimization problem, particularly in engineering, is to use the continuous relaxation of the mathematical model and generate a continuous solution. For Example 8.1 this is given in Equation (8.3). More than likely the discrete variables will not belong to the predefined discrete sets. These variables are then changed (or rounded) to the nearest discrete values. For problems with constraints, feasibility is checked. The best feasible solution is then chosen. Unlike continuous problems, there are no necessary and sufficient conditions to satisfy.

In the case of Example 8.1, Equations (8.2) and (8.3) provide the relevant information to establish the solution. For convenience, they are reproduced here:

$$x_1 \in R \tag{8.2a}$$

$$x_2 \in [0.5 \quad 1.5 \quad 2.5 \quad 3.5] \in \mathbf{X}_{2d} \tag{8.2b}$$

$$x_3 \in [0.22 \quad 0.75 \quad 1.73 \quad 2.24 \quad 2.78] \in \mathbf{X}_{3d} \tag{8.2c}$$

$$x_1^* = 2; \quad x_2^* = 2; \quad x_3^* = 2; \quad f^* = 0 \tag{8.3}$$

The continuous variable x_1 is chosen to have a value of 2.0. Figure 8.1 indicates four sets of discrete values for x_2 and x_3 around the continuous solution. The least value of the objective function at these points will be considered the solution. Evaluating the values of the objective function at those points:

$$x_1^* = 2; \quad x_2^* = 1.5; \quad x_3^* = 1.73; \quad f^* = 0.3758 \tag{8.4a}$$

$$x_1^* = 2; \quad x_2^* = 1.5; \quad x_3^* = 2.24; \quad f^* = 0.8552 \tag{8.4b}$$

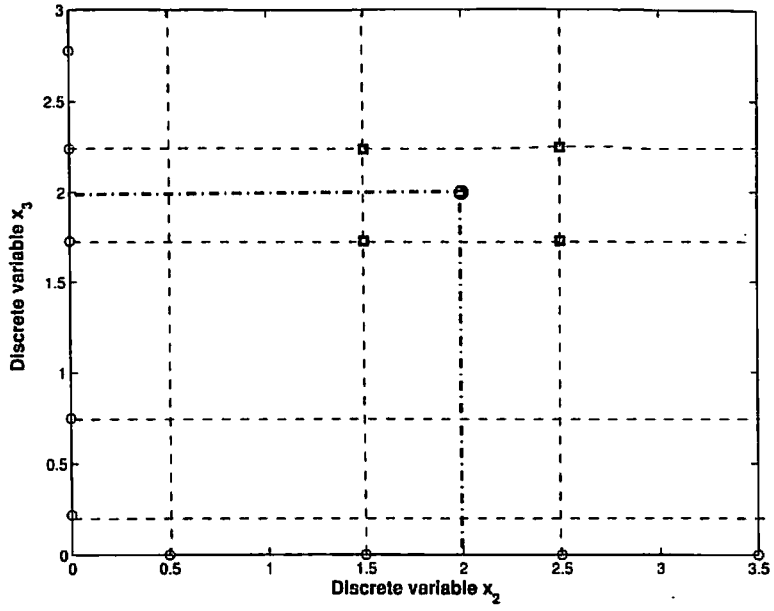

Figure 8.1 Discrete neighborhood of continuous solution.

$$x_1^* = 2; \quad x_2^* = 2.5; \quad x_3^* = 1.73; \quad f^* = 0.9158 \tag{8.4c}$$

$$x_1^* = 2; \quad x_2^* = 2.5; \quad x_3^* = 2.24; \quad f^* = 0.3752 \tag{8.4d}$$

From this exercise, the least value of the objective function is available in Equation (8.4d) and would be regarded as the adjusted optimum solution to the original MP problem. This is also the common practice in engineering design optimization.

The *best solution*, however, is obtained at

$$x_1^* = 1.7433; \quad x_2^* = 1.5; \quad x_3^* = 1.73; \quad f^* = 0.1782 \tag{8.5}$$

The difference between Equations (8.5) and (8.4a), only in the variable x_1, suggests that additional continuous optimization needs to be performed for each set of the discrete values selected, confirming the earlier observations by Fletcher. This optimization should be easier as the order of the mathematical model will be reduced by the number of discrete variables since they have been assigned numerical values. This simple example recommends at the very least a three-step procedure for problems that include discrete variables:

Step 1: A continuous relaxation that identifies several sets of discrete variables for further exploration.

Step 2: For each such set of discrete variable combination, a continuous optimization is performed to establish a new optimum value of the continuous variables and the corresponding objective function. If all of the variables are discrete, then only the function and constraints need to be evaluated at each of the set of variables.

Step 3: A simple comparison of the above solutions/values in Step 2 to identify the optimum solution to the discrete problem.

This unconstrained optimization example in three variables has demonstrated that discrete optimization requires a lot of work compared to continuous optimization. This work expands significantly if the number of variables increases or if the mathematical model is enhanced through the inclusion of constraints. Another essential feature in the above exploration is that no new mathematical conditions were necessary for establishment of the discrete solution beyond a simple comparison of the objective function. The nature of discrete variables and discrete functions precludes any sophisticated mathematical conditions established by derivatives of the functions involved in the model. Trapping and branching based on comparison of values are the mainstay of discrete algorithms. There techniques are classified as *heuristic methods*. This encourages unique and personal implementations of the search for discrete optimization that can be tailored for a class of problems.

8.2 DISCRETE OPTIMIZATION TECHNIQUES

There are three discrete optimization techniques in this section. The first one is Exhaustive Enumeration. This involves identifying the solution to the mathematical model for all possible combinations of the discrete variables. This is suggestive of the zero-order numerical techniques of Chapter 6. Those methods lacked sophistication as they involved only evaluation of the functions at a phenomenal number of points. They were able to take advantage of the plentiful computer resources available today. Imagine operating in the peer-to-peer computing environment afforded by "MP3" or "Gnutella" and solving the problem using all of the PCs in the world. The second method is the Branch and Bound method. This is based on partial enumeration where only part of the combinations are explored. The remaining are pruned from consideration because they will not determine the solution. This is currently the most popular method for discrete optimization for engineering design. The last method is Dynamic Programming, an elegant approach to optimization problems, but which did not gain favor because it involved significantly larger amounts of computation than competitive methods even for problems of reasonable size. It is restricted to problems that require a sequential selection of the design variables. Today, such resource limitations are disappearing in the world of powerful PCs and hence the method deserves to be revisited.

Standard Format—Discrete Optimization: The author is not aware of a standard format for the discrete optimization problem. It subsumes the format of the corresponding continuous relaxation problem. In this book the following format for the mixed optimization problem is used.

$$\text{Minimize} \quad f(\mathbf{X}, \mathbf{Y}), \quad [\mathbf{X}]_{n_c}; [\mathbf{Y}]_{n_d} \tag{8.6}$$

$$\text{Subject to:} \quad \mathbf{h}(\mathbf{X}, \mathbf{Y}) = [0]; \quad [h]_l \tag{8.7}$$

$$\mathbf{g}(\mathbf{X}, \mathbf{Y}) \le [0]; \quad [g]_m \tag{8.8}$$

$$x_i^l \le x_i \le x_i^u; \quad i = 1, 2, \ldots, n_c \tag{8.9}$$

$$y_i \in \mathbf{Y}_{d_i}; \quad [\mathbf{Y}_{d_i}]_{p_i}; \quad i = 1, 2, \ldots, n_d \tag{8.10}$$

$\mathbf{X}$ represents the set of n_c continuous variables. $\mathbf{Y}$ represents the set of n_d discrete variables. f is the objective function. $\mathbf{h}$ is the set of l equality constraints. $\mathbf{g}$ is the set of m inequality constraints. Expression (8.9) represents the side constraints on each continuous variable. Expression (8.10) expresses the side constraints on the discrete variables. Each discrete variable y_i must belong to a preestablished set of p_i discrete values $\mathbf{Y}_{d_i}$. If $n_d = 0$, then it is a continuous optimization problem. If $n_c = 0$, then it is a discrete optimization problem. If both are present, then it is a Mixed Problem.

Continuous Relaxation: The continuous relaxation of the mixed optimization problem (8.6)–(8.10) is identical to problem (8.6)–(8.10) with $\mathbf{Y}$ as a continuous variable set. The discrete constraint (8.10) is replaced by a continuous side constraint for the discrete variables of the form of (8.9). This is not explicitly developed since (8.6)–(8.9) and a modified (8.10) is fairly representative of the continuous relaxed mathematical model.

Reduced Model: The reduced model is important in subsequent discussions. It is the model solved after a part of the set of discrete variables are set at some allowable discrete values. This removes those variables from the problem as their values have been set. The mathematical model is then defined by the remaining discrete variables as well as the original continuous variables. If $\mathbf{Z}(n_z)$ are the remaining discrete variables that need to be solved, then

$$\text{Minimize} \quad \tilde{f}(\mathbf{X}, \mathbf{Z}), \quad [\mathbf{X}]_{n_c}; [\mathbf{Z}]_{n_z} \tag{8.11}$$

$$\text{Subject to:} \quad \tilde{h}(\mathbf{X}, \mathbf{Z}) = [0]; \quad [\tilde{\mathbf{h}}]_l \tag{8.12}$$

$$\tilde{g}(\mathbf{X}, \mathbf{Z}) \le [0]; \quad [\tilde{\mathbf{g}}]_m \tag{8.13}$$

$$x_i^l \le x_i \le x_i^u; \quad i = 1, 2, \ldots, n_c \tag{8.14}$$

$$z_i \in \mathbf{Z}_{d_i}; \quad [\mathbf{Z}_{d_i}]_{z_i} \tag{8.15}$$

Table 8.1a Optimal Value of Continuous Variable x_1 for Discrete Combination of x_2 and x_3

x_2/x_3	0.2200	0.7500	1.7300	2.2400	2.7800
0.5	0.9067	1.0833	1.4100	1.5800	1.7600
1.5	1.2400	1.4167	1.7433	1.9133	2.0933
2.5	1.5733	1.7500	2.0767	2.2467	2.4267
3.5	1.9067	2.0833	2.4100	2.5800	2.7600

Note that in the above, $\mathbf{Z}_{d_i}$ is not a new set. It corresponds to a reduced $\mathbf{Y}_{d_i}$. In Exhaustive Enumeration, generally it is an empty set. All of the discrete variables are set to some allowable value.

8.2.1 Exhaustive Enumeration

This is the simplest of the discrete optimization techniques. It evaluates an optimum solution for all combinations of the discrete variables. The best solution is obtained by scanning the list of feasible solutions in the above investigation for the minimum value. The total number of evaluations is

$$n_e = \prod_{i=1}^{n_d} p_i \tag{8.16}$$

If either n_d or p_i (or both) is large, then this represents a lot of work. It also represents an exponential growth in the calculations with the number of discrete variables. In a mixed optimization problem, this would involve n_e continuous optimum solution of the reduced model. If the mathematical model and its computer calculations are easy to implement, this is not a serious burden. If the mathematical model requires extensive calculations—for example, finite element or finite difference calculations—then some concern may arise. In such cases, limiting the elements of the discrete variables to smaller clusters may represent a useful approach.

Example 8.1: The exhaustive enumeration of Example 8.1 is available in **Ex8_1.m.**[1] Table 8.1a shows the optimum value of the continuous variable x_1 for every allowable combination of the discrete variables x_2 and x_3. Table 8.1b shows the corresponding value of the objective function for each of the cases. Since this is an unconstrained problem, no discussion of feasibility is involved. There are 20 values and the best solution for Example 8.1 is

$$x_1^* = 1.7433; \quad x_2^* = 1.5; \quad x_3^* = 1.73; \quad f^* = 0.1782 \tag{8.5}$$

There is some good news though. First, each of the 20 cases is essentially a *single-variable* optimization problem. The number of design variables in the original

[1]Files to be downloaded from the web site are indicated by boldface sans serif type.

Table 8.1b Optimal Value of Objective Function for Discrete Combination of x_2 and x_3

x_2/x_3	0.2200	0.7500	1.7300	2.2400	2.7800
0.5	1.9107	1.3542	2.7915	4.8060	7.8840
1.5	3.3240	1.3542	0.1782	0.8327	2.4707
2.5	8.0707	4.6875	0.8982	0.1927	0.3907
3.5	16.1507	11.3542	4.9515	2.8860	1.6440

model (3) is reduced by the number of discrete variables (2). Model reduction is involved in enumeration techniques.

Algorithm: Exhaustive Enumeration (A8.1)

Step 1. $f^* = \inf$, $\mathbf{X} = [0, 0, \ldots, 0]$

For every allowable combination of $(y_1, y_2, \ldots, y_{nd}) \Rightarrow (\mathbf{Y}_b)$

Solve Optimization Problem (8.11)–(8.15) (Solution $\mathbf{X}^*$)

If $\mathbf{h}(\mathbf{X}^*, \mathbf{Y}_b) = [0]$ and

If $\mathbf{g}(\mathbf{X}^*, \mathbf{Y}_b) \le [0]$ and

If $f(\mathbf{X}^*, \mathbf{Y}_b) < f^*$

Then $f^* \leftarrow f(\mathbf{X}^*, \mathbf{Y}_b)$

$\mathbf{X} \leftarrow \mathbf{X}^*$

$\mathbf{Y} \leftarrow \mathbf{Y}_b$

End If

End If

End If

End For

Example 8.2 illustrates the use of algorithm A8.1 with feasibility requirements.

Example 8.2 The Aerodesign Club has initiated fund-raising activities by selling pizza during lunch in the student activity center. Over the years they have collected data to identify a mathematical model that will reduce their cost which directly translates to higher profits. Only two types of pizza will sell—pepperoni (x_1) and cheese (x_2). The local pizza chain will deliver a maximum of 26 pizzas. Factoring into account wastage and the time available to make the sale they arrive at the following mathematical model:

$$\text{Minimize} \quad f(x_1, x_2) = (x_1 - 15)^2 + (x_2 - 15)^2 \tag{8.17}$$

$$\text{Subject to:} \quad g_1(x_1, x_2): \quad x_1 + x_2 \le 26 \tag{8.18a}$$

$$g_2(x_1, x_2): \quad x_1 + 2.5x_2 \le 37 \tag{8.18b}$$

$$x_1 \geq 0; \quad x_2 \geq 0, \text{ and integer} \tag{8.18c}$$

The graphical solution for continuous relaxation is available in Figure 8.2 and the analytical solution (not included) identifies that 12.86 pepperoni pizzas and 9.65 cheese pizzas will provide for the minimum cost of operation.

Ex8_2.m is the m-file that will perform the exhaustive enumeration of the problem. The discrete solution is

$$x_1^* = 12; \quad x_2^* = 10; \quad f^* = 34; \quad g_2 \text{ is active} \tag{8.19}$$

This confirms that neighboring discrete values around a discrete solution works well. Programming exhaustive enumeration is straightforward and particularly easy with MATLAB. Given the processing speed, large memory availability on the desktop, easy programming through software like MATLAB, exhaustive enumeration is probably a very good idea today. What recommends it even further is that the solution is a *global optimum*. Furthermore, unlike the earlier chapters where the concepts of derivatives, matrices, linear independence, and so on were essential to understand and implement the techniques of optimization, lack of such knowledge is no disadvantage to solving a discrete optimization problem using exhaustive enumeration. There is only one essential skill—translating the mathematical model into program code. With the experience provided in this book, this is definitely not an issue.

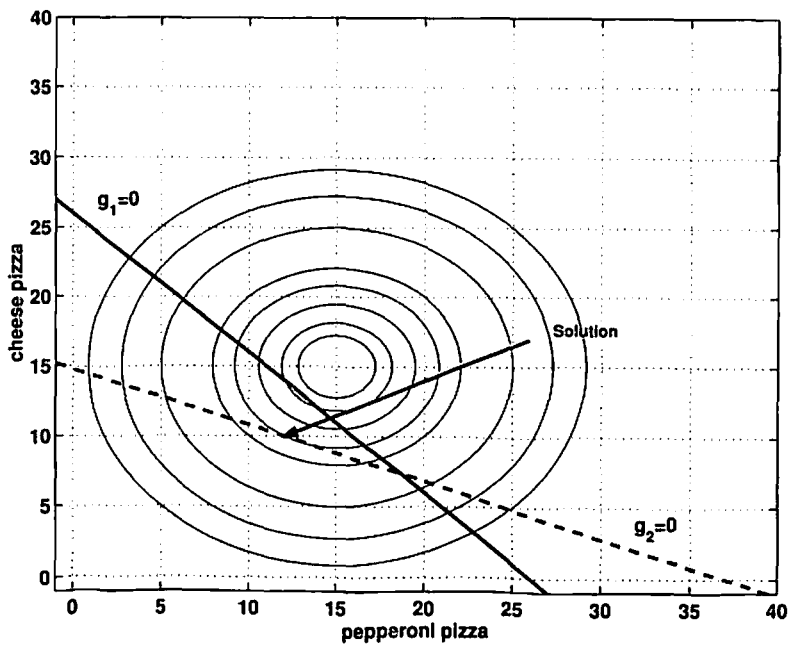

Figure 8.2 Graphical solution Example 8.2.

The solution in Equation (8.19) can be obtained with continuous relaxation of the problem. In this case the discrete solution has to be recovered by adjusting the continuous solution to neighboring discrete values.

Exhaustive Enumeration for Continuous Optimization Problems: This title must be qualified further—in lieu of gradient-based numerical techniques. The availability of serious desktop computing power, the promise of global optimization, and the ease of implementation suggest a *multistage* exhaustive enumeration strategy for continuous optimization problems. Exhaustive enumeration after all is a *search strategy*, similar to the ones used before. The search for solution is akin to the zero-order methods employed for unconstrained optimization (Chapter 6).

The higher stages in multistage enumeration involve enumerating on decreased intervals centered around the solution from the previous stage. The values of the design variables used for enumeration will incorporate progressively improved tolerance. The strategy must take into account the possibility that the initial interval might overlook global optimal solutions that do not appear attractive in a particular stage. Hence, more than one minimum is tracked in each stage. Each of these minimum can be considered an *enumeration cluster*. Each such cluster is multistaged until a prescribed tolerance is achieved. In engineering, manufacturing tolerance of the order of 0.0001 is achievable today. Tolerances below this limit probably need to be justified unless the area is microelectronics. From a programming perspective it just adds an extra loop to the implementation. The final solution obtained represents a continuous solution within the final tolerance implemented by the technique.

Such an exercise is left to the student. As with all implementation the merit of this approach lies in the tractability of the mathematical model and the need for a global optimum solution. Since the method is largely heuristic, there is no guarantee that the solution is actually a global optimum (of course assuming the functions involved are not convex).

8.2.2 Branch and Bound

Exhaustive or complete enumeration is possible only for a limited set of variables because of the exponential growth of effort required to examine all possible solutions. Using *partial* (or *incomplete* or *selective*) enumeration to arrive at the solution to the discrete optimization problem would provide the necessary advantage in handling large problems. Partial enumeration works by making decisions about groups of solutions rather than every single one. The Branch and Bound (BB) algorithm is one of the successful algorithms to use selective/partial enumeration. It does so by using relaxation models instead of the original complete discrete mathematical model. The BB algorithm uses a *tree* structure to represent *completions* of *partial solutions*. The tree structure uses *nodes* and *edges/links* to represent the trail of partial solutions. This is called *fathoming* the partial solution. The terminology is borrowed from operation research literature [3] and is explained below with respect to Example 8.1 which is reproduced here for convenience:

$$\text{Minimize} \quad f(x_1, x_2, x_3) = (x_1 - 2)^2 + (x_1 - x_2)^2 + (x_1 - x_3)^2 + (x_2 - x_3)^2 \quad (8.1)$$

$$x_1 \in R \quad (8.2a)$$

$$x_2 \in [0.5 \quad 1.5 \quad 2.5 \quad 3.5] \in \mathbf{X}_{2d} \quad (8.2b)$$

$$x_3 \in [0.22 \quad 0.75 \quad 1.73 \quad 2.24 \quad 2.78] \in \mathbf{X}_{3d} \quad (8.2c)$$

Partial Solution: A solution in which some discrete design variables are fixed and others are *free* or undetermined. Variables that are not fixed are represented by the symbol "*f*." For example,

$$\mathbf{X} = [f, 0.5, f] \quad (8.20)$$

represents the optimization problem with $x_2 = 0.5$ and x_1 and x_3 to be determined. The partial solutions are also termed the *nodes* of the tree. The free variables are usually solved using a continuous relaxation of the model.

Completions (of Partial Solution): Each partial solution or node can give rise to additional partial solutions/nodes directly under it. This is called *branching* or *expansion* of the node. In this new set of nodes another one of the free discrete variables is fixed and the remainder still continue to be free. For example, under the node represented by the partial solution in Equation (8.20), five additional nodes are possible:

$$\mathbf{X} = [f, 0.5, 0.22]; \quad \mathbf{X} = [f, 0.5, 0.75]; \quad \mathbf{X} = [f, 0.5, 1.73];$$
$$\mathbf{X} = [f, 0.5, 2.24]; \quad \mathbf{X} = [f, 0.5, 2.78]$$

These completions branch out and hence a treelike structure. There is a hierarchical structuring of the nodes. Each branch represents leads to a tier in which some variable is held at its prescribed discrete value.

Fathoming the Partial Solution: Expanding the node all the way till the end is termed *fathoming* a node. During the fathoming operation it is possible to identify those partial solutions at a node that do not need to be investigated further. Only the best solution at the node is *branched* or *completed*. Those partial solutions/nodes that will not be investigated further are *pruned* or *terminated* in the tree.

Active Partial Solution: A node that has not been terminated or pruned and has not been completed.

Edges/Links: These are node-to-node connections in a tree. They identify the value of the new variable that is fixed.

Root: The BB algorithm searches along a tree. It starts at the *root* where all the design variables are free. BB search stops when all partial solutions in the tree have been branched or terminated. The search is depth first—completion at a node is preferred to jumping across nodes in the same hierarchy.

Incumbent Solution: The incumbent solution is the best feasible solution known so far with all discrete variables assuming discrete values. When the BB search finally stops, the final incumbent solution, if one exists, is a global optimum solution. In the following the incumbent solution is denoted by f^{**}. The incumbent design variables are $\mathbf{X}^{**}$.

Example 8.1—Using Branch and Bound: Here the BB search is applied to Example 8.1. **BBEx8_1.m** calculates all of the data needed for generating the tree. The BB tree itself is developed in Figures 8.3 and 8.4.

Start: Node 0. All of the variables are free. The solution to the continuous relaxation of the objective function (solved as a continuous problem) is identified at the node.

Nodes 1–4: This is the first *tier* of the BB tree. The discrete variable x_2 is assigned each of the permissible discrete values at the nodes in this tier. This is indicated

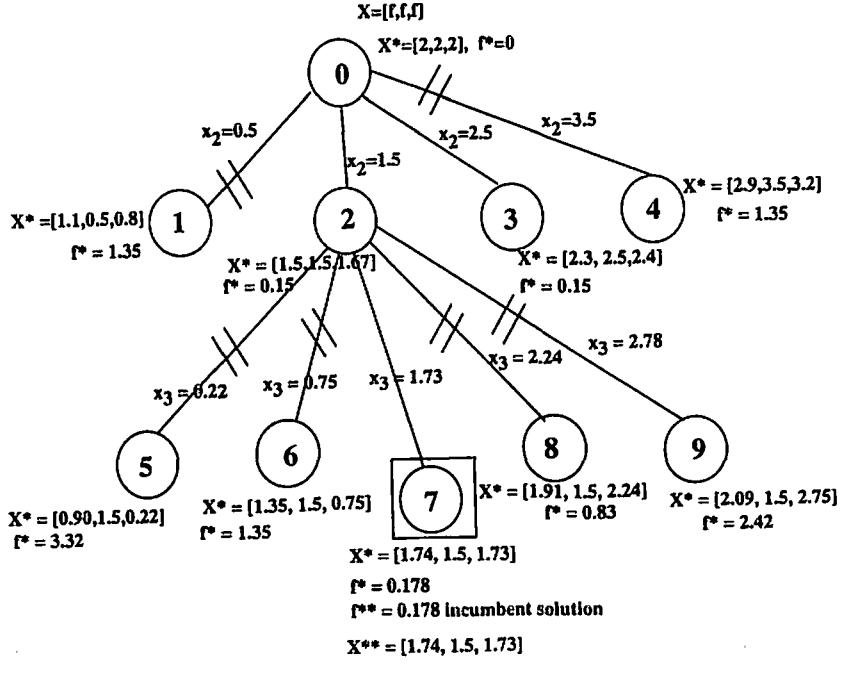

Figure 8.3 Branch and bound Example 8.1.

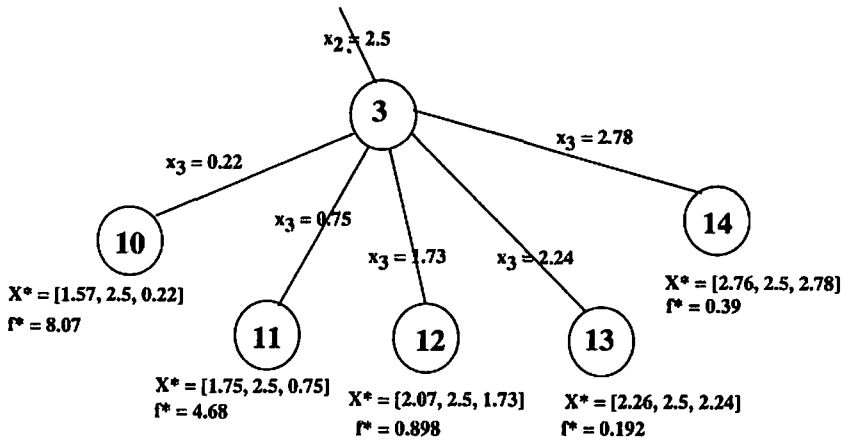

Figure 8.4 Branch and bound Example 8.1—fathoming Node 3.

on the edges between the nodes in Figure 8.3. The remaining variables are free/continuous. The optimum solution at the various nodes is displayed on the tree. The minimum solution is the one that will be fathomed at the next step. Here Nodes 2 and 3 are candidates for expanding the branches. Nodes 1 and 4 will not be fathomed (unless their optimum value is better than the incumbent solution)—they will be terminated or pruned as the objective function has a higher value. The continuous relaxation ensures a lower bound on the optimum value of the objective at the node. Node 2 is taken up for fathoming based on node ordering.

Completions of Node 2: This expands Node 2 by setting the discrete variable x_3 to its permissible set of discrete values. It creates branches Nodes 5–9. It is also the second tier of the tree. The solution is indicated at the various nodes. In all of them the two discrete variables are set to indicated values while x_1 is free, which is a continuous variable by definition. Since all of the discrete variables have discrete values, this node is also fathomed. Node 7 indicates the best solution. Since *all* of the discrete variables are assigned discrete values, the solution at this node is the *current incumbent solution*. The remaining nodes will be pruned. The current incumbent solution ($f^{**} = 0.1782$) is higher than the solution at node 3 ($f^* = 0.15$). This requires Node 3 to be fathomed.

Completions of Node 3: Figure 8.4 illustrates the branching at Node 3. The process is similar to completions of Node 2 above. The best solution is at Node 13 with a value of $f^* = 0.1927$. This is larger than the current incumbent solution. All of the nodes can be terminated.

No further nodes are available for expansion. The current incumbent solution is also the *global optimal solution*. This is the solution to the optimization problem. In this example, 15 partial solutions were necessary. Each partial solution can be considered an enumeration. In exhaustive enumeration, 20 enumerations were required for Example 8.1. There is a net gain of 5 enumerations. The BB search reduces the number of total enumerations needed to solve the problem.

Algorithm: Branch and Bound (A8.2): For even a small set of discrete variables actually drawing the BB tree structure is most likely a difficult enterprise. The tree structure merely identifies the search process. Example 8.1 did not have constraints so feasibility was not examined. With constraints present, choosing the partial solution to complete based only on minimum value of the objective function is not a justifiable strategy. In this case the best feasible solution is the candidate for completion. Also, an active feasible partial solution that has a better value than the current incumbent solution is also a candidate for completion. The following algorithm captures the essence of the Branch and Bound algorithm.

Step 0. Establish the root node with all discrete variables free.

Initialize solution index s to 0.

Set current incumbent solution to $f^{**} = \infty$, $\mathbf{X}^{**} = [\infty]$, (minimization problem).

Step 1. If any active feasible partial solutions remain (that is better than the current incumbent solution), select one as $\mathbf{X}^{(s)}$ and go to Step 2.

If none exists and there exists an incumbent solution it is optimal.

If none exists and there is no incumbent solution the model is infeasible.

Step 2. Obtain the completion of the partial solution $\mathbf{X}^{(s)}$ using a continuous relaxation of the original model.

Step 3. If there are no feasible completions of the partial solution $\mathbf{X}^{(s)}$, terminate $\mathbf{X}^{(s)}$, increment $s \leftarrow s + 1$, and return to Step 1.

Step 4. If the best feasible completion of partial solution $\mathbf{X}^{(s)}$ cannot improve the incumbent solution, then terminate $\mathbf{X}^{(s)}$, increment $s \leftarrow s + 1$, and return to Step 1.

Step 5. If the best feasible completion is better than the incumbent, update the incumbent solution

$$f^{**} \leftarrow f^{(s)}, \qquad \mathbf{X}^{**} \leftarrow \mathbf{X}^{(s)}$$

terminate $\mathbf{X}^{(s)}$, increment $s \leftarrow s + 1$, and return to Step 1.

Example 8.2 The Branch and Bound algorithm is applied to Example 8.2. The search is applied through the BB tree. The mathematical model for Example 8.2, where x_1, and x_2 are integers, is

$$\text{Minimize} \quad f(x_1, x_2) = (x_1 - 15)^2 + (x_2 - 15)^2 \tag{8.17}$$

$$\text{Subject to:} \quad g_1(x_1, x_2): \quad x_1 + x_2 \le 26 \tag{8.18a}$$

$$g_2(x_1, x_2): \quad x_1 + 2.5x_2 \le 37 \tag{8.18b}$$

$$x_1 \ge 0; \quad x_2 \ge 0 \tag{8.18c}$$

The solution is

$$x_1^* = 12; \quad x_2^* = 10; \quad f^* = 34; \quad g_2 \text{ is active} \tag{8.19}$$

To focus on the method and keep the figure compact, the discrete set for the design variables is reduced to four values around the solution in Equation (8.19). In the following development

$$x_1 \in [10 \quad 11 \quad 12 \quad 13]$$
$$x_2 \in [9 \quad 10 \quad 11 \quad 12]$$

Using Branch and Bound: The BB tree is shown in Figure 8.5. The data for the tree are calculated through **BBEx8_2.m** where the continuous relaxation is solved analytically using the KT conditions. The tree is branched as follows:

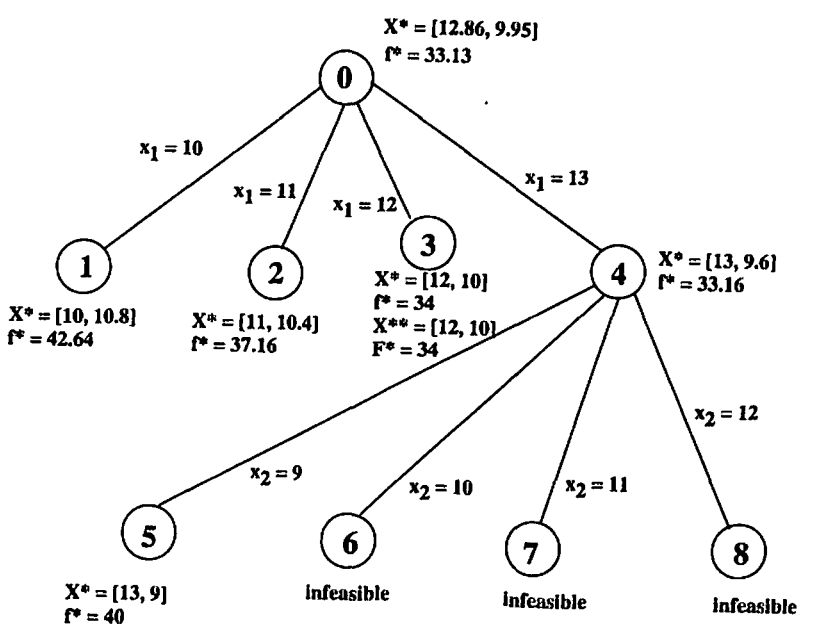

Figure 8.5 Branch and bound tree for Example 8.2.

Node 0: This is the root node. All the variables are free. The continuous relaxation of the problem is solved. The solution is indicated on the tree. Currently this is the only available active partial solution and hence needs to be completed.

Nodes 1–4: The partial solution at Node 0 is completed using variable x_1 from the reduced discrete set identified above. This leads to four branches and the optimal values are indicated on the tree. The best values are indicated at the nodes. For this problem they are also feasible. There are now four active partial solutions.

Node 3: The solution at Node 3 is also a feasible solution of the original discrete optimization problem since the design variables are discrete and belong to the permissible discrete set. Node 3 is *fathomed*. This provides the first incumbent solution.

There are three available active feasible partial solutions (Nodes 1, 2, and 4) for completion. Since the solutions at the nodes are all feasible, only the best active feasible partial solution is picked up for completion—Node 4. Nodes 1 and 2 can be terminated.

Completion Node 4: Node 4 has a feasible solution better than the current incumbent solution. It is completed by branching to Nodes 5, 6, 7, 8. Only Node 5 has a feasible solution. The best solution at Node 5 is not better than the current incumbent solution. Nodes 5, 6, 7, 8 can be terminated.

There are no more active partial solutions and the tree has been fathomed. The current incumbent solution is the optimal solution.

This completes the BB search applied to Example 8.2. The reduction in enumeration, compared to the exhaustive enumeration, is difficult to comment on for two reasons.

1. The deliberate choice to restrict the set of discrete values used to branch the tree seemed arbitrary. While this was done for convenience, a serious justification can be advanced based on the continuous solution at Node 0. The reduced discrete set is chosen to include a limited neighborhood/window of the continuous solution.
2. The exhaustive enumeration, based on discrete values, was easier to evaluate for a pure discrete problem. Applying KT conditions to obtain the possible partial solutions to the continuous relaxation, and choosing the acceptable solution is an involved exercise. Using numerical techniques like SQP to solve the problem, which will most likely be done in practice, has its own problems. Dependence on good starting values with no guarantee of convergence requires user intervention to identify the particular solution. It would require a lot of effort to automate the process so as to apply the Branch and Bound algorithm to general nonlinear mixed optimization problems. Linear mixed optimization problems do not possess this limitation, at least to the same extent.

In this example, the active partial solutions at the root and in the first tier were all feasible. What would be the strategy if the root node or other nodes have an infeasible active partial solution? In that case the BB search process must complete all active partial solutions and not only feasible active partial solutions. That is a modification that can be easily made to the Branch and Bound Algorithm above.

Other Search Techniques: The two search techniques in this section, namely, exhaustive enumeration, and branch and bound, provide an additional feature of discovering *global optimum* solutions. They also belong to the class of methods used to address global optimum. The easy availability of large and faster computing resources has brought renewed emphasis to globally optimal designs. *Simulated annealing* and *genetic algorithms* are among the leading candidates for global optimization. They are covered in the next chapter and can also be characterized as enumeration techniques, like the ones in this section. Both of them can and are used for discrete optimization. They are not discussed in this chapter. The application of most of these techniques is still heuristic and often depends on experience based on numerical experiments among classes of problems. Some applications are problem specific and require user intervention and learning. Standard implementations in these methods, especially for nonlinear problems, are not yet the norm.

8.2.3 Dynamic Programming

Discrete dynamic programming (DDP) is an exciting technique for handling a special class of problems. Examples 8.1 and 8.2 are not direct members of this class. Richard Bellman [4] was responsible for first introducing the concept and the algorithm. It is an optimizing procedure based on Bellman's principle of optimality. The principle is based on a sequence of decisions on partial solutions, so that when certain variables have been determined, the remaining variables establish an optimum completion of the problem [5]. Another expression of this principle is obtained from Reference 6. An optimal sequence of decisions has the property that at the current time, whatever the initial state and the previous decisions, the remaining decisions must be an optimal set with regard to the state resulting from the first decision. DDP is about a *sequence* of decisions (often in time) and is sometimes termed a *sequential decision problem*. Problems like Examples 8.1 and 8.2 do not directly fit this classification but they can be transformed to fit this requirement [5]. Very often, DDP problems can be described as an optimal path determination problem. Such path problems are solved using directed graphs (*digraphs*). These problems may actually have no connection to physical paths or distances. Like the BB tree, they enable better appreciation/application of the algorithmic procedure and are not strictly required. DDP requires definition of *states* and *stages*, and only the former is discussed here. The equations used to establish the principle of optimality are called *functional equations*. Example 8.3 will be used to define the new terms.

Example 8.3 An established university is interested in developing a new College for Information Technology during the current year. Table 8.2 illustrates the data

Table 8.2 Faculty Hiring Policy Data for Example 8.3

Item	Fall	Winter	Spring	Summer
Required Faculty	5	10	8	2
Recruitment Cost	10	10	10	10
Unit Faculty Cost	1	2	2	1
Unit Retaining Cost	2	2	2	2

necessary to calculate the cost of recruiting faculty for this enterprise for the first year of operation. The college will operate during four quarters (Fall, Winter, Spring, and Summer) indicated by the columns. Decisions are associated with the beginning of the quarters. The first row indicates the new faculty required to implement the courses that will be offered. The second row is the recruitment cost for new faculty each quarter, which is significant because of the competitive demand for qualified people in this area. The third row is the unit cost of new faculty in terms of the standard institute faculty cost. Course specialization and availability makes this cost different in different quarters. The last row is the unit cost for retaining extra faculty hired in the earlier quarter (to minimize cost over the year). What should be the hiring policy for the university to minimize cost?

This example is clearly illustrative of sequential decision making. Decisions are required to be made every quarter. For example, it is possible to recruit all needed faculty during the first quarter and pay the retaining cost, or it may cost less to hire during each quarter, or recruit for first three quarters and hire again in the fourth. The problem will be solved using an *optimal/shortest path* method even though there are no routes involved in the problem. The best ways to understand the shortest path algorithm is to draw a directed graph (digraph) between the nodes/states in the problem.

States: States represent points/conditions in DP where decisions have to be considered. These are sometimes referred to as *policy* decisions or just policies. This has a sequential connection as current decisions cannot be considered until the previous decisions have been made. Clearly in Example 8.3 they represent the beginning of each quarter. The number of states is usually one more than the number of decisions to incorporate the effect of the last policy/decision. *Nodes* in the digraph represent *states* in the DP problem. The digraph represents a connection between each pair of states that is permissible. This can be made clearer by stating that if a node is going to be affected by a decision at any previous node, then there is a direct connection (line/arc) between the nodes. The cost of the connection is indicated on the connection itself. An arrow can be used to indicate the direction of traverse along this line/arc segment. These lines/arcs therefore represent *decisions*. Once the digraph for the DP problem is available, the *optimum solution corresponds to the shortest path from the beginning to the ending state of the digraph* [3].

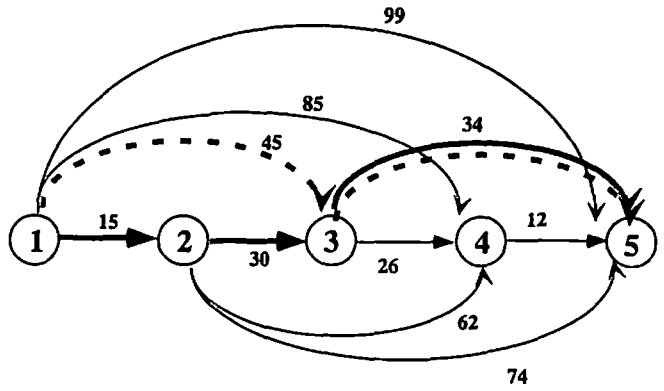

Figure 8.6 Digraph for Example 8.3.

Figure 8.6 is the digraph for the hiring policy. There are five states with the first four representing the quarters where the decisions must be made. Node 1 represents the Fall quarter and so on. Along the arcs on the digraph is the cost of the decision or policy that will originate at the current node and meet all of the requirements at the latter node. In order to use the information in the digraph effectively, and to develop the mathematical model, the quarters are assigned indices from 1 to 4. The following abbreviations are used: required faculty, *RF*; recruitment cost, *RC*; unit faculty cost, *FC*; unit retaining cost – holding cost, *HC*. The policy decisions or cost between the start node i and the end node j is $C(i, j)$. For example, $C(1, 4)$ represents the policy to hire in the Fall quarter, which will meet the staffing requirements for the Winter and Spring quarters also thus paying the appropriate retaining penalties:

$$C(1,4) = RC(1) + [RF(1) + RF(2) + RF(3)]*FC(1) + \{[RF(2) + RF(3)]*HC(1) + [RF(3)]*HC(2)\}$$

Ex8_3.m is the m-file that generates the data for the digraph for Example 8.3 and identifies the solution by solving the *functional equations* and backtracking the path to identify the optimal sequence. Note that for different problems only the cost calculation will differ. The functional equations and backtracking are elements of the DDP method and should be the same.

Functional Equation for Dynamic Programming: The functional equation is the essential part of the DDP algorithm or method. It employs a recursive procedure to identify the best solution at each state. Let *Value* (i) be the best value of the decision at node i, taking into account all possible arcs/paths that lead to node i. Let *Node* (i) be the node that led to the best policy at the current state. The recursion relations for n states can be set up as follows:

$$Value\ (1) = 0$$

$$Value\ (k) = \min\ \{Value\ (l) + C(l, k);\ 1 \le l < k\},\quad k = 2, \ldots, n+1$$

For example, in Example 8.3

$$Value\ (4) = \min\ \{Value\ (1) + C(1, 4);\ Value\ (2) + C(2, 4);\ Value\ (3) + C(3, 4)\}$$

$$Value\ (4) = \min\ \{[0 + 85];\ [15 + 62];\ [45 + 26]\} = 71$$

$$Node\ (4) = 3 \text{ (which established the minimum value)}$$

These calculations are embedded in **Ex8_3.m**. The output from the program is:

```
Example 8.3–Dynamic Programming
  Cost between pairs of nodes/states C(i,j)
  0   15   45   85   99
  0    0   30   62   74
  0    0    0   26   34
  0    0    0    0   12
  0    0    0    0    0
  Optimal path values at various nodes - Values(i)
  0   15   45   71   79
  Best previous node to reach current node -Node(i)
  0    1    2    3    3
```

The *Value* (4) is 71 and *Node* (4) is 3. The optimum value for the problem is *Value* (5), which is equal to 79.

Shortest Path: The shortest path is obtained by backtracking from the end state using the *Node* (i) information—which is the row of data above.

At Node 5 the best node from which to reach Node 5 is Node 3

0 1 2 3 **3**

At Node 3 the best node from which to reach Node 3 is Node 2

0 1 **2** 3 3

At Node 2 the best node from which to reach Node 2 is Node 1 (this is the starting node)

0 **1** 2 3 3

This solution is indicated by the thick solid line in Figure 8.6.

The actual hiring policy according to this solution is to hire just the required faculty for the Fall quarter in the Fall quarter. Do the same for the Winter quarter. For the Spring quarter, however, hire for both the Spring and Summer quarters.

Alternate Paths: There is an alternate solution for the *same optimal value*. The output from **Ex8_3.m** includes this information though it was not included above. In the output in the command window, each row contains additional nodes that will yield same optimal value. The only entry is 1 in third row and second column. This suggests that at the third node, *Node* (3) can also have the value of 1—that is, Node 1 also provides the same value to reach Node 3. This will cause the *Node* (*i*) information to change as

0 1 1 3 3

The shortest path can then be interpreted as follows:

At Node 5 the best node from which to reach Node 5 is Node 3

0 1 1 3 **3**

At Node 3 the best node from which to reach Node 3 is Node 1

0 1 **1** 3 3

In Figure 8.6 this is indicated by the thick dashed arcs. The hiring policy now would be to hire for Fall and Winter in Fall. Then hire for Spring and Summer in Spring.

Algorithm: Dynamic Programming (A8.3)

The following algorithm is identified in Reference 3 for paths that do not cycle. The original Bellman algorithm includes an iterative refinement that allows for cyclic paths. In Example 8.3 there are no cyclic paths. This algorithm is implemented in **Ex8_3.m**.

Step 0: Number nodes in sequential order. Arcs are defined for $j > i$. Set the start node s optimal value as

$$Value\ (s) = 0$$

Step 1: Terminate if all *Value* (*k*) have been calculated. If p represents the last unprocessed node, then

$$Value\ (p) = \min\{Value\ (i) + C\ (i, p)\text{: if arc } (i, p) \text{ exists } 1 \le i < p\}$$

$$Node\ (p) = \text{the number of Node } i \text{ that achieved the minimum above}$$

Return to Step 1

This was a simple example of DDP. It is important to note that solution in final *Value* (*i*) includes only the best solution at all states/nodes. Like other techniques, there are several different categories of problems addressed by DDP, including special discrete problems like the Knapsack problems. It is also possible to bring Examples 8.1 and 8.2 under the framework of DDP, though to impart a sequential nature in the process of selecting design variables in this case may involve a total transformation of the problem. Even then it may not be efficient to solve it using DDP. In practice, DDP has always been regarded as computationally complex and prohibitive. It is seldom used for problems with a large number of states, or if the estimation of cost is intensive. Another criticism of DDP is that it carries out a lot of redundant computation for continuous problems. Today, these are less of an issue than in the past. For stochastic and discrete problems DDP may still be an effective tool. Bellman has also indicated that it can be used to solve a set of unstable systems of equations. DDP also computes design sensitivity as part of its solution. This section was primarily meant to introduce DDP as a useful tool for some categories of problems. Readers are encouraged to consult the references for more information on the subject.

8.3 ADDITIONAL EXAMPLES

In this section, two additional examples are presented. The first one is an engineering example from mechanical design and illustrates how the discrete version of a four-variable problem very effectively *reduces to a single-variable problem* that can be easily addressed by enumeration. An unbelievable reduction in complexity yielding global optimization is obtained by elementary calculations. It definitely invites scrutiny regarding the effort in solving continuous nonlinear optimization problems. The second one is the application of the branch and bound search to a 0–1 integer programming problem. Much of the application of discrete optimization is in the area of decision making, and decisions like for or against, yes or no, are possible using a 0–1 linear integer programming model.

8.3.1 Example 8.4—I Beam Design

This example is a structural design problem. In fact, it is another variation/application of the I-beam structure seen earlier. The continuous problem was developed and solved as part of a student project. In this example the beam is used as a horizontal member in a fixed "E" series gantry crane. The hoist, with a capacity of $W = 2000$ lb, is at the center of the beam. The objective is to design a minimum mass beam that will not fail under bending, shear, and specified deflection. The length L of the beam is specified as 96 in. Steel was chosen for the material of the beam. The modulus of elasticity of steel is $E = 29$ E+06 psi. The weight density of steel is $\rho = 0.284$ lb/in^3. The yield strength in tension is $\sigma_y = 36$ E+03 psi. The yield strength in shear is $\tau_y =$ 21 E+03 psi. The specified allowable deflection is 0.25 in. A factor of safety of 1.4 on the load is assumed. Figure 8.7 illustrates the centric loading on the beam as well as the design variables. (All calculations are in inches and pounds.)

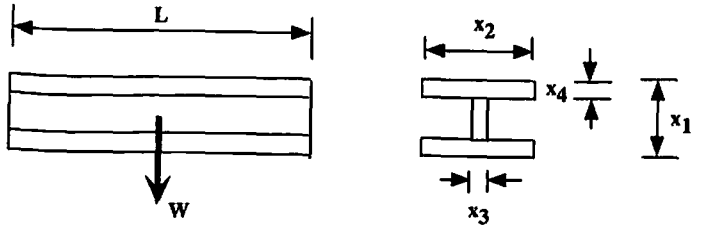

Figure 8.7 Design variables: Example 8.4.

The mathematical model for this problem can be expressed as:

$$\text{Minimize} \quad f(\mathbf{X}) = \rho L A_c$$

$$\text{Subject to:} \quad g_1(\mathbf{X}): WL^3 - 12EI \le 0 \quad \text{(deflection)}$$

$$g_2(\mathbf{X}): WLx_1 - 8I\sigma_y \le 0 \quad \text{(normal stress)}$$

$$g_3(\mathbf{X}): WQ_c - 2Ix_3\tau_y \le 0 \quad \text{(shear stress)}$$

Some geometric constraints to relate the various design variables are

$$g_4(\mathbf{X}): x_1 - 3x_2 \le 0$$

$$g_5(\mathbf{X}): 2x_2 - x_1 \le 0$$

$$g_6(\mathbf{X}): x_3 - 1.5x_4 \le 0$$

$$g_7(\mathbf{X}): 0.5x_4 - x_3 \le 0$$

The side constraints on the design variables are

$$3 \le x_1 \le 20, \quad 2 \le x_2 \le 15, \quad 0.125 \le x_3 \le 0.75, \quad 0.25 \le x_4 \le 1.25$$

The following relations define the cross-sectional area (A_c), moment of inertia (I), and first moment of area about the centroid (Q_c):

$$A_c = 2x_2x_4 + x_1x_3 - 2x_3x_4$$

$$I = \frac{x_2x_1^3}{12} - \frac{(x_2 - x_4)(x_1 - 2x_3)^3}{12}$$

$$Q_c = 0.5x_2x_4(x_1 - x_4) + 0.5x_3(x_1 - x_4)^2$$

The solution to the continuous relaxation model is

$$x_1^* = 5.4328; \quad x_2^* = 2.00; \quad x_3^* = 0.1563; \quad x_4^* = 0.25; \quad f^* = 48.239$$

The Discrete Problem: The discrete problem shares the same mathematical model except the values for the design variables will belong to a set of discrete values. It is possible to use either the complete enumeration or the BB search to solve the problem. Before proceeding further in this direction, it is useful to recognize that the reason for discrete optimization is to choose an off-the-shelf I-beam which will keep the cost and production time down. Several mills provide information on standard rolling stock they manufacture. Reference [7] is used to collect data for this example. If a standard S-beam is needed, the reference lists several beams with actual values for the design variables x_1, x_2, x_3, x_4. Choosing an I-beam from the list also identifies all of the design variables. This becomes a *single-variable problem*, the variable being the particular item from the list of beams. In this case there are no geometrical constraints necessary and neither are the side constraints. A complete enumeration can be performed on the candidate beams from the stock list, and the best beam can easily be identified. Compare this enumeration to solving the nonlinear mathematical model with seven functional constraints. When such an opportunity arises, exploiting the discrete optimization problem is highly recommended. On the other hand, consider a different problem where a plastic I-beam will be molded in the manufacturing facility. Here the continuous problem needs to be solved for the minimum cost.

Ex8_4.m will perform a complete enumeration of this example. It includes a truncated stock list from which the final solution is obtained. The programming is straightforward. The output in the Command window is

```
Solution found
---------------
Optimum Beam is :W6X9
Mass of the Beam : 71.543 (lb)
Depth, Width, Web Thickness, Flange Thickness (inches)
   5.900   3.940   0.170   0.215
Constraints: Deflection, Normal Stress, Shear Stress
(<= 0)
  -2422048221.600   -2468680.570   -86086.261
```

Note that the solution is significantly higher than the one obtained in the continuous relaxation, once again illustrating that the continuous relaxation provides a lower bound on the solution.

8.3.2 Zero–One Integer Programming

Many LP programs/models have special features which are usually exploited through special algorithms. An important class of LP problems require the variables be binary, that is, they can have one of two values. These problems can be set up so that values

for the variables are zero or one—defining a Zero–One Integer Programming Problem (ZIP). The general form of a ZIP is

$$\text{Minimize} \quad f(\mathbf{X}): \sum_{j=1}^{n} c_j x_j$$

$$\text{Subject to:} \quad \sum_{j=1}^{n} a_{ij} x_j \le b_i, \quad i = 1, 2, \ldots, m$$

$$x_j = 0 \text{ or } 1, \quad j = 1, 2, \ldots, n$$

Except for the binary side constraints on the design variables, this represents the mathematical model for an LP problem (the standard model requires equality constraints only). Most algorithms assume the c_j are positive. If any c_j is negative, then $1 - x_j$ is substituted for x_j in the problem.

Several varieties of optimization/decision problems involve binary variables. They include the knapsack (a pure ILP with a single main constraint), capital budgeting (a multidimensional knapsack problem), assembly line balancing, matching, set covering and facility location, and so on. Some examples are detailed below.

Example of a Knapsack Problem: Minimize the number of pennies (x_1), nickels (x_2), dimes (x_3), and quarters (x_4) to provide correct change for b cents:

$$\text{Minimize} \quad x_1 + x_2 + x_3 + x_4$$

$$\text{Subject to:} \quad x_1 + 5x_2 + 10x_3 + 25x_4 = b$$

$$x_1, x_2, x_3, x_4 \quad \text{are integers}$$

In the above problem, the unknown variables are not binary variables but are integers nonetheless.

Example of Capital Budgeting: The College of Engineering is considering developing a unique Ph.D. program in Microsystems in the next three years. This expansion calls for significant investment in resources and personnel. Six independent focus areas have been identified for consideration, all of which cannot be developed. Table 8.3 presents the yearly cost associated with creating the various groups over the three years. The last line in the table represents the available budget (in millions of dollars) per year allotted to this development. The last column is the value (in $100,000) the group is expected to generate in the next 5 years.

The capital budgeting problem uses six binary variables ($x_1, \ldots, x_6$), one for each item in the table. The ZIP problem can be formulated as

$$\text{Maximize} \quad 20x_1 + 3x_2 + 30x_3 + 5x_4 + 10x_5 + 5x_6$$

$$\text{Subject to:} \quad 3x_1 + 2x_2 + x_3 + 2x_4 + 5x_5 \le 10$$

Table 8.3 Capital Budgeting Example

Item	Group	Year1	Year2	Year 3	Value
1	Systems	3	1	0	20
2	Microsystems	2	5	0	3
3	Software	1	3	1	30
4	Modeling	2	3	0	5
5	Materials	5	0	1	10
6	Photonics	0	2	2	5
Budget		10	8	3	

$$x_1 + 5x_2 + 3x_3 + 3x_4 + x_6 \le 8$$

$$x_3 + x_5 + 2x_6 \le 3$$

$$x_1, \ldots, x_6 \in [0 \text{ or } 1]$$

Note this is not a minimization problem as required by the standard format.

Algorithms: There are several algorithms that have been developed for solving ZIP models. They all use implicit/selective enumeration through Branch and Bound search, together with continuous relaxation. However, the binary nature of the variables is exploited when completing the partial solutions or traversing the tree. The Balas Zero–One Additive Algorithm/Method is one of the recommended techniques for this class of problems [6]. The algorithm from the same reference is included below.

Algorithm: Balas Zero–One Additive Algorithm (A8.4)
The enumeration is tracked by two vectors $\boldsymbol{U}$ and $\boldsymbol{X}$. Both are of length n. If a partial solution consists of $x_{j1}, x_{j2}, \ldots, x_{js}$ assigned in order, the values of $\boldsymbol{X}$ corresponding to the assigned variables are 0 or 1. The remaining variables are free and have the value −1. The components of $\boldsymbol{U}$ are

$$u_k = \begin{cases} j_k & \text{if } x_{jk} = 1 \text{ and its complement has not been considered} \\ -j_k & \text{if } x_{jk} = 1 \text{ or } 0 \text{ and its complement has been considered} \\ 0 & \text{if } k > s \end{cases}$$

Step 1. If $b \ge 0$, optimal solution is $\mathbf{X} = [0]$. Otherwise $\mathbf{U} = [0]$, $\mathbf{X} = [-1]$. $f^* = \infty$, $\mathbf{X}^* = \mathbf{X}$ (incumbent solution)

Step 2. Calculate

$$y_i = b_i - \sum_{j \in J} a_{ij} x_{ij}, \quad J \text{ is the set of assigned variables}$$

$$\bar{y} = \min y_i, \quad i = 1, 2, \ldots, m$$

$$f = \sum_{j \in J} c_j x_j$$

If $\bar{y} \geq 0$ and $f < f^*$ then $f^* = f$, and $\mathbf{X}^* = \mathbf{X}$ Go to Step 6
Otherwise continue

Step 3. Create a subset T of free variables x_j defined as

$\mathbf{T} = \{j: f + c_j < f^*, a_{ij} < 0 \quad \text{for } i \text{ such that } y_i < 0\}$

If $\mathbf{T} = \emptyset$ (empty set) then Go to Step 6, Else continue

Step 4. Infeasibility test:

If i such that $y_i < 0$ and

$$y_i - \sum_{j \in T} \min(0, a_{ij}) < 0$$

Go to Step 6, Else continue

Step 5. Balas branching test: For each free variable x_j create the set $\mathbf{M}_j$

$$\mathbf{M}_j = \{i: y_i - a_{ij} < 0\}$$

If all sets $\mathbf{M}_j$ are empty Go to Step 6. Otherwise Calculate for each free variable x_j

$$\nu_j = \sum_{i \in M_j} (y_i - a_{ij})$$

where $\nu_j = 0$ if the set $\mathbf{M}_j$ is empty. Add to the current partial solution the variable x_j that maximizes ν_j. Go to Step 2

Step 6. Modify U by setting the sign of the rightmost positive component. All elements to the right are 0. Return to Step 2

If there are no positive elements in U, the enumeration is complete. The optimal solution is in the incumbent vector $\mathbf{X}^*$. The objective is in f^*. If f^* is ∞, there is no feasible solution.

Balas.m is the m-file that implements the Balas algorithm. It was translated from a Pascal program available in Reference 6. It is a program and prompts the user for the problem information—n, m, A, b, c. The test problem given in Reference 6 is included below.

Example 8.5

$$\text{Minimize} \quad f(\mathbf{X}): 10x_1 + 14x_2 + 21x_3 + 42x_4$$

$$\text{Subject to:} \quad -8x_1 - 11x_2 - 9x_3 - 18x_4 \leq -12$$
$$-2x_1 - 2x_2 - 7x_3 - 14x_4 \leq -14$$
$$-9x_1 - 6x_2 - 3x_3 - 6x_4 \leq -10$$
$$x_1, x_2, x_3, x_4 \in [0\ 1]$$

The output from **Balas.m**:

```
Solution exists
-----------------
The variables are:
  1   0   0   1
The objective function : 52.00
Constraints - Value and Right hand side
  -26   -12
  -16   -14
  -15   -10
```

This corresponds to the solution identified in Reference 6.

Complete Enumeration: The Balas algorithm performs partial enumeration. The complete or exhaustive enumeration (n_e) for n binary variables is given by the relation

$$n_e = 2^n$$

Typically, enterprise decision making involves several hundred variables and linear models. On the other hand, engineering problems characterized by a nonlinear model may have about 20 binary variables requiring $n_e = 1{,}048{,}576$ enumerations. This is no big deal with desktop resources today. Keep in mind that complete enumeration avoids any relaxation solution (purely discrete problems only).

CallSimplex.m: This is a function m-file that can be used to solve LP programs. It can be used to obtain solutions to LP relaxation models during the BB search. It is based on the algorithm available in Reference 6. It is translated from a Pascal program. The mathematical model is

$$\text{Minimize} \quad c^T x$$
$$\text{Subject to:} \quad [\mathbf{A}][\mathbf{X}] = [\mathbf{b}]$$
$$x \geq 0$$

Slack variables are included in the coefficient matrix [A]. Greater than or equal to constraints can be set up with negative slack variables as the program will

automatically execute a dual-phase simplex method. Another alternative is to use the LP programs provided by MATLAB. The function statement for using **Call Simplex.m** is

```
function [xsol, f] = CallSimplex(A,B,C,M,N)
```

To use the function, A, B, C, M, N have to be defined. M is the number of constraints and N is the number of design variables.

REFERENCES

1. Fletcher, R., *Practical Methods of Optimization*, Wiley, New York, 1987.
2. Papalambros, P. Y., and Wilde, J. D., *Principles of Optimal Design—Modeling and Computation*, Cambridge University Press, New York, 1988.
3. Rardin, R. L., *Optimization in Operations Research*, Prentice-Hall, Englewood Cliffs, NJ, 1998.
4. Bellman, R., *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.
5. White, D. J. *Dynamic Programming*, Holden-Day, San Francisco, 1969.
6. Syslo, M. M., Deo, N., and Kowalik, J. S., *Discrete Optimization Algorithms*, Prentice-Hall, Englewood Cliffs, NJ, 1983.
7. *Ryerson Stock Steel List*, Joseph T. Ryerson, Inc., 1995.

PROBLEMS

8.1 Translate Example 1.1 into a discrete optimization problem and solve by exhaustive enumeration.

8.2 Translate Example 1.2 into a discrete optimization problem and solve by exhaustive enumeration.

8.3 Translate Example 1.3 into a discrete optimization problem and solve by exhaustive enumeration.

8.4 Implement a multistage exhaustive enumeration for the continuous optimization of Example 1.1.

8.5 Translate Example 1.1 into a discrete optimization problem and solve by branch and bound method.

8.6 Translate Example 1.2 into a discrete optimization problem and solve by branch and bound method.

8.7 Translate Example 1.3 into a discrete optimization problem and solve by branch and bound method.

8.8 Identify appropriate stock availability for the flagpole problem in Chapter 4 and solve the corresponding discrete optimization problem.

8.9 Solve Example 1.3 using the stock information available in **Ex8_4.m**.

8.10 You are the owner of an automobile dealership for Brand X cars. Set up a DP for monthly inventory control based on cars expected to be sold, high ordering costs, unit costs for the car, and inventory costs for excess cars. Solve it using the code in **Ex8_3.m**.

8.11 Solve the following using Balas algorithm—using complete enumeration

$$\begin{aligned} \text{Minimize} \quad & f(\mathbf{X}) = x_1 + x_2 + x_3 \\ \text{Subject to:} \quad & 3x_1 + 4x_2 + 5x_3 \geq 8 \\ & 2x_1 + 5x_2 + 3x_3 \geq 3 \\ & x_1, x_2, x_3 \in [0\ 1] \end{aligned}$$

8.12 Program an exhaustive enumeration technique for zero–one IP using MATLAB. Solve Problem 8.10.

8.13 Is it possible to solve the capital budgeting example? If so, what is the solution using complete enumeration?

8.14 Solve capital budgeting example using Balas program.

8.15 Create an example which involves making binary choices regarding your participation in course and extracurricular activities to maximize your grades. Solve it.

9

GLOBAL OPTIMIZATION

This chapter introduces two of the most popular techniques for finding the best value for the objective function, also referred to as the *global optimum* solution. They are Simulated Annealing (SA) and the Genetic Algorithm (GA). The names are not accidental since the ideas behind them are derived from *natural processes*. SA is based on cooling of metals to obtain defined crystalline structures based on minimum potential energy. GA is based on the combination and recombination of the genes in the biological system leading to improved DNA sequences or species selection. In actual application, they are more like a continuation of Chapter 8 (Discrete Optimization), because these methods can be used for continuous and discrete categories of the optimization problems. They share an important characteristic with other optimization techniques in that they are primarily *search* techniques—they identify the optimum by searching the design space for the solution. Unlike the gradient-based methods, these techniques are largely heuristic, generally involve large amounts of computation, and involve some statistical reasoning. Global optimization is being pursued seriously today attracting a significant amount of new research to the area of applied optimization after a long hiatus. A significant drawback to many of these techniques is that they require *empirical tuning* based on the class of problems being investigated. There is also no easy way to determine technique/problem-sensitive parameters to implement an automatic optimization technique.

The *global optimum* cannot really be characterized as being different from the *local optimum*. The latter is referenced when only a part of the design space (as opposed to the complete space) is viewed in the neighborhood of a particular design point. The standard Kuhn-Tucker conditions, established and used in earlier chapters (for continuous problems), will identify the local optimum only. This is also more emphatic in numerical investigations of continuous problems where the algorithms are expected to move toward the optimum values close to where they are started (initial

guess). While it makes sense to look for the global optimum, there is no method to identify if such an optimum exists for all classes of problems. At the present time, only continuous *convex* problems are guaranteed to have a global solution. It is not difficult to argue that such a class represents only a miniscule set of useful problems and most real optimization problems are not necessarily convex. Therefore, in the literature, both of these methods are expected to produce solutions close to the global optimum solution, rather the global solution themselves. This is not to suggest that investigation of the global solution is a waste of time if the possibility of one cannot be established with certainty. Enough justification can be advanced by competitive and financial gains to warrant an investigation, even without the guarantee of useful results. Global optimization techniques are simple to implement while requiring exhaustive calculations. This can potentially soak up all of the idle time on the personal desktop computing resource, lending it immediate attractiveness.

The techniques and their discussion are very basic following the theme espoused in Chapter 8. The reader is encouraged to visit journal publications to monitor the maturing of these techniques. In this chapter, first some issues regarding global optimization are presented, followed by presentation of the SA and GA.

9.1 PROBLEM DEFINITION

The following development on global optimization is based on *unconstrained* continuous optimization. Solving constrained problems does not appear to have evolved to a common technique and is not illustrated in this book. Note that constrained problems can be converted to an unconstrained problem using a penalty formulation (Chapter 7). The problem can be defined as

$$\text{Minimize} \quad f[\mathbf{X}];\ [\mathbf{X}]_n \tag{9.1}$$

$$\text{Subject to:} \quad x_i^l \le x_i \le x_i^u \tag{9.2}$$

If this problem were *unimodal* (only one minimum), the optimization could easily be established by any of the appropriate techniques used in the book since the local minimum is also the global minimum. To investigate the global minimum it is therefore expected that there would be many *local* minimums, several of them having different values for the objective. The least among them is then the *global* solution. Local minimums, in the case of continuous problems, are those that satisfy the KT conditions, and will be numerically determined if the starting guess was in the neighborhood of the particular minimum. This definition may be applied to discrete optimization problems that use continuous relaxation to obtain the solution. For discrete problems using complete enumeration, the global optimization should be discovered automatically. Therefore, the discussion applies to discrete problems that use only incomplete enumeration.

9.1.1 Global Minimum

The global minimum for the function $f(\mathbf{X})$ exists if the function is continuous on a nonempty feasible set S that is closed and bounded [1].

The set S defines a region of the design space that is feasible. Any point in S is a candidate design point $\mathbf{X}$. In the n-dimensional Cartesian space, the point $\mathbf{X}$ corresponds to a vector of length $\|\mathbf{X}\|$. The boundaries of the region are part of the set S (*closed*). The length of the vector $\mathbf{X}$ should be a finite number (*bounded*). The definition for the global minimum $\mathbf{X}^*$ can be stated as

$$f(\mathbf{X}^*) \leq f(\mathbf{X}); \quad \mathbf{X} \in S \tag{9.3}$$

The conditions for the existence of the global minimum as given above are attributed to Weierstrass. While the global minimum is guaranteed if the conditions are satisfied, they do not negate the existence of the global solution if conditions are not satisfied. Considering the broad range and type of optimization problems, the conditions are fairly limited in their usefulness. The conditions for the local minimum are the same as constraint (9.3) except the set S is limited to a small region around $\mathbf{X}^*$.

Convexity: If an optimization problem can be shown to be *convex*, then the local minimum will also be a global minimum. A convex optimization problem requires the objective function to be convex. If constraints are present, then all of the inequality constraints must be convex, while the equality constraints must be linear.

Convex Set: This usually relates to the design vector (or design points) of the problem. Consider two design points $\mathbf{X}_1$ and $\mathbf{X}_2$. As before, the set S defines a design region and by implication both $\mathbf{X}_1$ and $\mathbf{X}_2$ belong to S. The line joining $\mathbf{X}_1$ and $\mathbf{X}_2$ contains a whole series of new design points. If all of these points also belong to S, then S is a convex set. A closed set S requires that the points $\mathbf{X}_1$ and $\mathbf{X}_2$ can also be on the boundary. If, however, the line joining $\mathbf{X}_1$ and $\mathbf{X}_2$ contains some points that do not belong to S, then S is not a convex set. For a two-dimensional design problem, convex and nonconvex sets are illustrated in Figure 9.1.

Convex Functions: A convex function is defined only for a convex set. Once again let S be a convex set from which two design points $\mathbf{X}_1$ and $\mathbf{X}_2$ are selected. Let f be the function whose convexity is to be established. Consider any point $\mathbf{X}_a$ on a line joining $\mathbf{X}_1$ and $\mathbf{X}_2$. The function f is convex if $f(\mathbf{X}_a)$ is less than the value of the corresponding point on the line joining $f(\mathbf{X}_1)$ and $f(\mathbf{X}_2)$. In n dimensions this is not easy to see but a one-dimensional representation is presented in Figure 9.2. The condition for convexity in the figure is

$$f(x_a) \leq f^+ \tag{9.4}$$

A mathematical condition of convexity is that the Hessian matrix (matrix of second derivatives) of the function f must be positive semidefinite at all points in the set S. Note that this corresponds to the KT conditions for unconstrained minimization

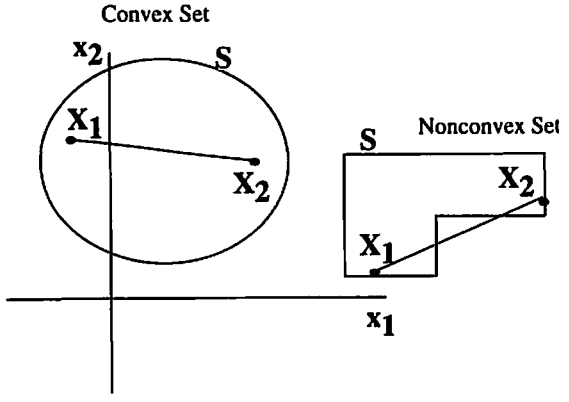

Figure 9.1 Convex and nonconvex set.

problems. From a practical viewpoint, the eigenvalues of the function must be greater than or equal to zero at all points in S.

It is important to realize that a global optimum may still exist even if the convexity conditions are not met. In real optimization problems, it may be difficult to establish convexity. In practice, convexity checks are rarely performed. An assumption is made that a global optimum exists for the problem. This is a subjective element reinforced by experience, insight, and intuition. This is also often justified by the possibility that a better solution than the current one can be found.

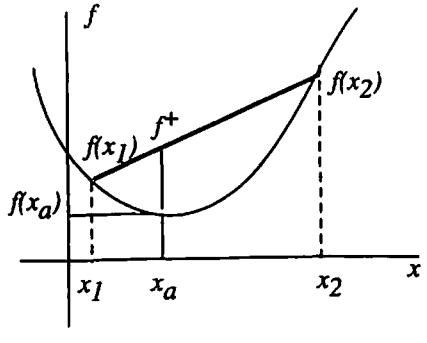

Figure 9.2 Convex function.

9.1.2 Nature of the Solution

It is assumed that the global optimum solution will be obtained numerically. A good understanding of the nature of the solution is possible with reference to a continuous design example. Extension to discrete problems is possible by considering that any continuous solution corresponds to a specific combination of the set of discrete values for the design variables. Figure 9.3 is an illustration of a figure with several minima and maxima with a strong global minimum at $x_1 = 4$, $x_2 = 4$. The figure, available through **fig9_3.m**,[1] is the plot of the function (also Example 9.1)

$$f(x_1, x_2) = -20 \frac{\sin\left(0.1 + \sqrt{(x_1 - 4)^2 + (x_2 - 4)^2}\right)}{0.1 + \sqrt{(x_1 - 4)^2 + (x_2 - 4)^2}} \tag{9.5}$$

Figure 9.4 illustrates the contour plot. The contour plot can be better appreciated in color using the colorbar for interpretation. Running the m-file will create both figures. From relation (9.5) it is clear the ripples will subsidize as x_1 and x_2 move much farther away from where the global minimum occurs, because of the influence of the denominator. The numerator is constrained to lie between −1 and +1 multiplied by the

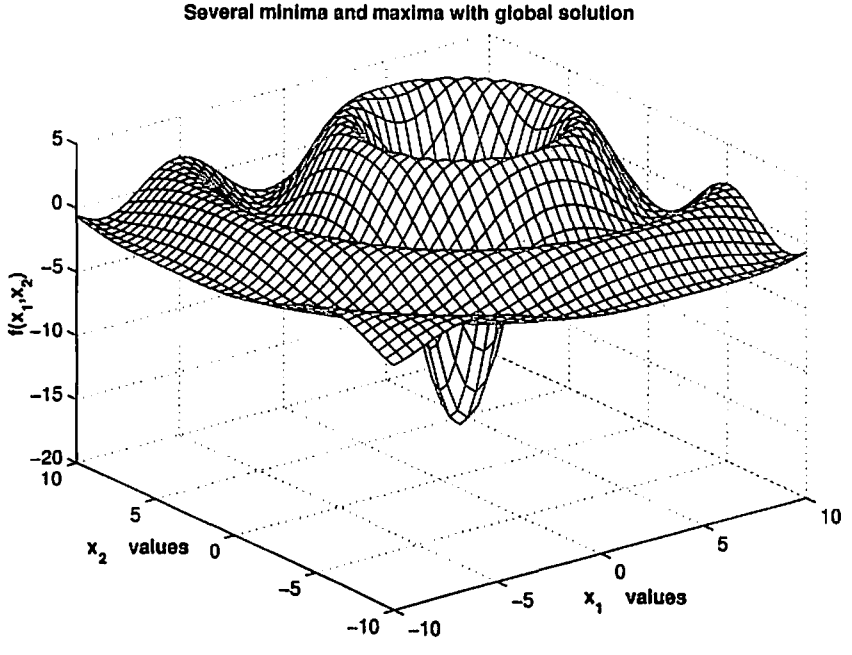

Figure 9.3 Surface plot of Example 9.1.

[1]Files to be downloaded from the web site are indicated by boldface sans serif type.

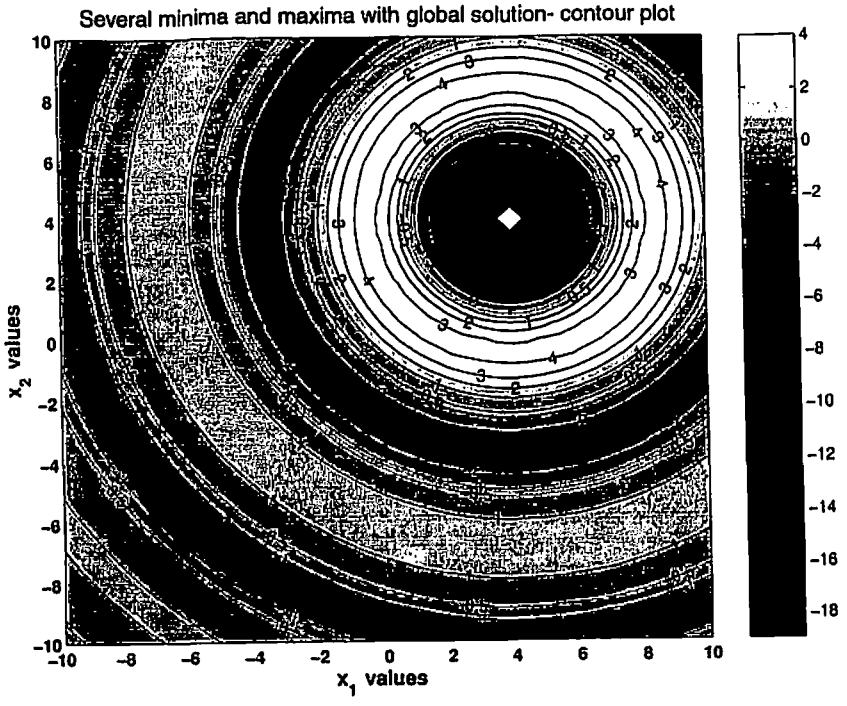

Figure 9.4 Contour plot of Example 9.1.

scaling factor of 20. In the limit the ripples should start to become a plane. This can be confirmed by executing **fig9_3.m** with different ranges for the variables.

From Figure 9.3 it is easy to spot the global solution by observation. Surrounding the global minimum are rings of local extrema. Numerically it is a challenge to negotiate all of the bumps and dips. A simple way to characterize numerical techniques for a minimization problem is that they attempt to find a ditch (or valley) to fall into. If this is treated as an unconstrained minimum problem, the KT conditions will lead the solution to be trapped in any one of the local minimum rings, depending on the location of the starting point for the iterations. If the iterations are started close to $x_1 = 4$, $x_2 = 4$, then it is more than likely the global optimum will be discovered. Staring at $x_1 = -5, x_2 = -5$, using standard numerical techniques will probably not lead to the global minimum.

This is a simple illustration but note there are infinite local minimums and maximums and a single global minimum. In real design problems with more than 2 variables, it is not readily apparent how many local minimums exist, and if the global minimum is really that distinct, even if it is postulated to exist. What about the complete enumeration technique? There is no way the global minimum can escape the

sifting due to this technique. There is also no technique that is simpler than complete enumeration. The problem is the time to finish searching the infinite points of the design space to discover the solution.

The solution to global optimum must therefore lie between standard numerical techniques and complete enumeration. The former will not find the solution unless it is fortunate. The latter cannot find the solution in time to be useful. Global optimization techniques tend to favor the latter approach because of the simplicity of implementation. To be useful the solution must be discovered in a reasonable time.

9.1.3 Elements of a Numerical Technique

The problems faced by a numerical technique searching for the global minimum have been touched on in the previous section. In most standard numerical techniques the local minimum serves as a strong magnet or a beacon for the solution. In most cases, design changes are not accepted if there is no improvement in the solution. If standard numerical techniques have to be adapted to look for global minimum, then they must encourage solutions that in the short run may increase the objective value so that they can find another valley at whose bottom is the global minimum. They must achieve this without sacrificing the significant potency of the local minimum, because a local minimum solution is better than no solution at all. Figure 9.5 illustrates these ideas using a single variable.

In Figure 9.5 the iterations start at point A. In a standard numerical technique, a search direction is found which leads to point B. At B, if conventional numerical techniques were used, a search direction leading to point C will be accepted, while a search direction leading to point D will be strongly discouraged. But, if point D was somehow reachable, then it could move on to E and then F and continue to the global

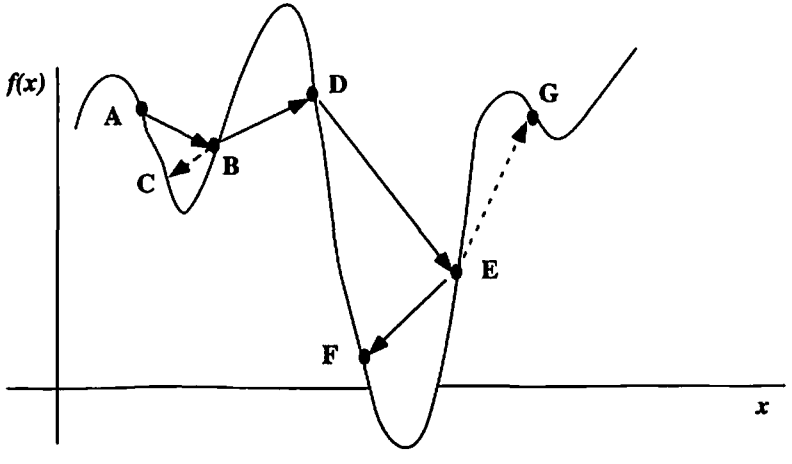

Figure 9.5 Strategy for global optimum search.

minimum. The main idea in finding the global minimum is to encourage solutions that will escape the attraction of the local minimum. This means at point E it is quite possible that the search direction may lead to point G. It is then required that a large number of iterations be permitted so that the valley holding the global minimum will be revisited. From this brief illustration the characteristics for a technique used to find a global minimum can be summarized as:

- Short-term increases in the objective function must be encouraged.
- Traditional gradient-based techniques will not be particularly useful because they are based on search directions that decrease the objective function at the current iteration. They will not permit increase of the objective function value.
- If descent-based search directions are *not* part of the technique, then any direction that decreases the objective should be accepted to promote attraction by the local minimum.
- As the location of the global minimum is not known, search must be permitted over a large region with no bias, and for a large number of iterations.
- Since acceptable directions can decrease or increase the objective function value, a heuristic determination of the search direction (or design change) must be part of the technique.
- These features resemble the zero-order technique for unconstrained optimization and therefore will require a large number of iterations to be effective. The large number of iterations will force the technique to be simple.
- The solution can only be concluded based on the number of iterations, as necessary conditions (available for continuous problems only) cannot distinguish between local and global optimum.

Without a doubt, the best technique is complete or exhaustive enumeration. It is therefore necessary to assume that complete enumeration *is not an option* in further discussion.

The problem description, illustration, and discussion have dealt with unconstrained minimization. Constraints can be included by addressing the above characteristics to apply to feasible solutions only. It is left to the reader to contemplate the ability to heuristically generate a large number of feasible design changes for consideration.

9.2 NUMERICAL TECHNIQUES AND ADDITIONAL EXAMPLES

This section introduces two popular methods for global optimization. Both are derived from natural physical processes that help identify optimum states. The first one, Simulated Annealing (SA), refers to a physical process that establishes different crystalline structures, each of which can be associated with different minimum potential energy. This is attempted by reheating and cooling of the materials/metals while controlling the rate of cooling. The second, the Genetic Algorithm (GA), works through mutation of candidate solutions until the best solution can be established, a

process that can be related to natural solution or biological evolution. These techniques are illustrated with respect to unconstrained problems in this section.

9.2.1 Simulated Annealing (SA)

The first application using SA for optimization was presented by Kirkpatrick et al. [2]. The examples covered problems from electronic systems, in particular, component placement driven by the need for minimum wiring connections. The algorithm itself was based on the use of statistical mechanics, demonstrated by Metropolis et al. [3], to establish thermal equilibrium in a collection of atoms. The term itself is more representative of the process of cooling materials, particularly metals, after raising their temperature to achieve a definite crystalline state. As mentioned earlier, the extension to optimization is mainly heuristic. The simple idea behind SA can be illustrated through the modified basic algorithm of SA due to Bochavesky et al. [4].

Algorithm: Basic Simulated Annealing (A9.1)

Step 1. Choose starting design $\mathbf{X}_0$. Calculate $f_0 = f(\mathbf{X}_0)$

(Need stopping criteria)

Step 2. Choose a random point on the surface of a unit n-dimensional hypersphere (it is a sphere in n-dimensions) to establish a search direction S

Step 3. Using a stepsize α, calculate

$$f_1 = f(\mathbf{X}_0 + \alpha S); \quad \Delta f = f_0 - f_1$$

Step 4. If $\Delta f \le 0$, then $p = 1$

Else $p = e^{-\beta \Delta f}$

Step 5. A random number r $(0 \le r < 1)$ is generated.

If $r \le p$, then the step is accepted and the design vector is updated

Else, no change is made to the design

Go to Step 2

If implemented, the algorithm has to executed for a reasonably large number of iterations for the solution to drift to a global minimum. The simplicity of the algorithm is very appealing. A careful observation will indicate that α and β are important parameters for the algorithm and they are likely to be problem dependent. Generally, the parameter β can be related to the Boltzmann probability distribution. In statistical mechanics it is expressed as $-k/T$; T is considered the annealing temperature, and k the Boltzmann constant. It dictates the conditional probability that a *worse* solution can be accepted. For the algorithm to be effective, it is recommended that p be in the range $0.5 \le p \le 0.9$. In formal SA terminology, β represents the annealing temperature. The following ideas in the algorithm are apparent:

- The algorithm is heuristic.
- The algorithm is suggestive of a biased random walk.
- Descent of the function is directly accommodated.
- Directions that increase the value of the function are sometimes permitted. This is important to escape local optimum. This is the *potential* to discover the global optimum through the algorithm.

This basic algorithm can definitely be improved. During the initial iterations it is necessary to encourage solutions to escape the local optimum. After a reasonably large number of iterations, the opportunity to escape local optimum (window of opportunity) must be continuously reduced so that the solution does not wander all over the place, and cancel the improvements to the solution that have already taken place. This can be implemented by scheduling changes in the parameter β as the iterations increase. In SA terminology, this is referred to as the *annealing schedule*. These modifications do not in any way decrease the problem/user dependence of the algorithm. This problem dependence is usually unattractive for automatic or standard implementation of the SA method. Researchers are engaged in construction of algorithms that do not depend on user-specified algorithm control parameters [5].

The SA algorithm for continuous variables implemented in the book includes many of the basic ideas from the original algorithm. There are many additional changes noted here. First the negative sign in the probability term is folded into β itself. A simple procedure is included for obtaining the value of β. It is based on the initial value of $\mathbf{X}_0$ and f_0. β is calculated to yield a value of $p = 0.7$ for a Δf equal to 50% of the initial f_0. The algorithm favors a negative value of β by design. Since the process for establishing an initial β illustrated above can determine a positive value, the algorithm corrects this occurrence. It is later illustrated through numerical experiments that a small magnitude of the parameter β allows the solution to wander all over the place, while a large value provides a strong local attraction. Instead of the SA step being incorporated when the objective increases, it is triggered when the objective is likely to increase based on the slope. In the algorithm below, the control parameter β is kept constant at its initial value. The calculation of α is borrowed from the traditional one-dimensional stepsize calculation using the Golden Section method (Chapter 5) if the objective is likely to decrease. If the slope of the objective is positive or zero, a fixed α is chosen. This value is a maximum of one of the previous values computed through the Golden Section calculations. Note that this does not ensure that the function is actually going to increase as required in the basic algorithm. Nevertheless, it appears to work as illustrated in the figures below. Stopping criteria are not built in and are left to the user. The algorithm is effective for Example 9.1. It has yet to be tested on a wide range of problems. It is also shown to work for Example 9.2.

Algorithm: Simulated Annealing (A9.2)

Step 1. Choose $\mathbf{X}_0$. Calculate $f(\mathbf{X}_0)$, $p = 0.7$, $\beta = -\ln(p)/0.5 \times f(\mathbf{X}_0)$

If $|\beta| < 1$, $\beta = 1$, If $\beta > 0$, $\beta = -1 \times \beta$

$i = 1$

Step 2. Choose a random point on the surface of a unit n-dimensional hypersphere to establish a search direction S

Step 3. Calculate $\nabla f(\mathbf{X}_0)$;

If $\nabla f(\mathbf{X}_0) < 0$, then compute 1-D α^*

$\Delta \mathbf{X} = \alpha^* S$; $\bar{\mathbf{X}} = \mathbf{X}_0 + \Delta \mathbf{X}$; $\mathbf{X}_0 \leftarrow \bar{\mathbf{X}}$; $i \leftarrow i + 1$

If $\nabla f(\mathbf{X}_0) \geq 0$,

$\Delta f = f(\mathbf{X}_0 + \tilde{\alpha} S) - f(\mathbf{X}_0)$; $\tilde{\alpha} = \max(\alpha^*, 1)$

$p = e^{\beta \Delta f}$

A random number r; $(0 \leq r < 1)$ is generated.

If $r \leq p$, then the step is accepted and the design vector is updated

$\Delta \mathbf{X} = \tilde{\alpha} S$; $\bar{\mathbf{X}} = \mathbf{X}_0 + \Delta \mathbf{X}$; $\mathbf{X}_0 \leftarrow \bar{\mathbf{X}}$; $i \leftarrow i + 1$

Else continue $i \leftarrow i + 1$

Go to Step 2

The algorithm is available in **SimulatedAnnealing.m**. It is called through **Call_SA.m**. The algorithm is applied to Example 9.1, which was used for illustration in Section 9.1. To execute the algorithm the following m-files (introduced in Chapter 6) are necessary: **UpperBound_nVar, GoldSection_nVar.m**. Following the practice of drawing contours for two-variable problems, two files are necessary for describing the example, the second primarily to quickly draw a contour plot using matrix calculations. In this case the files are **Example9_1.m** and **Example9_1plot.m**. The latter file name must be set in **SimulatedAnnealing.m**. The calling statement and variables are explained in **SimulatedAnnealing.m**. They can be read by typing

```
» help SimulatedAnnealing
```

in the command window (the current directory must contain the files)

The reader is strongly encouraged to run the example on his computer. It would be a singular coincidence if any of the results presented here are duplicated. Running numerical experiments is simply a matter of calling the SA algorithm and taking a look at the results (especially the plot). The design changes are illustrated using colors: red—representing decreasing slopes—and blue—for positive slopes that lead to design changes. The entrapment in the local minimum and the escape are clearly illustrated on the contour plots. The contour plots are filled color plots to identify areas on the figure where the function has a local minimum or maximum. Darker colors represent minimums and lighter colors the maximum. The examples were run usually for 1000 iterations. It is conceivable that they could have been stopped earlier. On a 200-Hz PII laptop it required less than a minute for each run. A large number of iterations is useful to indicate if the solution escapes the global minimum once having obtained it. While it was not evident in this example, it is likely problem dependent since in this example the global minimum was significantly larger than the local

minimums. Several features of the SA algorithm are illustrated using some numerical experiments.

Example 9.1 This is the function used in Section 9.1. The global minimum is at $\mathbf{X}^* = [4\ 4]$ with an objective value $f^* = -19.6683$. In the figures, the contours suggest a strong global minimum encircled by alternate local maximums and minimums. For most of the runs the starting value was far from the global minimum.

$$\text{Minimize} \quad f(x_1, x_2) = -20 \frac{\sin\left(0.1 + \sqrt{(x_1 - 4)^2 + (x_2 - 4)^2}\right)}{0.1 + \sqrt{(x_1 - 4)^2 + (x_2 - 4)^2}} \tag{9.5}$$

Five numerical experiments are presented below. It should be emphasized that these runs cannot be duplicated. Overall, the algorithm worked very well. For exactly the same starting values and number of iterations (calling **Call_SA.m** without any changes—about 40 times) the global minimum was found 95% of the time.

Run 1: Starting design = [−5, −5]; $\beta = -2.7567$; Number of iterations = 1000

Useful number of iterations (when design changes occurred) = 383

Final design = [3.9992, 3.9997]; Final Objective = −19.6683

Figure 9.6 illustrates the trace of the design changes. It escapes the starting relative minimum fairly easily to get trapped by a different local minimum. It is able to escape to the global minimum where it is strongly attracted to it. The traces in the local minimum appear to prefer to stay in the local minimum suggesting the solutions are not being encouraged to escape.

Run 2: Starting design = [4, 4]; $\beta = -0.0362$; Number of iterations = 1000

Useful number of iterations (when design changes occurred) = 991

Final design = [100.72, 127.75]; Final Objective = 0.128

Figure 9.7 illustrates the trace of the search for the global minimum. Most of the points are not in the plot as can be seen from the solution. This solution was started *at the global minimum*. The search process appears to wander all over the place and finally off the map. There is no pattern in the limited points seen on the figure, except the suggestion that the search is encouraged to escape. This can be tied to the low magnitude of β.

Run 3: Starting design = [4, 4]; $\beta = -1$; Number of iterations = 1000

Useful number of iterations (when design changes occurred) = 553

Final design = [3.8832, 4.3709]; Final Objective = −19.173

Figure 9.8 illustrates the trace of the design search. In this run the value of β was set to −1. Once again the starting value of the design was the global minimum itself. From the figure it is clear that the search is not allowed to escape the global minimum that easily. The choice of the control parameter was arbitrary, and several examples have to be studied to see if this value is problem independent. The value of 1 is typically a

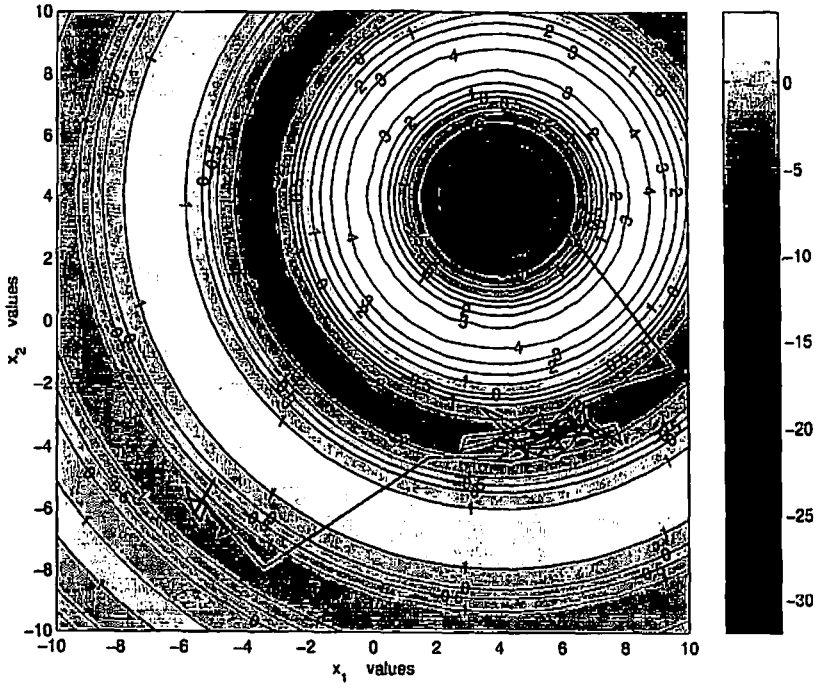

Figure 9.6 Simulated annealing Example 9.1—converged solution.

good choice for optimization algorithms, particularly for a well-scaled problem. What is clear from the runs so far is that a negative value of β of the order of 1 implements a local attractor.

Run 4: Starting design = [−5, −5]; β = −2.7567; Number of iterations = 1000

Useful number of iterations (when design changes occurred) = 558

Final design = [−3.3397, 6.2019]; Final Objective = −2.5674

This is a repeat of Run 1: Figure 9.9 traces the search for the global minimum for this run. It appears that 1000 iterations is not sufficient to determine the global minimum. While the solution has escaped the starting local minimum, it has not yet escaped the next local minimum. Since the valleys (local minimums) in this example are circular, note that the solution moves around in the valley but has not managed to escape.

Run 5: Starting design = [−5, −5]; β = −2.7567; Number of iterations = 10,000

Useful number of iterations (when design changes occurred) = 3813

Final design = [4.0037, 3.9996]; Final Objective = −19.6682

No figure is presented. The solution converges to the global minimum.

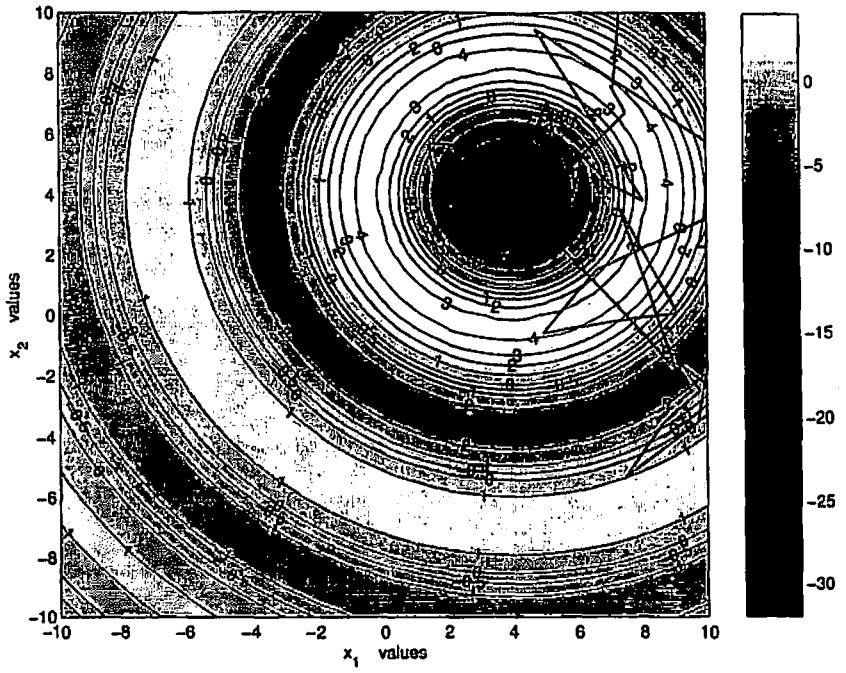

Figure 9.7 Simulated annealing—low β.

Discussion on the Numerical Experiments: The following observations are based on the experience of applying SA (A9.2) to Example 9.1. They correspond to the reported experience with SA in the general literature.

- Annealing schedule (changing β) is an important consideration in SA. Starting with a low value the control parameter β must be increased—not too fast or it will get trapped in a local minimum.
- Annealing schedule is developed through trial and error, insight that is not obvious, and based on probable behavior of the search.
- Length of time (or number of iterations) that the problem needs to be searched is difficult to determine. While this can be based on monitoring, given the stochastic nature of the search, it cannot be asserted that the global minimum has been established. The SA algorithm itself can be expected to converge while this property is too slow for practical use [6]. A larger number of iterations is more likely to yield the global solution.
- SA search does not subscribe to iterative improvement. It is an example of a stochastic evolutionary process. The implementation used here represents a memoryless search.

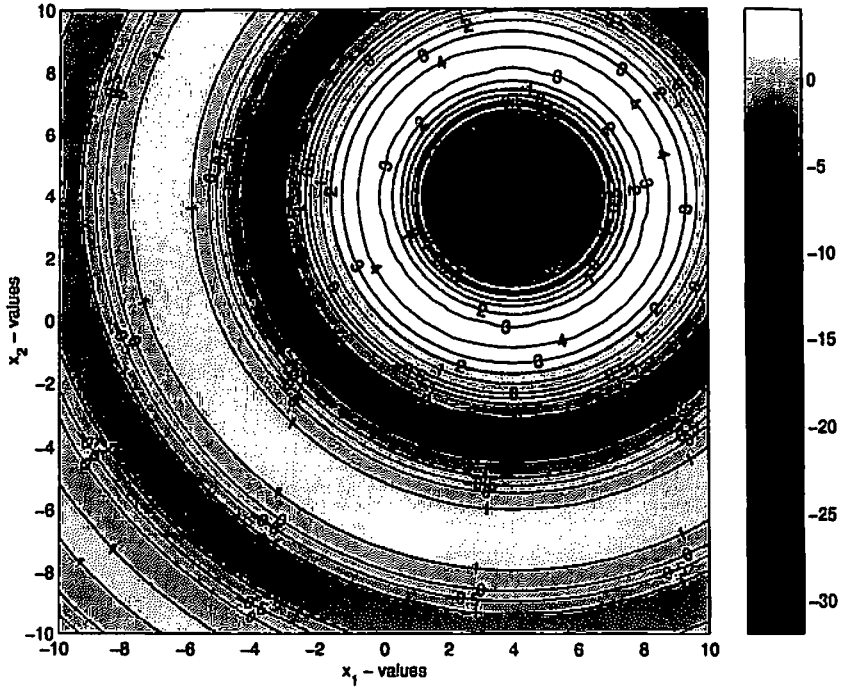

Figure 9.8 Simulated annealing: Increasing β—decreased opportunity to escape.

- Restarting from different design must still be implemented to overcome strong local influences.
- Restarting from the same initial design is also important for establishing convergence properties as this is a stochastic search process.

Constrained Problems: This chapter does not deal with constraints or discrete problems. There are two ways of handling constraints directly

1. If the new design point is not feasible, it is not accepted and no design change takes place. This is quite restrictive for random search directions.
2. The function that drives the algorithm during each iteration is the one with the largest violation. If the current point is feasible, then the objective function will drive the search. This make sense as the process is largely stochastic and is expected to wander around the design space.

Constrained problems can be expected to take a significantly larger number of searches. There is currently no procedure to identify convergence property of the SA with respect to constrained optimization.

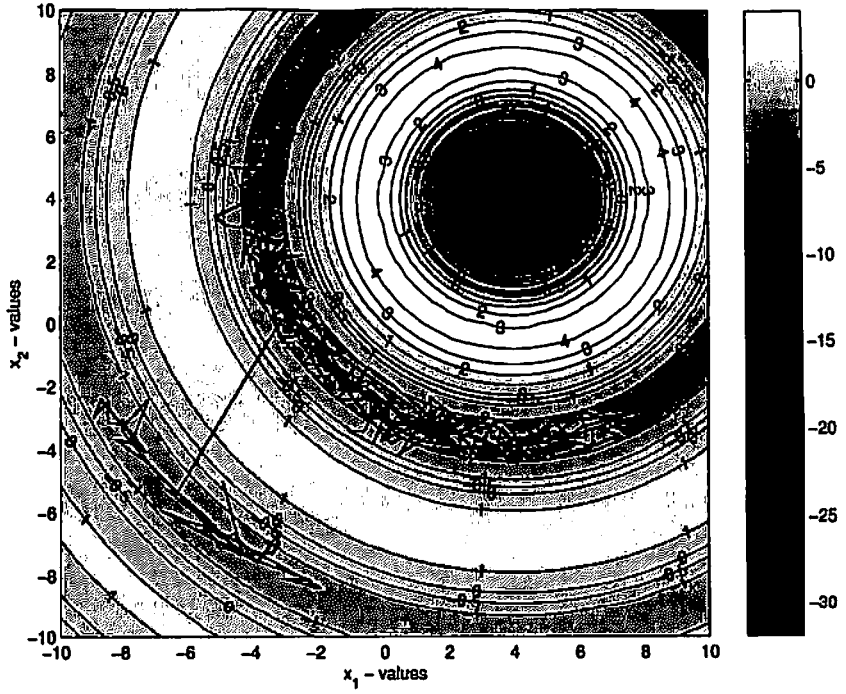

Figure 9.9 Simulated annealing: non-convergence—insufficient iterations.

Discrete Problems: In discrete problems, the SA algorithm will use a discrete design vector determined stochastically from the available set. Each element of the design vector is randomly chosen from the permissible set of values. The remaining elements of the SA algorithm are directly applicable. If the objective decreases, then the design change is accepted. If it increases, then a conditional probability of acceptance is determined allowing the solution to escape the minimum. This is left to the reader. The significant effort is the identification of the random design vector.

Example 9.2 This is a function of two variables that MATLAB uses for illustrating the graphics capability of the software. Typing *peaks* at the Command prompt will draw the surface plot. Typing *type peaks.m* at the Command prompt will provide the equations that were used to obtain the curve. This function used as the objective is available in **Example9_2.m** and **Example9_2plot.m**. SA is used to identify the global minimum. Figure 9.10 illustrates the contour plot along with the changes in the design. The function has two local minimums and three local maximums. The following represents a converged run.

Starting design = [−2, 2]; β = −2.5; Number of iterations = 1000

Useful number of iterations (when design changes occurred) = 338

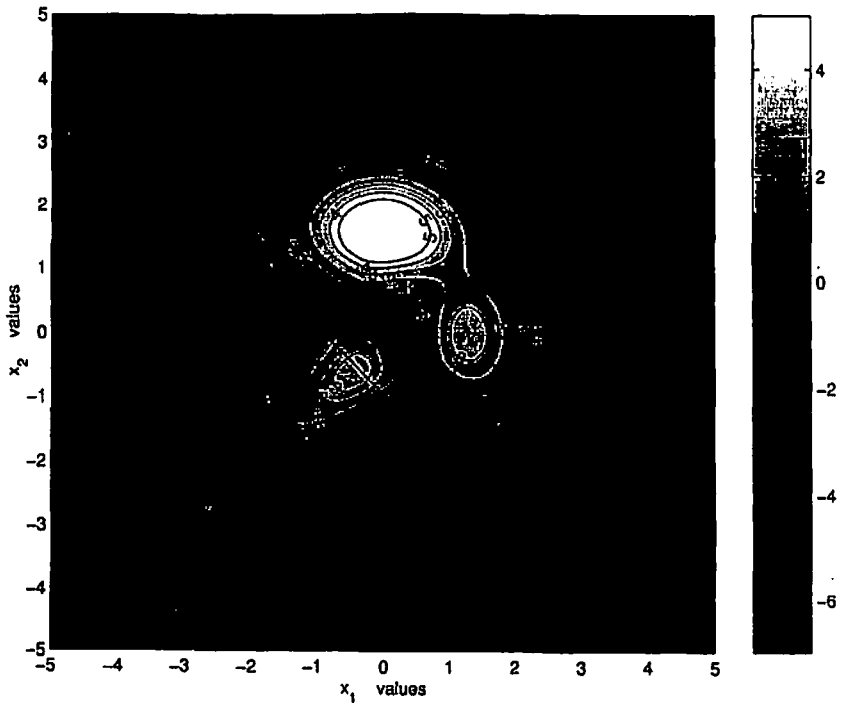

Figure 9.10 Simulated annealing: Example 9.2.

Final design = [0.02279, −1.6259]; Final Objective = −6.5511

This is the global minimum of the problem.

There was a significant change made to the SA algorithm based on initial tests with the execution. At the starting design, the design was largely positive (8.95). Changing it to a negative number did not permit the solution to escape the local minimum or to swing around the nearby local maximum. Numerical experiments suggest that the constant β be maintained between

$$-2.5 \le \beta \le -1 \tag{9.6}$$

Otherwise no change was made to the algorithm.

9.2.2 Genetic Algorithm (GA)

The Genetic Algorithm is an important part of a new area of applied research termed *Evolutionary Computation*. These are actually search processes and naturally useful

for discovering optimum solutions. Holland [7] was the first to use the technique, but its use as an optimization tool began in earnest in the late 1980s, developed momentum in the mid-1990s, and continues to attract serious interest today. The term *evolutionary* is suggestive of the natural process associated with biological evolution—primarily the Darwinian rule of the selection of the fittest. This takes place because the constant mutation and recombination of the chromosomes in the population yield a better gene structure. All of these terms appear in the discussions related to evolutionary computation. This basic process is also folded into the various numerical techniques. The discussions can become clear if the term *chromosomes* can be associated with *design variables*. Over the years, Evolutionary Computation has included *Genetic Programming*—which develops programs that mimick the evolutionary process; *Genetic Algorithms*—optimization of combinatorial/discrete problems; *Evolutionary Programming*—optimizing continuous functions without recombination; *Evolutionary Strategies*—optimizing continuous functions with recombination [8]. This classification is quite loose. For example, GAs have been used, and continue to be used effectively to solve continuous optimization problems. Evolutionary algorithms have been successfully applied to optimization problems in several areas: engineering design, parameter fitting, knapsack problems, transportation problems, image processing, traveling salesman, scheduling, and so on. As can be inferred from the list, a significant number are from the area of discrete programming.

This section and this book include only GAs, though all of the evolutionary algorithms share many common features. GAs generate and use stochastic information in their implementation and therefore can be considered global optimization techniques. They are exceptionally useful for handling ill behaved, discontinuous, and nondifferentiable problems. Unlike the SA, convergence to the global optimum in GAs can only be weakly established. GAs generate and use *population* of solutions—*design vectors* (**X**)—unlike the SA where a search direction (**S**) was generated. This distinction is essential to understand the difficulty of handling continuous problems in GAs. For example, using random numbers to generate a value for x_1, the values of 3.14, 3.141, or 3.1415 will all be distinct and have no relation to each other. The fact that they identify the same solution with different degrees of precision is ignored in GA. In SA, with the algorithm demonstrated in the previous section, getting close to the minimum is possible because of the stepsize computation. In GA, typically there may be no minor adjustments for neighboring solution. Such a predicament does not arise in the consideration of discrete problems. Hence, GA is more widely used for discrete problems. A basic GA is assembled below [9].

Generic Genetic Algorithm (GGA) (A9.3)

Step 1: Set up an initial *population* $P(0)$—an initial set of solution or chromosomes

Evaluate the initial solution for fitness—differentiate, collate, and rate solutions

Generation index $t = 0$

Step 2: Use *genetic operators* to generate the set of *children* (crossover, mutation)

Add a new set of randomly generated population (*immigrants*)

Reevaluate the population—*fitness*

Perform *competitive* selection—which members will be part of next generation

Select population $P(t+1)$—same number of members

If not converged $t \leftarrow t+1$

Go To Step 2

The various terms are explained below.

Chromosomes: While it is appropriate to regard each chromosome as a design vector, there is an issue of representation of the design vector for handling by genetic operators. It is possible to work with the design vector directly (as in the included example) or use some kind of mapping, real (*real encoding*) or binary (*binary encoding*). Earlier work in GA used binary encoding. In this book the design vector is used directly for all transactions. Real encoding is recommended. A piece of the chromosome is called an *allele*.

Fitness and Selection: The fitness function can be related to the objective function. The population can be ranked/sorted according to the objective function value. A selection from this population is necessary for (1) identifying parents and (2) identifying if they will be promoted to the next generation. There are various selection schemes. A fraction of the best individuals can be chosen for reproduction and promotion, the remaining population being made up of new immigrants. This is called *tournament* selection. A modification of this procedure is to assign a probability of selection for the ordered set of the population for both reproduction and promotion. This probability can be based on *roulette wheel*, *elitist selection*, *linear* or *geometric ranking* [10]. The implementation in this book uses ranking without assigning a probability of selection. It uses the objective function value to rank solutions.

Initial Population: The parameter here is the number of initial design vectors (chromosomes) for the starting generation, N_p. This number is usually kept the same in successive generations. Usually a normal random generation of the design vectors is recommended. In some instances, *doping*, adding some good solutions to the population, is performed. It is difficult to establish how many members must be considered to represent the population in every generation.

Evaluation: The various design vectors have to be evaluated to see if they can be used for creating children that can become part of the next generation. Fitness attributes can serve well for this purpose. For the case of unconstrained optimum, the objective function $f(\mathbf{X})$ is a good measure. Since the global minimum is unknown, a progressive measure may be used.

Genetic Operators: These provide ways of defining new populations from existing ones. This is also where the evolution takes place. Two types of operation are recognized. The first is *crossover* or *recombination*, and the second *mutation*.

Crossover Operators: There are several types of crossover strategies. A *simple crossover* is illustrated in Figure 9.11. Here, a piece of the chromosome—an allele—is exchanged between two parents P_1, P_2 to produce children C_1, C_2. The length of the piece is usually determined randomly. Mathematically, if the parents are $\mathbf{X} = [x_1, x_2, \ldots, x_n]$ and $\mathbf{Y}$ (defined similarly), and r is the randomly selected truncation index, then the children $\mathbf{U}$ (= C_1) and $\mathbf{V}$ (= C_2), by simple crossover are defined as [11]

$$u_i = \begin{cases} x_i, & \text{if } i < r \\ y_i, & \text{otherwise} \end{cases}$$

$$v_i = \begin{cases} y_i, & \text{if } i < r \\ x_i, & \text{otherwise} \end{cases} \tag{9.7}$$

Arithmetic crossover is obtained by a linear combination of two parent chromosomes to yield two children [9]. If λ_1 and λ_2 are randomly generated numbers, then the children can be defined as

$$C_1 = \lambda_1 \mathbf{X} + \lambda_2 \mathbf{Y}$$
$$C_2 = \lambda_2 \mathbf{X} + \lambda_1 \mathbf{Y} \tag{9.8}$$

If

$\lambda_1 + \lambda_2 = 1$; $\lambda_1, \lambda_2 > 0$—it is a convex combination

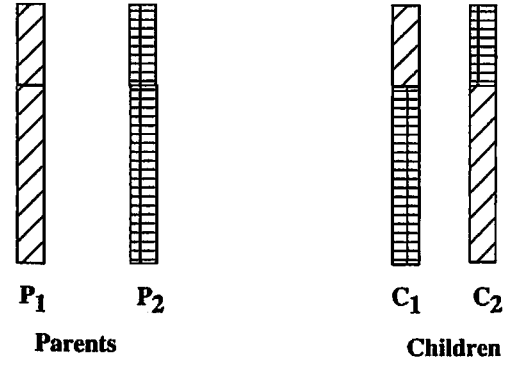

Figure 9.11 Genetic Algorithm: Simple crossover.

If there are no restrictions on λ's, it is an *affine combination*. For real λ's it is a *linear combination*. An important crossover operator is a *direction-based* crossover operator, if $f(\mathbf{Y})$ is less than $f(\mathbf{X})$ and r is a unit random number:

$$C = r^*(\mathbf{Y} - \mathbf{X}) + \mathbf{Y}$$

Mutation: Mutation refers to the replacement of a single element of a design vector by a randomly generated value. The element is also randomly chosen. For the design vector **X**, for a randomly selected element k

$$C = \begin{cases} x_i, & i \neq k \\ x_k, & i = k \end{cases}$$

There are other types of mutation including nonuniform mutation and directional mutation. The reader is referred to Reference 9 for a relevant description.

Immigrants: This is the addition of a set of randomly generated population in each generation before the selection is made for the next generation. The search process in GA is broadly classified as *exploration* and *exploitation*. Exploration implies searching across the entire design space. Exploitation is focusing more in the area promising a solution. Typically, crossover and mutation are ways to narrow the search to an area of design space based on the characteristics of the current population—suggesting a local minimum. To keep the search open for the global minimum, other areas must be presented for exploration. This is achieved by bringing in an unbiased population and having them compete for promotion to the next generation along with the current set. In many variations these are allowed to breed with the current good solutions before evaluation.

The ideas presented here are basic to all of evolutionary computation and specific to GA. They are also simple but complete. For each category listed above there are more sophisticated models that are being researched and developed and the reader is encouraged to seek information from the current literature. The specific GA algorithm implemented in the book incorporates all of the elements above in a simple implementation.

Genetic Algorithm (A9.4) The algorithm actually implemented is given below. In line with many other algorithms available in this text, it is provided as a way to understand and appreciate the working of the GA. It is also meant to provide a starting point for the user to implement his own variation of the GA since these techniques are sensitive to the specific problem being studied. Another important feature missing from this implementation is an encoding scheme. The design vector is handled directly to provide a direct feedback.

Step 1. Set up an initial population. Ten sets of design vectors are randomly created. From this set, an initial generation of two vectors is selected. This is the pool available for propagation. This is also the minimum to perform crossovers. The fitness function is the objective function.

Generation index $t = 1$

Maximum generation index = t_{max}

Step 2. Genetic Operations

Two Simple crossovers

Two Arithmetic crossovers (convex)

One Directional crossover

Mutation of all seven design vectors so far

Four immigrants

Selection of the best two design vectors (from 18 vectors) for promotion to the next generation

If $t = t_{max}$, Stop

increment generation index: $t \leftarrow t + 1$

Go to Step 2

Example 9.3 The example for this section is a Bezier curve fitting problem. Given a set of xy data, find the *fifth*-order Bezier curve that will fit the data. The objective function is the least squared error between the original data and the data generated by the Bezier curve. This problem has been addressed earlier through a continuous deterministic technique in an earlier chapter. There are *eight* design variables associated with the problem. There are several reasons for the choice of the problem. First, the global solution, if it exists, must be close to zero, and second, this provides a way to describe eight design variables in two-dimensional space (plot). The best value for the design vector in every generation is plotted to see how the curve fit improves. Since encoding is not being used in the algorithm, a reasonably long design vector is necessary to observe the crossovers. The original data are obtained from the following function:

$$f(x) = 1 + 0.25x + 2e^{-x}\cos 3x \tag{9.9}$$

Figure 9.12 illustrates the setup of the example including the eight design variables, which are the x and y components of the internal vertices that determine the fitted curve completely. The function is hard coded in **CallGA.m**. There are several reasons why this is not the best example for illustration of GA. First, it is an example in continuous variables. Second, local minimums are not very distinguishable and there are innumerable many of them. The global minimum is not substantially different from the local minimum value to warrant exploration of the global solution. Third, the problem is not ill behaved. The most important reason depends not on the algorithm but on the computation of the objective function—calculating the least squared error.

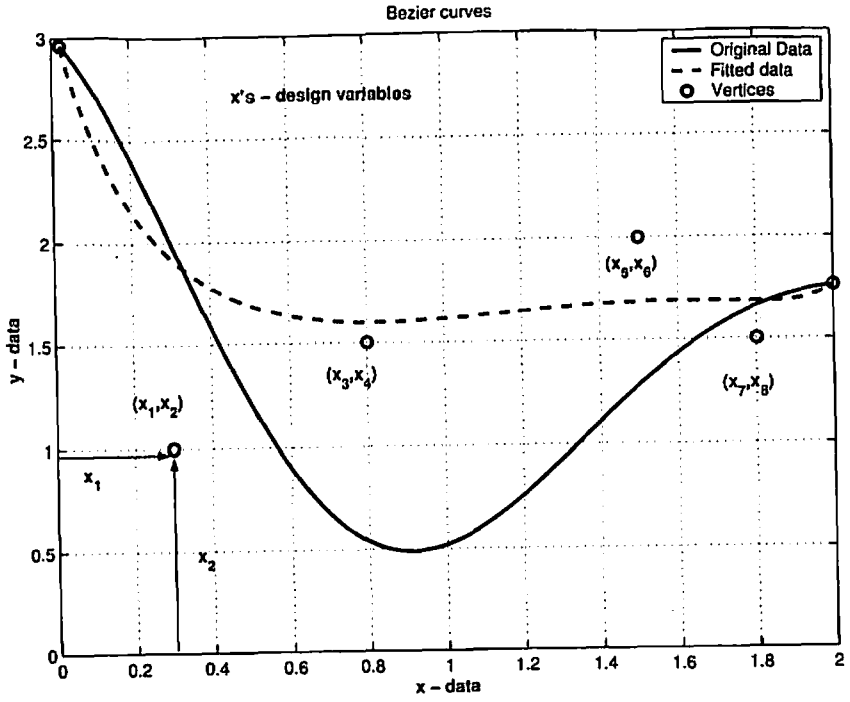

Figure 9.12 Genetic Algorithm, description of Example 9.3.

It uses a Newton-Raphson technique that is prone to failure if the curve folds over severely, or the vertices are very close together. One way to overcome this is to cause the x-location of the vertices to monotonically increase (which is implemented in the population generation function). This upsets many of the crossover operations since they are now conditionally random instead of purely random.

The following files are necessary for this specific example relating to the handling of Bezier curves: **coeff.m, Factorial.m, Combination.m, Bez_Sq_Error.m** and are available in the code directory for the chapter.

The GA in Algorithm (A9.4) is implemented using the following m-files.

CallGA.m	Setting up parameters and data for the algorithm
GeneticAlgorithm.m	Directs the algorithm
populator.m	Generates random design vector
SimpleCrossover.m	Performs simple crossover between two parents
ArithmeticCrossover.m	Performs convex crossover between two parents
DirectionalCrossover.m	Performs one directional crossover
Mutation.m	Performs mutation of the design vectors
DrawCurve.m	Draws the curve for each design vector

Figures 9.13 to 9.17 capture the application of GA to Example 9.3 through 200 generations. The reader should be able to run the example from the code directory of Chapter 9. Because of the heuristic nature of the generation of solutions, the solution trail will be different from that shown below. Running the example several times by invoking CallGA from the Command window should illustrate that the GA always manages to closely approximate the curve after 200 generations. The maximum generation value is not predictable and cannot be established with certainty. It relies on numerical experimentation. This number appears easily to be over 1000 in many investigations. The reader is urged to experiment with the example to explore this issue. In running **CallGA.m** the best values of the design vector and the corresponding objective function value are captured in the variable Xbest—an array of 201 rows and 9 columns (assuming 200 generations). The first 8 columns carry the best design for the generation and the last column is the corresponding objective function value.

Figure 9.13 is the best design from the starting generation. It has an objective function value of 32.13. This is the error squared over 100 data points of the original data and the curve generated by the design variables. The design variables are the vertices illustrated in the figure. The basic curve of Figure 9.13 is obtained using the

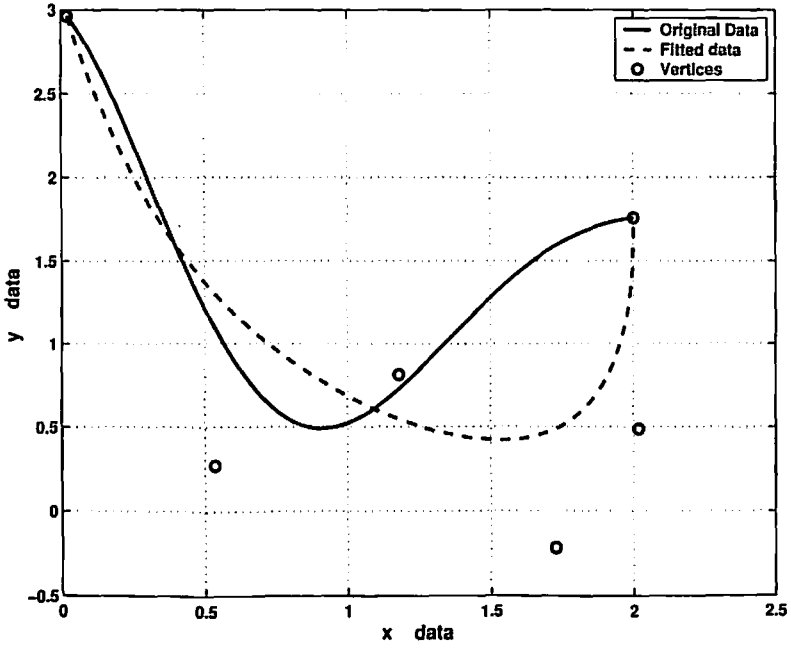

Figure 9.13 Best fit at generation $t = 1$.

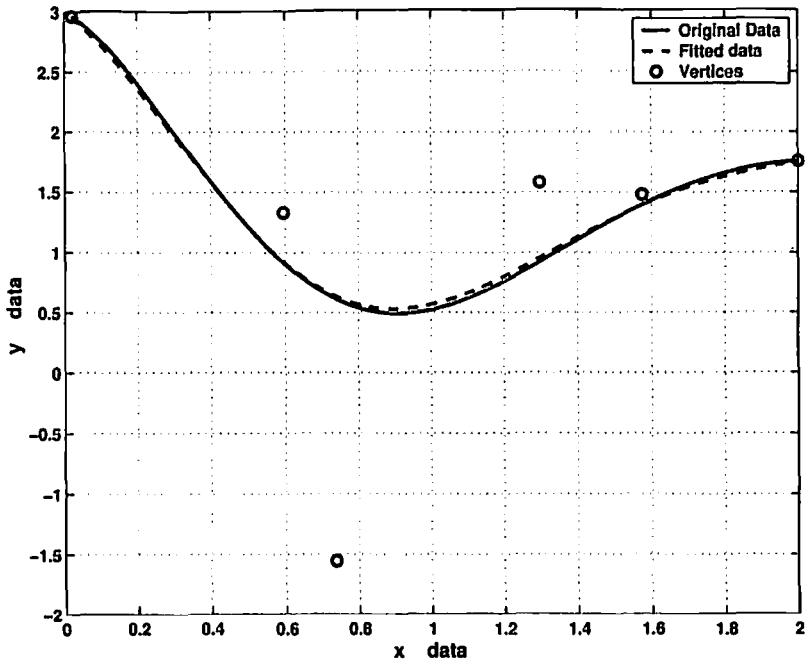

Figure 9.14 Best fit: generation $t = 200$.

DrawCurve.m function. To draw the initial curve (the figure is edited subsequently with *plotedit*) in the Command window:

```
>> DrawCurve(Xbest(1,1:8))
```

Figure 9.14 is the best design at the final generation (generation number = 200). The objective function is 0.0729. It is clear from the figure that the GA is attempting to find the solution. This can be established by looking at Figure 9.15 which is the iteration/generation history of the GA for this example. The objective function is generally decreasing through the generations. Figure 9.16 is a scaled representation to display the behavior of the GA at later iterations. From the data it is also apparent that this example needs to run for more than 200 generations. The final figure, Figure 9.17 represents the movement and the clustering of the vertices through the generations. It is suggestive of local refining that is associated with the GA.

Suggested Extensions: Most of the engineering design problems involve constraints. Currently the most popular way to handle constraints is to reformulate the problem using penalty functions (Chapter 7). There are several other alternatives to

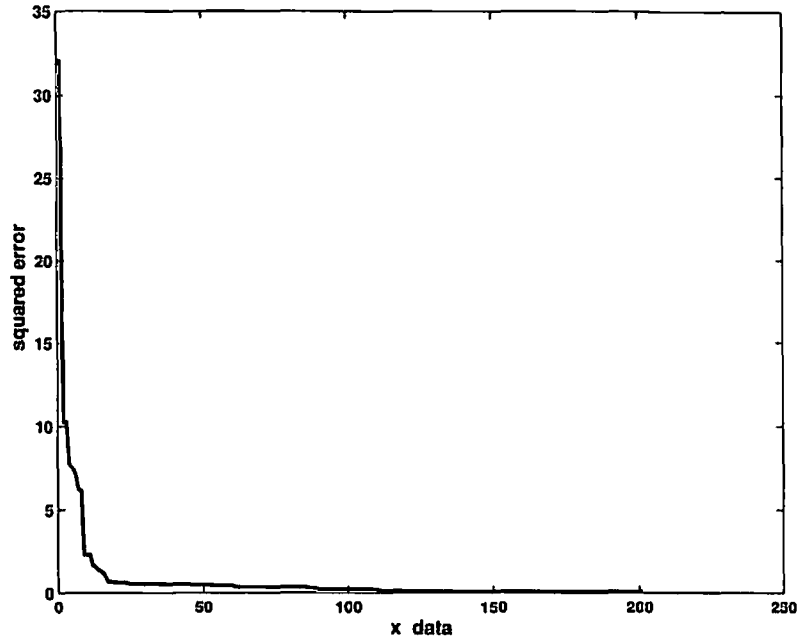

Figure 9.15 Objective function for every generation.

this procedure. It is possible to separately accumulate only designs that are feasible while continuing to implement the standard procedure. Second, use a strategy to drive the search process through constraint satisfaction and decreasing the objective, simultaneously or separately. The algorithms indicated in this chapter can easily incorporate the needed changes.

The algorithmic procedure as implemented does not really care if the problem is continuous or discrete. It just works on a current set of values for the design variables. Discrete values will primarily require minor changes in the way the population is generated. Also the implementation of the crossover and mutation operators will need to be changed. These changes in no way detract from the major idea behind the GA.

Closing Comments: In summary, for a heuristic technique, the GA is very impressive. The same can be said for the SA. Including the fact that practical engineering problems must require partly discrete representation, the techniques in Chapters 8 and 9 are very relevant to the practice of design and development. With the enormous computational resource in the desktops and laptops, an effective programming language like MATLAB, and a wide variety of algorithms that are available, it is possible now to design an individual approach to many optimization

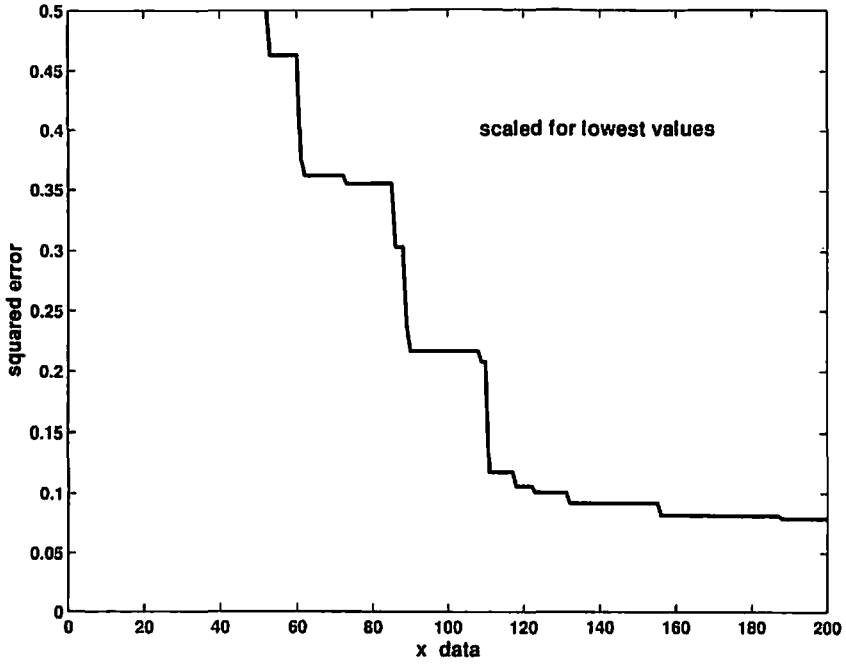

Figure 9.16 Objective function scaled to show continual decrease for later generation.

problems. This is important to practicing engineers who can bring intuition and insight for the particular problem they have to solve in a tight time frame. In the engineering curriculum, traditional design optimization courses tend to deal only with deterministic continuous algorithms. That in itself was more than the time available for instruction. Discrete optimization or global optimization are necessarily skills learned by personal endeavor. The overemphasis on deterministic programming needs to be revisited in the light of current developments in the area of optimization. Today, computer programming in engineering departments and its reinforcement is usually adjunct to other developments in the curriculum. New programming platforms like MATLAB offer a way to resolve this disadvantage. New algorithms also reflect a simplicity that can be easily translated and applied.

This book has been about programming in MATLAB, learning about traditional and current techniques in design optimization, and translating these optimization techniques into applications. It is also about exploration and developing an insight into the process of design optimization. The code that is provided is only meant to be a starting point.

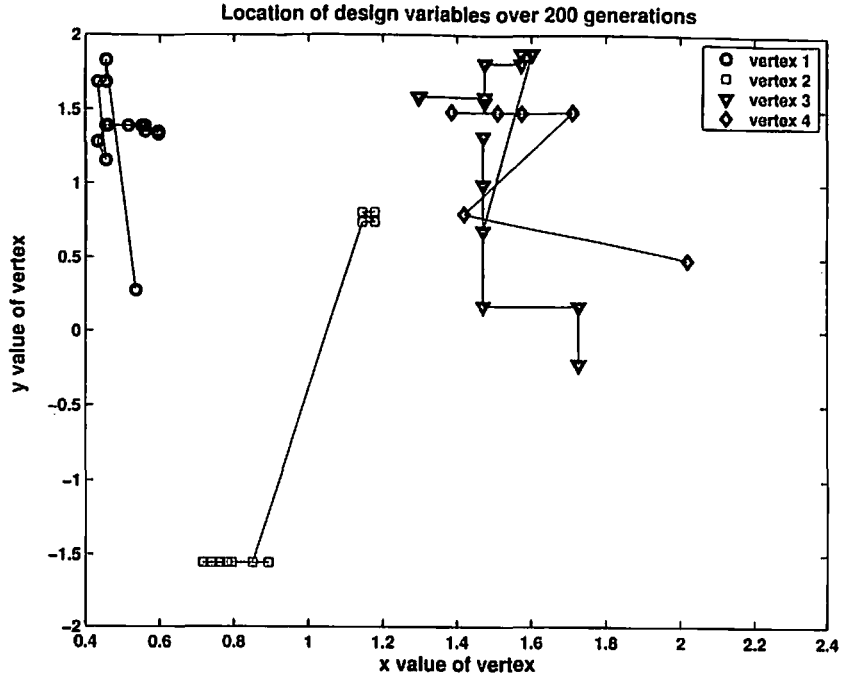

Figure 9.17 Trace of the design variables through generation.

REFERENCES

1. Arora, J. S., *Introduction to Optimal Design*, McGraw–Hill, New York, 1989.
2. Kirkpatrick, S., Gelatt, G. C., and Vecchi, M. P., Optimization by Simulated Annealing, *Science*, Vol. 220, No. 4598, May 1983.
3. Metropolis, N., Rosenbluth, A., Rosenbluth, M., Telle, A., and Teller, E., Equations of State Calculations by Fast Computing Machines, *Journal of Chemical Physics*, Vol. 21, pp. 1087–1092, 1953.
4. Bochavesky, I. O., Johnson, M. E., and Myron, L. S., Generalized Simulated Annealing for Function Optimization, *Technometrics*, Vol. 28, No. 3, 1986.
5. Jones, A. E. W., and Forbes, G. W., An Adaptive Simulated Annealing Algorithm for Global Optimization over Continuous Variables, *Journal of Global Optimization*, Vol. 6, pp. 1–37, 1995.
6. Cohn, H., and Fielding, M., Simulated Annealing: Searching for an Optimal Temperature Schedule, *SIAM Journal on Optimization*, Vol. 9, No. 3, pp. 779–782, 1999.
7. Holland, J. H., Outline of a Logical Theory of Adaptive Systems, *Journal of ACM*, Vol. 3, pp. 297–314, 1962.

8. Dumitrescu, D., Lazzerini, B., Jain, L. C., and Dumitrescu, A., *Evolutionary Computation*, International Series on Computational Intelligence, L. C. Jain, (ed.), CRC Press, Boca Raton, FL, 2000.
9. Gen, M., and Cheng, R., *Genetic Algorithms and Engineering Optimization*, Wiley Series in Engineering Design and Automation, H. R., Parsaei (series ed.), Wiley–Interscience, New York, 2000.
10. Goldberg, D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison–Wesley, Reading MA, 1989.
11. Houck, C. R., Joines, J. A., and Kay, M. G., A Genetic Algorithm for Function Optimization, paper accompanying the GAOT (Genetic Algorithm Optimization Toolbox), North Carolina State University.

PROBLEMS

No formal problems are suggested. The reader is encouraged to develop his own variants of the two algorithms, including formal ones available in the suggested literature. He is also encouraged to consider extensions to constrained and discrete problems. He is recommended to use his versions to explore various examples found in the literature. In comparison with the deterministic techniques (until Chapter 8), these heuristic methods are truly fascinating and illustrate considerable robustness, and what is even more challenging to our perception, they can solve any problem with no reservations whatsoever. They are incredibly basic in construction, simple to program, and easy to deploy. They only require a lot of time to search the design space.

10

OPTIMIZATION TOOLBOX FROM MATLAB

The MATLAB family of products includes the Optimization Toolbox, which is a library of m-files that can be used to solve problems in the area of optimization. It contains many of the methods that have been explored in this book, and several that are not available in the Toolbox. For those considering the practice of design optimization seriously, the Optimization Toolbox is a worthy investment. Many of the techniques and discussions in the book were kept simple to illustrate the concepts, and the m-files were used to provide MATLAB programming experience. Most of the m-files available on the web site will need to be developed further for serious practical application. Another important reason for having access to the Optimization Toolbox is that it allows one to take advantage of MATLAB's *open source* policy, that is, to have access to the source files. You may be able to modify and extend those files to meet your special optimization requirements. While some of the programs from the Toolbox have been used or referenced elsewhere in this book, this chapter collects them in a single place for easy reference. As can be expected, the material of this chapter will be dated. The chapter is based on MATLAB Version 5.3. As of this writing, Version 6.0/6.1 was shipping. In general, the MATLAB products, especially the Optimization Toolbox, have reached a level of maturity such that newer versions should not invalidate the information in this section. What should be expected, however, is that the material in this chapter will most likely be upgraded, enhanced, and extended with each new release of MATLAB.

10.1 THE OPTIMIZATION TOOLBOX

Most of the basic information in this chapter is from the user's guide for the Optimization Toolbox [1]. If you are buying a specially bundled CD that includes the Optimization Toolbox, then you may not have Reference 1 in book form. You may have Reference 1 as a PDF file on the CD or you may have access to it. For example, on the Windows NT platform for default directory installation the PDF version of Reference 1 is available at

C:\MATLABR11\help\pdf_doc\optim\optim_tb.pdf

Reference 1 provides examples of use, discusses special cases, algorithms used, and many other significant topics. This chapter does not attempt to document them. Limited duplication is used to bring attention to the most useful features. The emphasis is on illustrating the use of the Toolbox for various examples introduced in the book. In fact, a large number of the techniques in the Toolbox are not used in the book and are not covered in this section. The experience provided in this chapter should be sufficient to understand and use the new ones effectively.

To use the programs or m-files in the Toolbox, three important actions must be initiated. One, formulate the problem in the format expected by MATLAB with respect to the particular function that will be called. Mostly, this will involve using appropriate procedures to set the right-hand side of the constraints to zero or constant values. The Toolbox also makes a point of distinguishing between linear and nonlinear constraints. Second, set/change the parameters for the optimization algorithms. Third, use the appropriate optimizer and ensure that a valid solution has been obtained. All of these actions naturally depend on the actual mathematical model of the problem.

10.1.1 Programs

The different types of optimization problems that can be solved with the version of the Toolbox installed on the computer can be discovered in the Command window, Help window, or in the browser using Help Desk. In the MATLAB Command window type

```
>> help   optim
```

The output is reproduced below. They are all of the various m-files in the Toolbox and their optimization function is also identified.

```
Optimization Toolbox.
   Version 2.0 (R11) 09-Oct-1998

What's new.
  Readme - New features, bug fixes, and changes in this
version.
```



```
Nonlinear minimization of functions.
  fminbnd   - Scalar bounded nonlinear function
minimization.
  fmincon   - Multidimensional constrained nonlinear
minimization.
  fminsearch - Multidimensional unconstrained nonlinear
minimization, by Nelder-Mead direct search method.
  fminunc   - Multidimensional unconstrained nonlinear
minimization.
  fseminf   - Multidimensional constrained minimization,
semi-infinite constraints.

Nonlinear minimization of multi-objective functions.
  fgoalattain   - Multidimensional goal attainment
optimization.
  fminimax - Multidimensional minimax optimization.

Linear least squares (of matrix problems).
  lsqlin - Linear least squares with linear constraints.
  lsqnonneg - Linear least squares with nonnegativity
constraints.

Nonlinear least squares (of functions).
  lsqcurvefit - Nonlinear curvefitting via least squares
(with bounds).
  lsqnonlin - Nonlinear least squares with upper and lower
bounds.

Nonlinear zero finding (equation solving).
  fzero - Scalar nonlinear zero finding.
  fsolve - Nonlinear system of equations solve function
solve.

Minimization of matrix problems.
  linprog - Linear programming.
  quadprog - Quadratic programming.

Controlling defaults and options.
  optimset   - Create or alter optimization OPTIONS
structure.
  optimget - Get optimization parameters from OPTIONS
structure.

Demonstrations of large-scale methods.
  circustent - Quadratic programming to find shape of a
circus tent.
  molecule   - Molecule conformation solution using
unconstrained nonlinear minimization.
```

```
  optdeblur - Image deblurring using bounded linear
least-squares.

  Demonstrations of medium-scale methods.
  optdemo - Demonstration menu.
  tutdemo - Tutorial walk-through.
  bandemo - Minimization of banana function.
  goaldemo - Goal attainment.
  dfildemo - Finite-precision filter design (requires
Signal Processing Toolbox).
  datdemo - Fitting data to a curve.
```

The `tutdemo` is recommended for a demonstration of the features of the Toolbox. Of the above items, `optimset`, `linprog`, `quadprog`, `fminunc`, and `fmincon` are further explored in this chapter.

A clearer picture of the capability of the functions can be obtained from the tabular graphical representation of the problem sets in the first chapter of Reference 1. For example, the four techniques referenced in this chapter are illustrated in Table 10.1.

10.1.2 Using Programs

Each of the functions in the Toolbox that implements an optimization technique can be used in several ways. These are documented in Reference 1 and are also available online through one of the mechanisms of obtaining help in MATLAB. For example, typing **help fmincon** in the Command window identifies the following different ways of using the function/program (the following is edited to show only types of usage).

```
  FMINCON Finds the constrained minimum of a function of
several variables.
  FMINCON solves problems of the form:
     min F(X) subject to:
  A*X <= B, Aeq*X = Beq (linear constraints)
```

Table 10.1 Graphical Description of Some Toolbox Functions

Type	Notation	function
Unconstrained minimization	$\min_x f(x)$	`fminunc`, `fminsearch`
	$\min_x f^T x$ such that	
Linear programming	$[A][x] \le [b]$; $[A_{eq}][x] = [b_{eq}]$; $l \le x \le u$	`linprog`
	$\min_x \frac{1}{2}[x]^T [\mathbf{H}][x] + f^T x$ such that	
Quadratic programming	$[A][x] \le [b]$; $[A_{eq}][x] = [b_{eq}]$; $l \le x \le u$ $\min_x f(x)$; such that $c(x) \le 0$; $c_{eq}(x) = 0$	`quadprog`
Constrained minimization	$[A][x] \le [b]$; $[A_{eq}][x] = [b_{eq}]$; $l \le x \le u$	`fmincon`


```
  C(X) <= 0, Ceq(X) = 0 (nonlinear constraints)
  LB <= X <= UB

  X=FMINCON(FUN,X0,A,B)
  X=FMINCON(FUN,X0,A,B,Aeq,Beq)
  X=FMINCON(FUN,X0,A,B,Aeq,Beq,LB,UB
  X=FMINCON(FUN,X0,A,B,Aeq,Beq,LB,UB,NONLCON)
  X=FMINCON(FUN,X0,A,B,Aeq,Beq,LB,UB,NONLCON,OPTIONS)
  X=FMINCON(FUN,X0,A,B,Aeq,Beq,LB,UB,NONLCON,OPTIONS,...
   P1,P2,...)
  [X,FVAL]=FMINCON(FUN,X0,...)
  [X,FVAL,EXITFLAG]=FMINCON(FUN,X0,...)
  [X,FVAL,EXITFLAG,OUTPUT]=FMINCON(FUN,X0,...)
  [X,FVAL,EXITFLAG,OUTPUT,LAMBDA]=FMINCON(FUN,X0,...)
  [X,FVAL,EXITFLAG,OUTPUT,LAMBDA,GRAD]=FMINCON(FUN,X0,...)
  [X,FVAL,EXITFLAG,OUTPUT,LAMBDA,GRAD,HESSIAN]=FMINCON
   FUN,X0,...).
```

There are 12 ways of using the *fmincon* function. The differences depend on the information in the mathematical model that is being used and the postoptimization information that should be displayed. In using the different function calls, any missing data that do not exist for the particular examples must be indicated using a null vector ([]).

There are three *keywords* in the above list of function calls that need special attention in constrained nonlinear optimization.

EXITFLAG: For fmincon this indicates if the optimization was successful.

```
If EXITFLAG is:
   >0 then FMINCON converged to a solution X.
   0 then the maximum number of function evaluations was
reached.
   < 0 then FMINCON did not converge to a solution.
```

OUTPUT: This is special MATLAB structure element. *Structure* is a programming construct used in many higher-level languages. It is similar to *array* in MATLAB. An array will contain only similar elements—like all numbers or all strings. A structure, on the other hand, can contain disparate elements, such as several numbers and strings. The elements of the structure can be accessed by using the name of the structure element followed by a period and the predefined keyword for the particular element. For example, two of the elements of the OUTPUT are:

`OUTPUT.iterations`: the number of function evaluations
`OUTPUT.algorithm`: the name of the algorithm used to solve the problem

LAMBDA: This is also another structure in MATLAB and contains the Lagrange multipliers. The values of theses multipliers are useful for verifying the satisfaction of the necessary conditions of optimization. The values can be recovered by the same procedure indicated for the OUTPUT structure.

`LAMBDA.lower`: value of the multipliers at the lower bound
`LAMBDA.ineqnonlin`: value of the multipliers tied to the inequality constraints (β's in the book)

These items were associated with the function *fmincon*. Other functions may have different/additional types of information available.

10.1.3 Setting Optimization Parameters

There are several parameters that are used to control the application of a particular optimization technique. These parameters are available through the OPTIONS *structure*. All of them are provided with default values. There will be occasions when it is necessary to change the default values. One example may be that your constraint tolerances are required to be lower than the default. Or EXITFLAG recommends a larger number of function evaluations to solve the problem. These parameters are set using a *name/value* pair. Generally the name is usually a string while the value is typically a number. The list of parameters is available in Reference 1 and can be accessed most easily in the Command window by typing

```
>>    help  optimset
```

A partial list of the parameters in the OPTIONS structure, copied from the Command window, follows.

```
  OPTIMSET PARAMETERS
  Diagnostics - Print diagnostic information about the
function to be minimized or solved [ on | {off}]
  DiffMaxChange - Maximum change in variables for finite
difference gradients [ positive scalar | {1e-1} ]
  DiffMinChange - Minimum change in variables for finite
difference gradients [ positive scalar | {1e-8} ]
  Display - Level of display [ off | iter | {final} ]
  GradConstr - Gradients for the nonlinear constraints
defined by user [ on | {off} ]
  GradObj - Gradient(s) for the objective function(s)
defined by user [ on | {off}]
  Hessian - Hessian for the objective function defined by
user [ on | {off} ]
  HessUpdate - Quasi-Newton updating scheme [ {bfgs} | dfp
| gillmurray | steepdesc ]
  LineSearchType - Line search algorithm choice [
cubicpoly | {quadcubic} ]
  MaxFunEvals - Maximum number of function evaluations
allowed [ positive integer ]
  MaxIter - Maximum number of iterations allowed [
positive integer ]
```



```
  TolCon - Termination tolerance on the constraint
violation [ positive scalar ]
  TolFun - Termination tolerance on the function value [
positive scalar ]
  TolX - Termination tolerance on X [ positive scalar ]
```

The parameter names are identified in bold font. These parameters are the ones that will control a standard optimization application. The default values for the parameter are given within the curly braces. For example:

```
>> options = optimset('Diagnostics','on','Display',
'iter','MaxIter','500','HessUpdate','dfp','TolCon','1.
0e-08')
```

If the above statement is used prior to actually calling an optimization technique, the default values for those parameters will be changed in the OPTIONS structure. The default values for a particular technique can be found by typing the Command window

```
>>   options = optimset('fmincon')
```

There are several other ways to use the optimset program itself. For details consult Reference 1.

10.2 EXAMPLES

The examples illustrated in this section are continuous optimization problems discussed in the book. They are from linear programming, quadratic programming, unconstrained minimization, and constrained minimization. Only the appropriate mathematical model is introduced prior to illustrating the use of the Toolbox function.

10.2.1 Linear Programming

The LP example used here includes all three types of constraints. The toolbox function is *linprog*.

Example 10.1

$$
\begin{aligned}
\text{Maximize} \quad & z = 4x_1 + 6x_2 + 7x_3 + 8x_4 \\
\text{Subject to:} \quad & x_1 + x_2 + x_3 + x_4 = 950 \\
& x_4 \geq 400 \\
& 2x_1 + 3x_2 + 4x_3 + 7x_4 \leq 4600 \\
& 3x_1 + 4x_2 + 5x_3 + 6x_4 \leq 5000 \\
& x_1, x_2, x_3, x_4 \geq 0
\end{aligned}
$$

The *linprog* program solves the following problem:

$$\text{Minimize} \quad f^T x; \quad \text{subject to } [A][x] \le [b]$$

The objective function for the example needs to be changed to a minimization problem. The greater than or equal to constraint needs to be changed to a less than or equal to one. The *linprog* program allows equality constraint and explicit bounds on variables. The equality constraint can be set up with A_{eq} and b_{eq}. There are not many parameters that need to be changed except switching of the 'LargeScale' problem to 'off.' For only four design variables leaving it on should be permissible. The form of *linprog* that will be used is

```
[X,FVAL,EXITFLAG] = LINPROG(f,A,b,Aeq,beq,LB,UB)
```

This form is not explicitly documented in Reference 1 or in the help documentation. **Ex10_1.m**[1] is the m-file that is used to set up and run the problem. It is not displayed here but running the file in the Command window produces the following output.

```
Optimization terminated successfully.
X =
    0.0000
  400.0000
  150.0000
  400.0000
FVAL =
 -6.6500e+003
EXITFLAG =
    1
```

The solution to the problem is

$$x_1^* = 0; \quad x_2^* = 400; \quad x_3^* = 150; \quad x_4^* = 400; \quad z^* = 6650$$

10.2.2 Quadratic Programming

Quadratic programming refers to a problem with a quadratic objective function and linear constraints. While such a problem can be solved using nonlinear optimization techniques of Chapter 7, special quadratic programming algorithms based on the techniques of linear programming are normally used as they are simple and faster. If the objective function is a quadratic function in the variables, then it can be expressed in quadratic form. For example,

$$f(x_1, x_2) = x_1 + 2x_2 + x_1^2 + x_1 x_2 + 2x_2^2$$

can be expressed as

[1]Files to be downloaded from the web site are indicated by boldface sans serif type.

$$f(x_1, x_2) = [1 \quad 2]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \frac{1}{2}[x_1 \quad x_2]\begin{bmatrix} 2 & 1 \\ 1 & 4 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$f(x_1, x_2) = [c]^T\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \frac{1}{2}[x_1 \quad x_2]\,[\mathbf{H}]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$f(x_1, x_2) = [c]^T\,[x] + \frac{1}{2}[x]^T\,[\mathbf{H}]\,[x]$$

Example 10.2 Solve the quadratic programming problem

$$\begin{aligned} \text{Minimize} \quad & f(x_1, x_2, x_3) = [-1 \quad -2 \quad -3]^T\,[x] + \frac{1}{2}[x]^T\begin{bmatrix} 3 & -1 & 1 \\ -1 & 2 & 1 \\ 1 & 1 & 4 \end{bmatrix}[x] \\ \text{Subject to:} \quad & x_1 + 2x_2 - x_3 \le 2.5 \\ & 2x_1 - x_2 + 3x_3 = 1 \\ & x_1, x_2, x_3 \ge 0 \end{aligned}$$

The program to be used from the Toolbox is quadprog. The standard format for this program is

$$\begin{aligned} \text{Minimize} \quad & f(x) = c^T x + \frac{1}{2} x^T\,\mathbf{H}x \\ \text{Subject to:} \quad & A\,x \le b \\ & A_{eq}\,x = b_{eq} \\ & LB \le x \le UB \end{aligned}$$

The solution can be found in **Ex10_2.m**. The default optimization parameters are used so that the options structure does not have to be changed. The actual calling statement used is

```
[x,fval,EXITFLAG] = quadprog(H,c,A,b,Aeq,beq,LB,UB)
```

The solution is formatted and displayed in the Command window. It is copied from the Command window:

```
Optimization terminated successfully.
Final Values
Optimum Design Variables
------------------------
  0.5818   1.1182   0.3182
Optimum function value
----------------------
  -1.9218
```

```
Lagrange Multipliers for inequality constraint
-----------------------------------------------
  0.0218
Inequality constraint
----------------------
  2.5000
Lagrange Multipliers for equality constraint
-----------------------------------------------
  0.0164
Equality constraint
---------------------
  1.0000
```

10.2.3 Unconstrained Optimization

Example 10.3 is used to illustrate the use of fminunc, the program to be used for unconstrained minimization. It is a simple curve fitting example (a lot of unconstrained minimization problems in engineering are curve fitting used for developing correlation from empirical data). The Toolbox has programs specifically for curve fitting. It may be a useful idea for the reader to compare the solutions using those programs as this provides an opportunity to use programs from the Toolbox not illustrated in this chapter.

Example 10.3 A set of data over 100 points representing a cosine function is created. A quadratic polynomial is used to fit the data. An unconstrained minimization problem based on the sum of squares of the error between original data and the polynomial, over all the data points is solved for the coefficients of the polynomial. This is an example of the classical least squared error problem.

The example is solved in the m-file **Ex10_3.m**. Running the example also displays a comparison of the fit on a figure which is not included here. Two solutions are obtained in the example starting from the same initial guess. The first one is solved using *user-supplied* gradients. The second is the default *finite difference* gradients. The results are copied from the Command window:

```
Final Values - User specified Gradients
Optimum Design Variables
-------------------------
  -0.4309   -0.0365   1.0033
Optimum function value
----------------------
  2.4055e-004
Gradients of the function
-----------------------------
1.0e-007*
  -0.1324   -0.2198   -0.3275
```



```
Final Values - Default Finite Difference Gradients
Optimum Design Variables
-------------------------
  -0.4309   -0.0365   1.0033
Optimum function value
----------------------
2.4055e-004
Gradients of the function
-----------------------------
1.0e-004 *
  -0.0121   -0.0270   0.5715
```

It is generally believed that the actual values of gradients would lead to a faster and better solution. For fminunc to use the gradients computed by the user, changes have to be made to the options structure as well as the file where the objective function is calculated (**obj10_3.m**). In the options structure the 'GradObj' parameter should be set to 'on.' This is important otherwise MATLAB does not expect the gradients to be provided by the user. The option for this example is set by

```
options = optimset('LargeScale','off','Display',
'final', 'TolFun', '1.0e-08','GradObj','on');
```

This causes the program to look for the gradients in the second element of the return value. The function statement used for **obj10_3.m** is

```
function [f, grad] = obj10_3(x)
```

Both the objective function (f) and the gradients (grad) are calculated in the same function m-file.

10.2.4 Constrained Optimization

Example 10.4 The example is the same as Example 8.4 and represents the constrained example for this section. The mathematical model is

$$\text{Minimize} \quad f(\mathbf{X}) = \rho L A_c$$

$$\text{Subject to:} \quad g_1(\mathbf{X}): WL^3 - 12EI \le 0.25 \ \text{(deflection)}$$

$$g_2(\mathbf{X}): WLx_1 - 8I\sigma_y \le 0 \ \text{(normal stress)}$$

$$g_3(\mathbf{X}): WQ_c - 2Ix_3\tau_y \le 0 \ \text{(shear stress)}$$

Some geometric constraints to relate the various design variables are

$$g_4(\mathbf{X}): x_1 - 3x_2 \le 0$$

$$g_5(\mathbf{X}): 2x_2 - x_1 \le 0$$

$$g_6(\mathbf{X}): x_3 - 1.5x_4 \le 0$$

$$g_7(\mathbf{X}): 0.5x_4 - x_3 \le 0$$

The side constraints on the design variables are

$$3 \le x_1 \le 20, \quad 2 \le x_2 \le 15, \quad 0.125 \le x_3 \le 0.75, \quad 0.25 \le x_4 \le 1.25$$

The following relations define the cross-sectional area (A_c), moment of inertia (I), and first moment of area about the centroid (Q_c):

$$A_c = 2x_2x_4 + x_1x_3 - 2x_3x_4$$

$$I = \frac{x_2x_1^3}{12} - \frac{(x_2 - x_4)(x_1 - 2x_3)^3}{12}$$

$$Q_c = 0.5x_2x_4(x_1 - x_4) + 0.5x_3(x_1 - x_4)^2$$

The parameters for the problem are W = 1.4*2000, L = 96, E = 29 E+06, ρ = 0.284, σ_y = 36 E+03, τ_y = 21 E+03. There are three nonlinear inequality constraints and four linear inequality constraints. The code for running the program is available in **Ex10_4.m**.

The program to be used is fmincon. The following OPTION parameters are changed.

```
options = optimset('LargeScale','off','Display', ...
'iter', 'TolFun',1.0e-08);
```

The calling statement used for fmincon with the objective function available in objective.m, and the nonlinear constraints in nonlin_cons.m, is

```
[x, fval, exitflag, output, lambda] = ...
fmincon('objective',x0,A,B,[],[],LB,UB,'nonlin_cons',
options);
```

The linear inequality constraints are set up in the matrix A and vector B. There are no linear equality constraints and they are indicated by the null matrices. The lower and upper bounds are applied through the vectors LB and UB. The customized part of the output displayed in the Command window is included below. Accessing the various structure values will provide details about the performance of the algorithm.


```
Final Values
Optimum Design Variables
------------------------
  6.8671  2.2890  0.1802  0.2500
Optimum function value
----------------------
  62.4905
Final Nonlinear Constraints
---------------------------
1.0e+009 *
  -2.7304  -0.0025  -0.0001
Lagrange Multipliers for Nonlinear constraints
----------------------------------------------
  0  0  0
Lagrange Multipliers for Linear constraints
-------------------------------------------
  2.5879  0  0  0
```

The reader is encouraged to run the example with 'Diagnostics' on, explore the options structure, and the output structure.

REFERENCE

1. Coleman, T. F., Branch, M. A., and Grace, A., *Optimization Toolbox—for Use with MATLAB®*, User's Guide, Version 2, MathWorks Inc., 1999.

INDEX