1

INTRODUCTION

COPYRIGHTED MATERIAL

Optimization is, in essence, a search for the best *objective* when operating within a set of constraints. For example you want the *lightest* car that can at least deliver standard performance and possess good safety features while being powered a hybrid engine. Maybe you want to generate *maximum* shareholder return on investment while *meeting* the 10 percent sales growth and holding inventory at the *minimum*. Sometimes you want to *minimize* commuting time by choosing alternate routes during peak traffic periods. You can come up with several examples of this kind from everyday experience. What you have done is loosely define an *optimization* problem that needs a solution. The procedure by which you will establish a solution to the above examples uses *optimization techniques*. A second read through these sentences will suggest you are really talking about designs that are qualified by the words *lightest, maximum, minimize*. All of these, and many others like it can be described by the word **optimum**. In addition, these designs must be within a *certain envelope* or satisfy certain conditions or must be limited, if they are to be acceptable. These are the *constraints* on the design. With this basic definition you can identify problems of *design optimization*. Most of the book is about formally expressing these kinds of problems and looking at several techniques that can be employed to establish the solution.

Optimization is an essential part of design activity in all major disciplines. These disciplines are not restricted to engineering. In product development, competition demands producing economically relevant products with embedded quality. Today, globalization demands that additional dimensions such as location, language, and expertise must also merit consideration as new constraints in the development process. Improved production and design tools coupled with inexpensive computational resources have made optimization an important part of the process. 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. The ability to make the *best choice* is a perpetual desire among us all. The techniques that are used in optimization are also used for obtaining solutions to nonlinear problems in many disciplines—so the subject has an attraction to a wider audience from many fields.

In this book, optimization is often associated with **design**, be it a product, service or a 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. It forms part of the psyche of every aerospace designer. Just recently, Boeing introduced the *Dreamliner* to the world. The aircraft uses more than 50% plastic in its structure, replacing metals, without compromising on safety. Today, an emerging discipline is multidisciplinary optimization (MDO) of greater significance to aerospace designers, where structural design and aerodynamics are combined to produce an optimal vehicle shape. A good example is the Boeing blended wing-body design. 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 it has taken some time to mature. In the past, 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 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.

Today, it is inconceivable that current replacement products, such as the car, the house, the desk, the pencil, and so on, are not designed optimally in some sense or another. The most obvious contemporary example of the use of optimization manifests in the same streamlined bubblelike look in all cars from all manufacturers. Minimizing drag coefficient, particularly through software, improves fuel consumption. This is an easier option for meeting fuel consumption standards without requiring a change in the engine performance, which has proved very stubborn.

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 buy software that will optimize your connection to the Internet. You can have affordable access to resources that will allow you to perform all these various optimizations. Seeing the variety of problems that need to generate optimum solutions,

the study of optimization is actually more of a tool that can be applied to a variety of disciplines. If so, all of the optimization problems from many disciplines should be described in a common way.

The partnership between *design* and *optimization* activity is still more often found in *engineering*. This book recognizes that connection. Much 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. A Google search on "o*ptimization of*", at the time of this writing, resulted in 128 million hits, the first six of them related to optimization in digital circuits, trading system, immunomagnetic separation, object-oriented programming, risk measures, and chemical process.

1.1 OPTIMIZATION FUNDAMENTALS

Optimization was described as the process of *search* for the solution that is more useful than several others. Qualitatively, this assertion implicitly recognizes the necessity of choosing among alternatives. This book deals with optimization in a quantitative way. This means that an outcome of applying optimization techniques to the problem, design, or service must yield numbers that will define our solution—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 prior to the application of the optimization methods. In many situations, coming up with a mathematical model will prove to be very challenging. The development of a suitable mathematical model presupposes knowledge of content in the particular design area that the optimization problem is being formulated. Consider the design activity in the following cases:

1. New consumer research, with deference to the obesity problem among the general population, suggests that people should drink no more than about 0.25 liter of soda pop at a time. The fabrication cost of the redesigned soda can is proportional to the surface area, and can be estimated at $1.00 per square centimeter of the material used. A circular cross-section is the most plausible, given current tooling available for manufacture. For aesthetic reason, the height must be at least twice the diameter. Studies indicate that holding comfort requires a diameter between 5 and 8 cm. Create a design that will cost the least.
2. Design a cantilevered beam, of minimum mass, carrying 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 be sufficiently strong in bending and shear. There is also a limit on its deflection.

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.
4. The first-order differential equation $(t + 1)dy/dt - (t + 2)y = 0$, subject to the initial condition $y(0) = 1$, can be solved analytically using the power series method. Set up an alternate procedure using optimization.

This list represents four problems that will be used to define the elements of **problem formulation**. Each problem requires information from the specific area or discipline it refers to. To recognize or design these problems assumes that the designer is conversant with the particular subject matter. Such kinds of problem are quite common, and they are expressed in far greater detail than formulated here. The problems are kept simple to focus on optimization issues. The last example is included to show that optimization ideas and techniques can migrate very well to solve standard problems in a nontraditional manner. In fact, with creative formulation, you should be able to solve most problems through optimization. Optimization also provides an opportunity to establish useful values for designs that are unique or one of a kind.

1.1.1 Elements of Problem Formulation

In this section, we will introduce the formal elements of the optimization problem. Please keep in mind that optimization presupposes the knowledge of the design rules for the specific problem, primarily the ability to describe the design in mathematical terms. For engineering problems this means that the designer is aware of the relevant physics from the particular area, including all of the

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

expressions and techniques used in mathematical analysis of the design. The terms in optimization include **design variables, design parameters**, and **design functions**. Traditional design practice—that is, design without regard to optimization, includes all of these elements, except it is not necessary to formally recognize them as such. 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.

Design Variables: Design variables are entities that *define* a particular design. The values of a complete set of these variables will establish a specific design. In the search for the optimal design, the values of these entities will change over a prescribed range, hence the tag *variables*. The number of these design variables used to be very significant in the early days of optimization, with the recommendation that the set be as small a possible. This prohibition was related to available computational resources and is no longer a limitation in applied optimization. The type of these variables, *continuous*, or *discrete*, or *integer*, or *mixed*, is important in identifying and setting up the quantitative optimal design problem and the optimization procedure. It is crucial that this choice capture the essence of the object being designed and at the same time provide a quantitative characterization of the design problem. We will develop the elements of optimization assuming the variables are continuous. In applied mathematical terminology, design variables serve as the **unknowns** of the problem being solved. Using 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 size of the object, like 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. They must be **linearly independent**. This means that you cannot establish the value of one of the design variables viathe 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, height, 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 criteria of linear independence for applying the techniques of optimization. Many of these techniques are borrowed from linear algebra, where this property is necessary for a solution. 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 two.

The *set* of design variables is identified as the **design vector**. This vector will be column vector in this book. In fact, all vectors are column vectors in this book, unless indicated otherwise. The length of this vector, which is $\boldsymbol{n}$, 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 lowercase $\boldsymbol{x}$ for the variable and the uppercase $\boldsymbol{X}$ for the vector. All the techniques of optimization in the book are based on the **generic mathematical model**. The subscript on $\boldsymbol{x}$, 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 only necessary for mathematical convenience. This book will refer to the design variables in one of four ways:

1. $[\boldsymbol{X}]$: (the square parenthesis defines a vector) the vector of design variables.
2. $\boldsymbol{X}$ or $\boldsymbol{x}$ (without subscripts): referring to the vector again, omitting the square brackets for convenience if appropriate.
3. $[x_1, x_2, \ldots, x_n]^{\mathbf{t}}$: indicating the vector through its elements. Note the transposition symbol $^{\mathbf{t}}$ to identify it as a column vector.
4. x_i, $\mathrm{i} = 1, 2, \ldots, n$: referring to all the elements of the design vector.

The above notational convenience is extended to all vectors in the book. Once again, vectors refer to a collection or a related set of values.

Design Parameters: In this book, *design parameters* identify constants that will not change as different designs are generated and compared during optimization. Expressing the design and its properties requires more than design variables. Please be aware that 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. The book will draw attention to these issues when the occasion arises. Examples of parameters include material property, applied loads, and choice of shape. The parameters in the generic mathematical model are recognized similar to the design vector, except that we use the character $\boldsymbol{p}$. Therefore $[\boldsymbol{P}]$, $\boldsymbol{P}$, $\boldsymbol{p}$, $[p_1, p_2, \ldots, p_q]$ represent the parameters of the problem. Note the length of the parameter vector is $\boldsymbol{q}$. It is important to restate that except in the discussion of modeling, the parameters will not be explicitly referred, as they are primarily predetermined constants in the evaluation of the design.

Design Functions: The *design functions* will 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 constraints. The *design objective*, as its name implies, drives the search for the optimal design. The satisfaction of the *constraints* establishes the validity of the design. 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): This is a more specific name for the design objective function. The traditional design optimization problem is defined using a single objective function. The format of this statement is usually to minimize or maximize 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 or minimization** 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 a maximization problem. 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. 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 generic 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 $f(\boldsymbol{X})$. Single objective problems have only one function, denoted by $\boldsymbol{f}$. It is a scalar (not a vector). Note, although the objective function (and the other functions too) depends on $\boldsymbol{P}$ (parameter vector), it is not explicitly included in the format since it does not vary during the search for solution. We should recognize that if the parameter value changes, then the optimization problem must be solved again. In other words, the solution will be valid only for a defined parameter vector.

Multi-objective and **multi-disciplinary designs** are important developments today. Multi-objective design, or **multiple objective design**, refers to using several different design functions as objectives to drive the search for optimal design. Essentially, they permit a noticeably larger set of design solutions by permitting the mathematical model to represent design functions from several disciplines. 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 single reconstructed objective optimization problems 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 it as a single objective problem with additional constraints based on the remaining objective functions can usually generate a good solution. In multi-objective problems the *objective function* will be a vector.

Between the first and the second edition of this book, **multi-disciplinary optimization** (MDO) problems were being seriously addressed. They are predominantly addressed in aerospace industry, where people are reconciling

different requirements from structural design and aerodynamic design together. One of the requirements is to identify design variables that will define the structure as well as directly affect the aerodynamic analysis leading to new mathematical models. MDO optimization problems tend to be very large to accommodate aerodynamic analysis of reasonable fidelity. External aerodynamic problems are quite finicky and are difficult to deal with even without optimization. Another feature of the coupling between structure and aerodynamics is that objective function tends to have a large number of local minimums. In such cases, global optimization techniques, which are notoriously slow, must be part of the solution process. This is an exciting area, and definitely not for the resource poor. Although it raises 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: Design functions will be dependent on the design variables and parameters. A well-described optimization problem is expected to include several such functions that will ensure that the design will exist and interact well with its operating environment. Multiple constraint function can be represented as a vector of constraint functions. The format of these functions requires them to be compared to some numerical value that is established by design requirement, or the designer. This value remains constant during the optimization of the problem. The comparison is usually set up using the three standard *relational* operators: $=$, $\leq$, and $\geq$.

Consider our first example. Let $fun_1(X)$ represent the function that calculates the volume of the new soda can we are designing. The constraint on the design can be expressed as:

$$fun_1(X) = 250 \text{ cc}$$

In the second example, let $fun_2(X)$ be the function that calculates the deflection of the beam under the applied load. The constraint can be stated as:

$$fun_2(X) \leq 1 \text{ mm}$$

The constraint functions can be classified as *equality* constraints [like the one involving $fun_1(X)$ above] or *inequality* constraints [like the expression involving $fun_2(X)$ above].

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

Equality Constraints: Equality constraints are mathematically neat and analytically 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 $\boldsymbol{h}$. There may be more than one equality constraint in the design problem. A vector representation for equality constraints is introduced through the following representation. $[\boldsymbol{H}]$, $[h_1, h_2, \ldots, h_l]$, and h_k: $k = 1, 2, \ldots, l$ are ways of identifying the equality constraints. The dependence on the design variables $\mathbf{X}$ is often omitted for convenience. Note the length of the vector is $\boldsymbol{l}$. 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 of design variables in the problem, $\boldsymbol{n}$, must be greater than the number of equality constraints, $\boldsymbol{l}$, $(\boldsymbol{n} > \boldsymbol{l})$, for a valid optimization problem. If $\boldsymbol{n}$ is equal to $\boldsymbol{l}$, $(\boldsymbol{n} = \boldsymbol{l})$, then the problem can be solved without any reference to the design objective. If $(\boldsymbol{n} < \boldsymbol{l})$, then you have an over determined set of relations which could result in an inconsistent problem definition. The set of equality constraints must be ***linearly independent*** for a meaningful problem. Broadly, this implies you cannot obtain one of the constraints from elementary arithmetic operations on the remaining constraints. This is to ensure that the methods based on linear algebra will not fail. 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:

$$h_1(\mathbf{X}): \quad fun_1(X) - 250 = 0$$

In practical problems, equality constraints are rarely employed as they are less flexible compared to inequality constraints.

Inequality Constraints: Inequality constraints are more natural in problem formulation. They provide more choices for the design. The symbol representing inequality constraints in the abstract model is $\boldsymbol{g}$. There may be more than one inequality constraint in the design problem. The various vector representation for inequality constraints are $[G]$, $[g_1, g_2, \ldots, g_m]$, and g_j: $j = 1, 2, \ldots, m$. $\boldsymbol{m}$ represents the number of inequality constraints. All design functions explicitly or implicitly depend on the design (or independent) variable $\boldsymbol{X}$. $\boldsymbol{g}$ is used to describe both less than or equal to $(\leq)$ constraints and greater than or equal to $(\geq)$ constraints. The strictly greater than $(>)$ and the strictly less than $(<)$ are not used in optimization because the solutions are usually expected to lie at the constraint boundary $(\boldsymbol{g} = \mathbf{0})$. 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 $fun_2(X)$ is set up as:

$$g_1(\boldsymbol{X}): \quad fun_2(X) - 1 \leq 0$$

In the case of inequality constraints a distinction is made whether the design variables lie on the constraint boundary $(\boldsymbol{g} = \mathbf{0})$ or in the interior of the region bounded by the constraint $(\boldsymbol{g} < \mathbf{0})$. If the design variables determine a solution

on the boundary of the constraint the constraint acts like an equality constraint. In optimization terminology, this particular constraint is referred to as an **active constraint**. If the design variables determine a solution that does not lie on the constraint boundary, that is they lie inside the region of the constraints, they are considered **inactive constraints**. An inequality constraint can therefore be either *active* or *inactive*.

Side Constraints: Side constraints are necessary part of the solution techniques, especially numerical ones. It expresses the acceptable region for the design variable. Each design variable must be bound by numeric values for its lower and upper limit. The designer makes this choice based on anticipation of an acceptable design. The *design space*, the space that will be searched for optimal design, is the **Euclidean** or **Cartesian n-dimensional** space generated by the $\boldsymbol{n}$ independent design variables $\boldsymbol{X}$. This is a generalization of the three dimensional physical space we are familiar with. For 10 design variables, it is a 10-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 $\boldsymbol{n}$ dimensional considerations are mere extrapolation of the three dimensional reality. Although we cannot geometrically define them we can deal with the numbers describing the design. The **side constraints** limit the search region, implying that only solutions that lie within a certain region will be acceptable. It defines an $\boldsymbol{n}$ dimensional rectangular region (*hyper cube)* 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 problem cannot be 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 over zealously. There must be a sufficient space for the numerical techniques to conduct a robust search for the optimal design.

The Standard Format: These definitions allow us to assemble the generic *mathematical model* for optimization through the various design functions:

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

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

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

$$g_m(x_1, x_2, \ldots, x_n) \le 0$$

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

The same problem can be expressed concisely using this notation:

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

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

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

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

The generic model in vector notation (length of vector is shown in the definition):

$$\text{Minimize} \quad f(\boldsymbol{X}), [\boldsymbol{X}]_{\boldsymbol{n}} \tag{1.9}$$

$$\text{Subject to:} \quad [\boldsymbol{h}(\boldsymbol{X})]_{\boldsymbol{l}} = \boldsymbol{0} \tag{1.10}$$

$$[\boldsymbol{g}(\boldsymbol{X})]_{\boldsymbol{m}} \le \boldsymbol{0} \tag{1.11}$$

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

The previous mathematical model expresses the standard optimization problem in natural language as:

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

The techniques in the book will apply to the problem described in the standard format, which refers to a single objective function. To solve any specific design problem, it is required to reformulate the problem in this manner so that the methods can be applied directly. Also in the book, the techniques are developed progressively, starting from the standard model without constraints. 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 information 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 four design problems introduced earlier will be translated to the *standard format* after first identifying the mathematical model. The second problem requires information from a course in mechanics and materials. This should be within the scope of most engineering students.

Example 1.1 New consumer research, with deference to the obesity problem among the general population, suggests that people should drink no more than

about 0.25 liter of soda pop at a time. The fabrication cost of the redesigned soda can is proportional to the surface area, and can be estimated at \$1.00 per square centimeter of the material used. A circular cross-section is the most plausible, given current tooling available for manufacture. For aesthetic reason, the height must be at least twice the diameter. Studies indicate that holding comfort requires a diameter between 5 and 8 cm. Create a design that will cost the least.

Figure 1.1 represents a sketch of the can. In most product designs, particularly in engineering, it is necessary to work with a figure. The diameter d and the height h are sufficient to describe the soda can. What about the thickness t of the material of the can? What are the assumptions for the design problem? Is t small enough to be ignored in the calculation of the volume of soda in the can? Another important assumption could be that the can will be made using a given stock roll. Another one is that the material required for the can will include only the cylindrical surface area and the area of the bottom. The top will be fitted with an end cap that will provide the mechanism by which the soda can be poured. The top is not part of this design problem. In the first attempt at developing the mathematical, we could start out by considering the quantities identified, including the thickness, as design variables:

$$\text{Design variables:} \quad d, h, t$$

Reviewing the statement of the design problem, one of the parameters is the cost of material per unit area that is given as \$1.00 per square meter. Let us identify the cost of material per unit area as constant $\boldsymbol{C}$. During the search for the optimal solution this quantity will be held at the given value. Note, if this value changes then our cost of the can will correspondingly change. This is what we mean by a **design parameter**. Typically, change in parameters will require a new solution to the optimization problem:

$$\text{Design parameter:} \quad C$$

The design functions will include the computation of the volume enclosed by the can and the surface area of the material used. The volume in the can is $\pi d^2 h/4$. The surface area is $\pi dh + \pi d^2/4$. The aesthetic constraint requires that

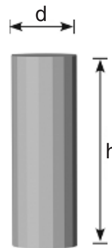

Figure 1.1 Example 1.1 — Design of a new beverage can.

h* ≥ 2*d. 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:

$$\text{Minimize} \quad f(d,h,t): \quad C(\pi dh + \pi d^2/4) \tag{1.13}$$

$$\text{Subject to:} \quad h_1(d,h,t)): \quad \pi d^2 h/4 - 250 = 0 \tag{1.14}$$

$$g_1(d,h,t): \quad 2d - h \le 0 \tag{1.15}$$

$$5 \le d \le 8; \quad 4 \le h \le 20; \quad 0.001 \le t \le 0.01$$

In the mathematical model (1.13–1.15) of the optimization problem for the first example the values of the design variables are expected to be expressed in consistent units-centimeters. It is the responsibility of the designer to ensure correct and accurate problem formulation including dimensions and units. Note that the cost C was originally expressed as meter squared. Hence a scaling factor must be used in (1.13). We will call this C_1.

Intuitively, there is some concern with the problem as expressed by the 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***, but we have already decided this value will not change during optimization). If this were a serious design example then the cans have to be designed for impact, stacking strength, and stresses occurring during transportation and handling. In that case ***t*** will probably be a critical design variable. This will require several additional structural constraints in the problem. Moreover, it is likely that development of these functions will not be a simple exercise. This could serve as an interesting extension to this problem for homework or project. The new mathematical model for the optimization problem after dropping t, and expressing [*d, h*] as $[x_1, x_2]$ becomes:

$$\text{Minimize} \quad f(x_1,x_2): C_1(\pi x_1 x_2 + \pi x_1^2/4) \tag{1.16}$$

$$\text{Subject to:} \quad h_1(x_1,x_2): \pi x_1^2 x_2/4 - 250 = 0 \tag{1.17}$$

$$g_1(x_1,x_2): 2x_1 - x_2 \le 0 \tag{1.18}$$

$$5 \le x_1 \le 8; \quad 4 \le x_2 \le 20$$

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, simple geometrical relations were sufficient to set up the optimization problem.

Example 1.2 Design a cantilevered beam, of minimum mass, carrying a point load F at the end of the beam of length L. The cross-section of the beam will

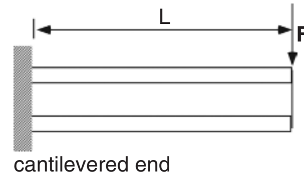

Figure 1.2 Example 1.2 — Cantilever beam.

be in the shape of the letter I (referred to as an I-beam). The beam should be sufficiently strong in bending and shear. There is also a limit on its deflection.

Figure 1.2 represents the side view of the beam carrying the load. Figure 1.3 describes 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 of the beam, which is uniform along the length. A handbook on mechanical engineering or a 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 the variables identified in Figure 1.3 will strongly affect the solution. They are good candidates for being 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?

Intuitively, the larger the value of F, the greater is the cross-sectional area necessary to handle it. Since the mass of the beam will be directly proportional to the area of cross-section, it can be easily concluded that if F were a design variable then it should be set at its lower limit, No techniques are required for this conclusion. If we know the optimum value of F without effort, then F is a good candidate for a parameter rather then a design variable. The larger the value of *L* the greater is the moment that F will generate about the cantilevered end. This will require an increase in the cross-section properties for strength of the structure. By the same reason, *L* is not a good choice for the design variable since

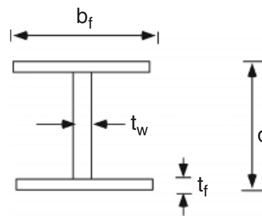

Figure 1.3 Example 1.2 — Cross-section shape.

it will be at its lowest value for an optimum design. To represent a material as a design variable we need to use its set of structural properties in the formulation of the design functions. Some useful material properties are: its specific weight, γ; 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}; in tension; its ultimate strength in shear, τ_{ult}. There are still more properties like coefficient of thermal expansion, conduction coefficient, and fatigue limit that will be useful is special problems. Optimization techniques will identify values of the design variables at the solution. 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, or a combination of material properties that are inconsistent with known ones. This for all intents and purposes will not generate a useful solution and it is certainly beyond the scope of the designer. Again, material as a parameter makes a lot of sense. To summarize, the optimization problem should be reinvestigated if F, L, or if the *material* changes. Concluding from this discussion on modeling, the following are defined:

$$\text{Design parameters:} \quad F, L, \{\gamma, E, G, \sigma_{\text{yield}}.\tau_{\text{yield}}, \sigma_{\text{ult}}, \tau_{\text{ult}}\}$$

$$\text{Design variables:} \quad d, t_w, b_f, t_f$$

In the development of the mathematical model, standard technical definitions are used. Much of the information can be obtained from a mechanical engineering handbook, or an elementary book on mechanics of materials. 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 A_c of the cross-section of the beam is $(2b_f t_f + t_w(d - 2t_f))$. 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 moment of inertia I_c, in terms of the design variables, is $(b_f d^3/12) - ((b_f - t_w)(d - 2t_f)^3/12)$. 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 first moment of area Q_c can be calculated as $(0.5b_f t_f(d - t_f) + 0.5t_w(d - t_f)^2)$. The maximum deflection (δ_L) of the beam will be at the end of the beam calculated by the expression $FL^3/3EI_c$. In the following, for ease of representation 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 $\boldsymbol{X} = [x_1, x_2, x_3, x_4]$ and the problem in standard format is as follows:

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

$$\text{Subject to:} \quad g_1(\boldsymbol{X})\text{:}\ FLx_1/2I_c - \sigma_{\text{yield}} \le 0 \tag{1.20a}$$

$$g_2(\boldsymbol{X})\text{:}\ FQ_c/I_c x_2 - \tau_{\text{yield}} \le 0 \tag{1.20b}$$

$$g_3(\boldsymbol{X})\text{:}\ FL^3/3EI_c - \delta_{\max} \le 0 \tag{1.20c}$$

$$0.01 \le x_1 \le 0.25; \quad 0.001 \le x_2 \le 0.05; \tag{1.20d}$$

$$0.01 \le x_3 \le 0.25; \quad 0.001 \le x_4 \le 0.05 \tag{1.20e}$$

The designer must ensure that the problem definition is also consistent with the unit system chosen for the parameters and variables. 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; $\sigma_{\text{yield}} = 250E + 06$ N/m^2; $\tau_{\text{yield}} = 145E + 06$ 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 if symmetry was imposed. Standard failure criteria with respect to combined stresses or principal stresses can also be included through additional functions. If the cantilevered end is bolted, then additional design functions regarding bolt failure needs 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.

$$\text{Design variables:} \quad x_1, x_2, x_3$$

The values in the tables can be regarded as parameters. For this problem, it is more useful to work with the values directly and hence identifying them as parameters does not serve any useful purpose. The information in Table 1.1 is used to set up the design functions in terms of the design variables directly. 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 the number of machines (g_1). The machines must satisfy the floor space constraint (g_2). 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 operator 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:

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

or

$$\text{Minimize} \quad f(\boldsymbol{X}): \; -18^{*}55^{*}x_1 - 18^{*}50^{*}x_2 - 21^{*}50^{*}x_3$$

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

$$g_2(\boldsymbol{X}): 3x_1 - x_2 + x_3 \le 30 \tag{1.22b}$$

$$g_3(\boldsymbol{X}): 3x_1 + 6x_2 + 6x_3 \le 100 \tag{1.22c}$$

$$x_1 \ge 0; \;\; x_2 \ge 0; \;\; x_3 \ge 0 \tag{1.22d}$$

Equations (1.21) and (1.22) express the mathematical model of the problem. Note in this problem that 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 (1.22) are different from the previous two examples. Here, the right-hand side of the less than and equal to 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 solution technique for this type of problem is very different than the solution to first two examples, which are recognized as **nonlinear programming problems**. Linear programming problems are essential in decision making in commerce and business. They are critically explored in those disciplines. Before moving on to the next section, the inequality constraint in (1.22a) stands out because of the large numbers in all of the terms. Dividing the constraint through by 1,000,000 will not change the problem. This is termed **scaling**, and is an important step in optimization.

Example 1.4 The first-order differential equation $(t+1)dy/dt - (t+2)y = 0$, subject to the initial condition $y(0) = 1$, can be solved analytically using the power series method. Set up an alternate procedure using optimization.

The differential equation and initial condition is rewritten as follows:

$$(t+1)y' - (t+2)y = 0; \quad 0 \le t \le 3 \tag{1.23}$$

$$y(0) = 1 \tag{1.24}$$

The solution, in terms of power series, can be expressed as

$$y(t) = \sum_{m=0}^{\infty} c_m t^m = c_0 + c_1 t + c_2 t^2 + \cdots \tag{1.25}$$

The initial condition in (1.24) will determine the value of $\boldsymbol{c}_0$ as 1. That leaves the determination of the remaining coefficients. These will be our design variables of the problem. Let us consider the series solution for this problem to 10 terms. We require *nine design variables*. We will also monitor the differential equation at 100 points in the interval of t, labeled as t_i. The solution, at each t_i:

$$y(t_i) = y_i = 1 + x_1 t_i + x_2 t_i^2 + \cdots + x_9 t_i^9; \quad i = 1..100 \tag{1.26a}$$

where the x_j are the design variables. The derivatives at each i can be expressed as

$$y'(t_i) = y_i' = x_1 + 2x_2 t_i + \cdots + 9x_9 t_i^8; \quad i = 1..100 \tag{1.26b}$$

At each t_i Equation (1.23) will measure the error in the series solution (if the term is nonzero).

$$E_i = (t_i + 1)y_i' - (t_i + 2)y_i$$

We will drive this error to zero by setting up our objective function as

$$f(X) = \sum_{i=1}^{100} E_i^2 \tag{1.27}$$

This can be stated as the minimum of the square error over the range of the variable indicated at the 100 points. This is an unconstrained optimization problem. The exact solution to the problem is

$$y(t) = (1+t)e^t = 1 + 2t + \frac{3}{2}t^2 + \frac{2}{3}t^3 + \frac{5}{24}t^4 + \frac{1}{20}t^5 \cdots$$
$$+ \frac{7}{720}t^6 + \frac{1}{630}t^7 + \frac{1}{4480}t^8 + \frac{1}{36288}t^9 + \frac{1}{329891}t^{10} \tag{1.28}$$

As formulated, the optimization problem has nine design variables. You could do this to other classical methods of solution, too. If it works, you are numerically identifying an analytical solution. Another advantage is that it is a simple matter to identify a different example by this method, but the analytical procedure must be worked out in detail for each different example. Chapter 11 addresses using optimization in nontraditional ways to solve nonlinear differential equations using Bezier surfaces, using exactly this idea.

1.1.3 Nature of Solution

We will look at the nature of solutions to the optimization problem using the first three examples we have modeled. Examples 1.1 and 1.2 describe a nonlinear programming problem. This means that some of the design functions in the model are nonlinear. They contain terms that include the product of design variables, or the variables themselves raised to powers other than 1. Example 1.3, as noted earlier

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 non-linear programming problems. Example 1.4 is a nonlinear problem in nine variables because the terms are squared. It is quite simple to solve numerically, especially using the optimization toolbox. In this chapter it has served to showcase how optimization can provide an alternative to a classical method of solution to math problems. Examples 1.1 and 1.2 are dealt first. Between the two, we notice that Example 1.1 is quite simple in comparison to Example 1.2. Second, the four variables in Example 1.2 make it difficult to use illustration to establish some of the discussion. We will use Example 1.1 in the following 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 represents the three-dimensional plot of Example 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. The region of the solution, which is at the intersections of the functions, is hidden and difficult to see. Figure 1.5, the contour plot, is an alternate representation of the problem. The solution will be clearer in this figure. The horizontal axis represents the diameter (d, x_1), The vertical axis represents the height (h, x_2). In Figure 1.5 the *equality*

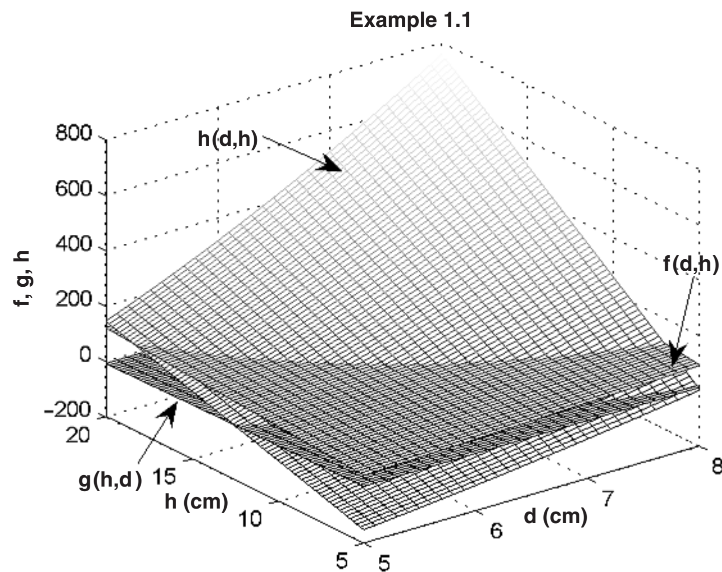

Figure 1.4 Graphical representation of Example 1.1.

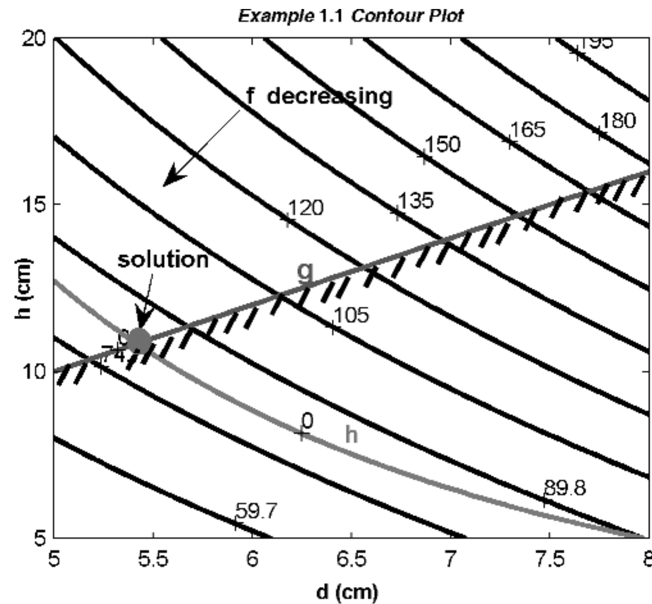

Figure 1.5 Contour plot of Example 1.1.

constraint $h(x_1, x_2)$ is marked appropriately as h on the figure. Since there is only one constraint the subscript is unnecessary. Any pair of values on this line will give a volume of 250 cc. The *inequality constraint* g is also identified. The pair of values on this line *exactly* satisfies the aesthetic requirement. The dashed lines on the side of the inequality constraint establish the *disallowed* region for the design variables. The constraints are drawn thicker for emphasis. The *feasible* region is on the other side of the dashed lines. The *objective function* is represented through several contours. Each contour is associated with a fixed value of the objective function and these values are shown on the figure. Since this is a minimum problem we are interested in the decreasing cost. The range of the two axes establishes the *side constraints*. The objective function f, and the equality constraint h are non-linear and therefore they are not drawn as straight lines. This is substantiated by the products of the two unknowns (design variables) in both these functions. The inequality constraint g 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 while 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 is a graphical solution to the optimization problem. First, the *design space* is the region spanned by the side

constraints—the boxed area. The *feasible* region is the space of acceptable designs. The presence of equality constraints complicates this discussion, since the solution must lie on the constraint. In this problem the feasible region is *on the equality constraint* above the intersection with the inequality constraint and limited by the y axis. The *optimal* solution is the feasible solution with the minimum value for the objective function. For this example, there are many feasible solutions (points on the feasible portion of the equality constraint) but only one optimal solution. There are *infinite* feasible solutions but a *unique* optimal solution. In this problem/figure, the smallest value of f is desired. The lowest value of f is about 82. Since this is in dollars, this can is quite expensive. Maybe there was an error in the cost. It should have been per square meter. Will it change the optimum solution? We will look at this and other answers later (you can run the code and see if it changes the solution). The solution to the optimization problem 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 5.5 and 11 cm, respectively. From Figure 1.5, it can be seen that g is an **active** constraint.

Solution to Example 1.3 In contrast to Example 1.1, all the relationships in the mathematical model, expressed by Equations (1.21) and (1.22), are linear. The word equation is generally used to described the equivalence of two quantities on either side of the *equal* (**=**) 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. This reduces the problem to two design variables allowing a graphical solution to be constructed. The mathematical model, with two design variables x_1, x_2, is reconstructed using the information that x_3 is now a parameter with the value of five. Substitute for x_3 in the model for Example 1.3, and cleaning up the first constraint:

$$\text{Maximize} \quad f(\boldsymbol{X}) \colon 990^{*}x_1 + 900^{*}x_2 + 5{,}250 \tag{1.29}$$

$$\text{Subject to:} \quad g_1(\boldsymbol{X}) \colon 0.4x_1 + 0.6x_2 \le 8.5 \tag{1.30a}$$

$$g_2(\boldsymbol{X}) \colon 3x_1 - x_2 \le 25 \tag{1.30b}$$

$$g_3(\boldsymbol{X}) \colon 3x_1 + 6x_2 \le 70 \tag{1.30c}$$

$$x_1 \ge 0; \quad x_2 \ge 0 \tag{1.30d}$$

An interesting observation in this 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

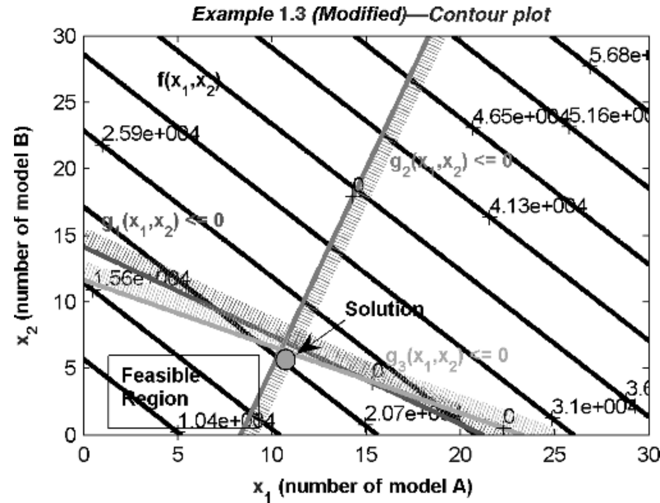

Figure 1.6 Contour plot of modified Example 1.3.

the number of equality constraints. Therefore, the *inequality* constraints are not related to the number of design variables used. This fact also applies to nonlinear constraints. Figure 1.6 is the plot of the functions (1.29) and (1.30). Choosing the first quadrant also satisfies the side constraints. The linearity of all the functions is indicated by the straight lines on the figure. Again, several contours of f are shown to aid the recognition of the solution. The figure was generated by MATLAB. A special piece code will generate the hash marks in MATLAB. In lien of hash marks, sometimes a thicker line is drawn parallel to the constraint ahead of the hash lines—or the forbidden region. The feasible region is enclosed by the two axes and the 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 spotted at the intersection of the constraints g_2 and g_3. The values of the design variables at the solution can be read off from the plot. We are not done yet. Before proceeding further it is necessary to acknowledge that a solution of 11.5 machines for model A 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 course. Many real-life designs, especially in the use of off-the-shelf materials and components determine discrete models. Discrete/integer programming is the area that addresses these problems. The nature of the methods searching

the optimal solution in discrete programming is very different from those 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, except in the Chapter 8, are 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 continuous optimization technique we develop in this book. Chapter 8 discusses discrete optimization with examples.

Getting back to the *linear programming* problem we just discussed, it can be observed that only the boundary of the feasible region will effect the solution. Here is a way that will make it 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 current objective function. Note the solution always appears to lie at the intersection of the constraints or the *corners*. Solution can never come from inside of the feasible region, unlike in *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 the solution could lie inside the feasible region. Figure 1.7 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 modified Example 1.3 and Figure 1.6.

Nonlinear Problems: In practice, solutions to nonlinear problems are obtained through *numerical techniques* that are applied iteratively. The methods or techniques for finding the solution to optimization problems are called *search methods*. These methods will require the following:

- Several *iterations* before the solution can be obtained.
- Each iteration or search is executed in a consistent manner where information from the previous iteration is utilized in the compute values in the present sequence.
- This consistent manner or process by which the search is carried out 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.

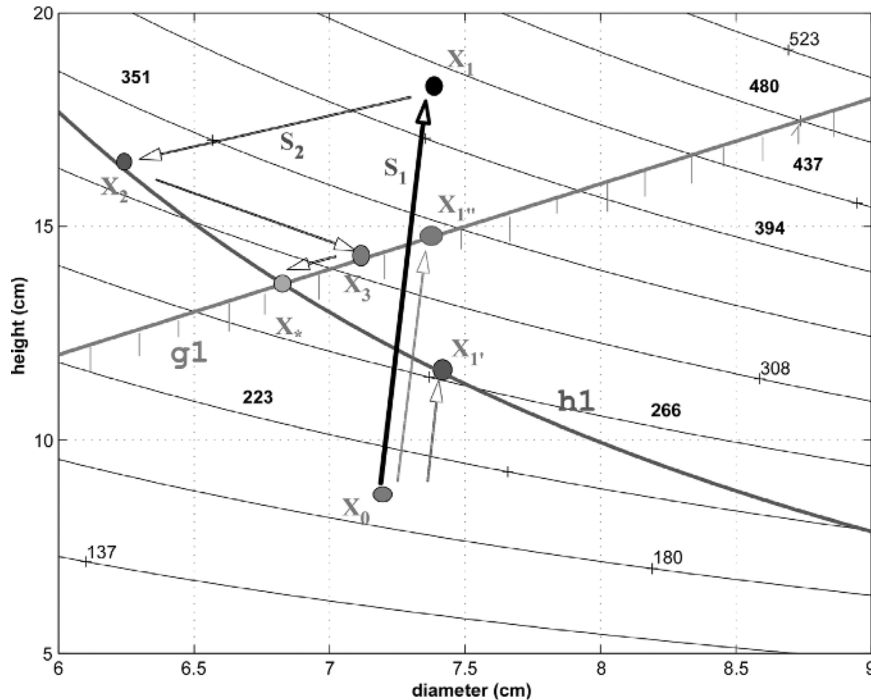

Figure 1.7 A typical search method for nonlinear problems.

- The algorithm is applied by converting these steps, through some computer language, into code that will execute on the computer. In this book, the algorithm is developed and translated into MATLAB code.

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 given the nonlinear nature of the problems, the success of optimization may hinge on his ability to choose a good initial solution. Any search method, even when used consistently, may fail to solve a particular problem. The degree and type of nonlinearity may frequently cause the method to fail, assuming there are no inconsistencies in problem formulation. Another method may yet solve the problem. 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.

These iterative methods are currently the only way to implement numerical solutions to *nonlinear* problems, irrespective of the discipline in which these problems appear or the nature of the problems they define. Nonlinear problems are plentiful in optimization, optimal control, structural and fluid mechanics, quantum physics, and astronomy. Although 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 dynamics systems, which are described by non-linear differential equations. In structural mechanics, there is the possibility of nonlinear integral equations. Computational fluid dynamics deals mostly with nonlinear partial differential equations. The principal aim of search methods is 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.

The graphical representation of a problem similar to Example 1.1 can be seen in the background of Figure 1.7. It is embellished with several iterations of a *hypothetical* search method. The initial starting guess is the point marked X_0. The solution is at X_*. Remember, 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 algorithm and methods. Using two variables also allow the concepts to be reinforced using geometry and graphics.

The design at the point $\boldsymbol{X}_0$ is not feasible. It is not useful to discuss if it is an optimal solution. At least one more iteration is necessary to obtain the solution. Consider the next solution at $\boldsymbol{X}_1$. Most methods in optimization will attempt to move to $\boldsymbol{X}_1$ by first identifying a **search direction**, $\mathbf{S}_1$. This is a **vector** (same dimension as design variable vector $\boldsymbol{X}$) pointing from $\boldsymbol{X}_0$ to $\boldsymbol{X}_1$. The actual displacement from $\boldsymbol{X}_0$ to $\boldsymbol{X}_1$, call it $\boldsymbol{\Delta X}_1$ is some multiple of $\boldsymbol{S}_1$, say $\boldsymbol{\alpha S}_1$. $\boldsymbol{X}_1$ once again is not feasible though it as improvement on $\boldsymbol{X}_0$ since it satisfies the inequality constraint $\boldsymbol{g}_1$ (lying in the interior of the constraint). Alternately, using the same search direction $\mathbf{S}_1$, another technique/method may choose for its next point, $\boldsymbol{X}_{1'}$, at a distance $\boldsymbol{\beta S}_1$. This point satisfies the constraint $\boldsymbol{h}_1$ but violates $\boldsymbol{g}_1$. A third strategy would be to choose $\boldsymbol{X}_1$. This lies on the constraint boundary of $\boldsymbol{g}_1$. Please note that all the three choices are not feasible. Which point is chosen is decided by the particular implementation of the algorithm. Assume point $\boldsymbol{X}_1$ is the next point in this discussion.

Point $\boldsymbol{X}_1$ now becomes the new point $\boldsymbol{X}_0$. A new search direction $\boldsymbol{S}_2$ is determined. Point $\boldsymbol{X}_2$ is located. Another sequence of calculations locates $\boldsymbol{X}_3$. The next iteration yields the solution. The essence of updating the design during successive iteration lies in two major computations, the **search direction** $\boldsymbol{S_i}$ and **stepsize** $\boldsymbol{\alpha}$. The change in the design vector, defined as $\boldsymbol{\Delta X}$, which represents the vector difference, for example, $(\boldsymbol{X_2} - \boldsymbol{X_1})$, is obtained as $\boldsymbol{\alpha S_i}$. For each iteration, the following sequence of activity can be identified:

Step 0: Choose $\boldsymbol{X_0}$

Step 1: Identify $\boldsymbol{S}$
Determine $\boldsymbol{\alpha}$
$\boldsymbol{\Delta X} = \boldsymbol{\alpha S}$

$X_{new} = X_0 + \Delta X$
Set $X_0 \leftarrow X_{new}$
Go to Step 0

This 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 we will explore in this book are mostly different by 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 will be several ways to satisfy this condition.

Linear Programming Problem: It was previously mentioned that the solution to these problems would be on the boundary of the feasible region (also called *feasible domain*). In Figure 1.6, the axes and some of the constraints determine this region. This region 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. You have to acquire a *feasible* corner to start the iteration. This will be explored more fully in the chapter on linear programming. A reminder here once again 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.

So far, Chapter 1 has dealt with the introduction of the optimization problem. It was approached from an engineering design perspective, but the mathematical model, in abstract terms, is not sensitive to any particular discipline. A few design rules with respect to specific examples were included in the discussion. The standard mathematical model for optimization problem 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 representation of the model. 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 iterative computation.

The numerical methods used in this area are iterative methods. Numerical solutions are obtained through running computer programs. These programs involve two components. The first is an algorithm that will establish the iterative set of calculations. The second is the translation of the algorithm into computer codes, using a *programming language*. This finished code is referred to as *software*. The early software in optimization ran on *mai frames*, or large computer systems, using legacy programs written in FORTRAN, Pascal, or C. Since the mid-1990s the standard computation environment has transformed to individual personal

computers (PCs), running Windows, Macintosh, Linux, or other flavors of UNIX, operating systems. They are more powerful than old mainframe computers and are inexpensive to boot. Users interact with their PCs through a graphical user interface (GUI) avoiding time and tedious effort in learning the software and as well as setting up the problem. New *programming paradigms*, especially, *object-oriented programming*, has thrown up a lot of new and improved ways to develop software. Today engineering software is created by general programming languages such as C, C++, Java, Pascal, and Visual Basic. Many software vendors develop their own extensions to the standard language, providing a complete environment for specific deployment 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, Inc. with their flagship products MATLAB and Simulink. They also provide a complementary collection of Toolboxes, which deliver software for specific disciplines and applications. Between the first and the second edition of this book, the use of MATLAB has grown exponentially in academia and industry. It has also evolved from core mathematical areas to envelop almost every discipline. Of specific note is the *Optimization Toolbox*, which implements most of the concepts developed in the book. Knowledge of MATLAB is a skill that is in demand from those entering the workforce to work in analytical areas. In this book, MATLAB is used for numerical and graphical 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 the book, the reader will become 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, an enormous part of MATLAB will not be used. It should be possible to explore those with the experience gained in this course. 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 with its use. The reader will need access to MATLAB to be able to use this book effectively.

1.2 INTRODUCTION TO MATLAB

MATLAB is introduced by Mathworks as the language for technical computing. Borrowing from the description in an earlier brochure, valid even now, 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 the following:

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

The use of MATLAB has exploded recently and so has its features. This continuous enhancement makes MATLAB an ideal vehicle in exploring problems in all disciplines that manipulate mathematical content. This ability is multiplied by application-specific solutions through *toolboxes*. To know 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, please visit MATLAB at http://www.mathworks.com.

1.2.1 Why MATLAB?

An answer may have been necessary at the time of the first edition of this book. Today, with MATLAB being used in over 3,500 universities and over 1,000 research and industrial institutions across the globe, it would be a serious omission *not* to incorporate its support in learning about applied optimization. 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 the subject. In the next chapter, its powerful visualization features are used for graphical optimization. In the rest of the book the other built in features of MATLAB is utilized to translate algorithms into functioning code, in a fraction of the time needed in other languages such as C or FORTRAN. At the end, we will see some examples from its optimization toolbox.

Most books on optimization provide excellent coverage of the techniques and algorithms, and some provide a printed version of the code. Almost all new editions have moved away from legacy environments to embrace new software environments that are user friendly, with shallower learning curves. These software systems are feature rich and very resourceful in delivering various aspects of computation. MATLAB's code is compact, and it is reasonably intuitive. To make problems work requires only few lines of code compared to traditional computer languages. Instead of programming from the ground up, standard MATLAB pieces of code can be threaded together to develop the new algorithm. It avoids the standard separate compilation, link, and execution 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 as they are text files. 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. MATLAB is a globally relevant product and can work with C and Java if desired.

1.2.2 MATLAB Installation Issues

The code in this book will work on MATLAB installed on an individual PC or Mac, a networked PC or Mac, or individual or networked UNIX workstations. MATLAB takes care of the issue of portability as user-created code is in ASCII text. The book assumes you have a working and appropriately licensed copy of MATLAB in the machine in front of you. This book uses MATLAB Version 7.2 (R2006a), as this was the current version when the manuscript was started. The version shipping during the first print is R2008a. The code was developed on a PC. During the development of the previous edition, the author had access to UNIX and Mac platforms. This time the development is based on a 32-bit Windows PC. New releases of MATLAB are supposed to be seamless across platforms in terms of user use and experience. If there are issues that are different, the author solicits feedback from the user. In terms of code itself, newer versions of MATLAB should not be a problem. There was however some significant changes in some commands with version 7. The same command can be run in a backwards-compatible form by including 'V6' as the first item in the Command. Installing MATLAB on your machine should not be a problem, as the software is accompanied by detailed installation instructions. If you are using a networked version, please consult your system administrator. While many of you may have been exposed to MATLAB, this book assumes that you are new to MATLAB. You are strongly urged to use the MATLAB help resources through its excellent documentation system, particularly the Help browser.

Start MATLAB from the start menu on the PC or through an icon on the desktop (Mac, UNIX, Linux users should find the program as a menu item in the appropriate menu). It is a good idea to click the various menus on the top menubar. You will see various menu sub-items that are some of the common commands you are likely to use. You can also customize how MATLAB will tile the various windows when it opens. My personal preference is using only the Command window and the Editor window in stand-alone mode instead of the default appearance, since it will be less distracting. If you are already a MATLAB user, you are welcome to use your favorite setup. You can change the way MATLAB opens by clicking through the menus *Desktop* → *Desktop Layout* → *Command Window Only*.

The MATLAB window, Figure 1.8, also called the **Command window**, is the means through which commands are issued to MATLAB during interactive use. Note that MATLAB command prompt is ≫. The second graphical window that will be very useful is the MATLAB M-file **Editor/Debugger window**, shown in Figure 1.9. This window is opened by clicking the *new* file icon in the Command

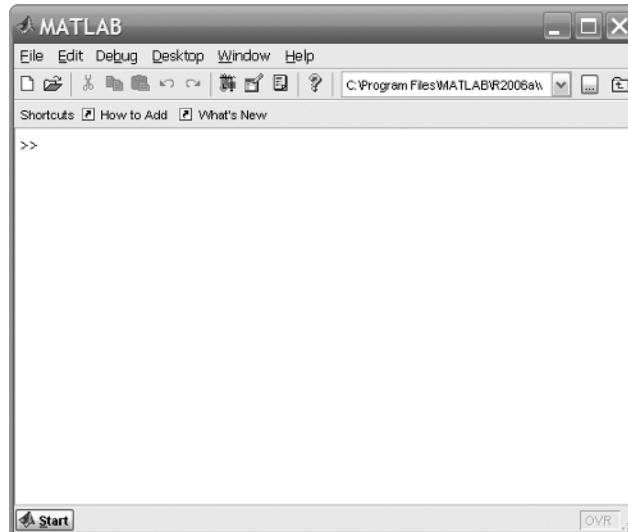

Figure 1.8 MATLAB Command window in the PC.

window under the *File* menu. This is the window where we will write the code. The editor uses color for enhancing code readability. It is also a **debugger;** it helps us find and correct errors in code. The Command window and the Editor window can spawn additional dialog boxes (if necessary) for setting additional features during the MATLAB session, like the path *browser* and *workspace browser*. These windows are very similar in other platforms.

Although this is not so much an installation issue, it would be appropriate to discuss the way the 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, the set of commands that were used and what worked is stored in the **History window**. Large sequences of commands are usually difficult to keep track off in an interactive session. Another option is to store the sequence of commands in a text-file (ASCII), and execute the complete file in MATLAB. This is called a **script m-file** and can be considered a batch execution—a batch of commands. 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 are saved, and the file is executed in MATLAB again. Executing MATLAB through script files is predominantly followed in the book. MATLAB allows you to revert to interactive mode any time by just typing in the Command window. In order to enhance the programming experience and expose you to more features of MATLAB, a lot of bookkeeping activities are done through code rather than the rich resources provided by newer versions of MATLAB. In a sense, interactive use

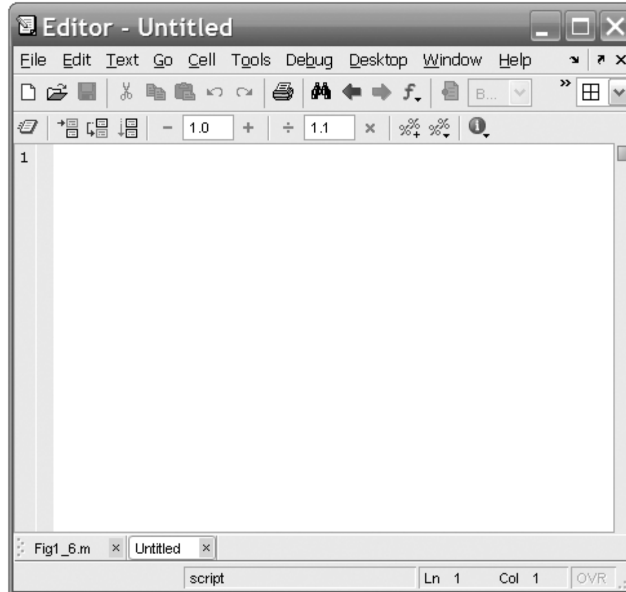

Figure 1.9 MATLAB editor/debugger window on the PC.

of MATLAB is eschewed so that code will run faster and the user is introduced to more programming.

1.2.3 Using MATLAB the First Time

MATLAB will be used interactively in this section. The author strongly recommends a hands-on approach to understanding MATLAB. This means typing the code yourself. This is essential to understand the syntax errors, the debugging, and rectifying of the code. The author is not aware of a single example where the reader learned programming without typing/entering code. Moreover, *your* intuitive understanding of MATLAB commands and programming is based on *your* practice and not the authors. In this book, MATLAB code segments are courier font with the boldface style used for emphasizing commands or other pieces of information. Anything else is recommendations, suggestions, or exercises. The script file name containing the commands is in italic.

Before We Start: Table 1.2 includes a few punctuation and special characters that you will use often. It is a good idea to become familiar with them. 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, which is (**==**).

Table 1.2 Some Punctuation and Special Characters

Punctuation/ Characters	What It Does
>>	MATLAB command prompt.
	You have to wait for the prompt before MATLAB will listen to you. It applies mostly for interactive sessions or when MATLAB is busy computing (see ^c).
%	All text to the right of the % is a comment and is ignored by MATLAB.
	Commenting your code seriously helps you remember what you were trying to achieve when you wrote the piece of code.
	Comments are used liberally in the code in the book. Without comments you will have difficulty understanding what you coded just a week ago.
%%	Together, they give you a bold comment.
;	A semicolon prevents information echoing to the screen.
	Removing it will cause information to be printed to the screen. This is useful when you want to debug the code.
,	A comma is used as a command separator and will cause information to be printed to the screen. You can also use a semicolon as a command separator.
:	A colon is used to establish a range of values. By itself, it implies all of the values.
...	A succession of three periods at the end of the line implies that the code will continue to the next line.
	You cannot continue variable name across lines. You cannot continue a comment on another line.
^c	Ctrl C (hold down Ctrl and C together) will stop MATLAB execution and return the control prompt.

Some Additional Features:

- MATLAB is **case sensitive**. An a is different than A. This is difficult for persons who are used to FORTRAN. All MATLAB built-in commands are in *lowercase*. Hence, you can define your own variables in uppercase so that they do not coincide with built-in functions or variable names.
- MATLAB does not need a **type definition** or a **dimension** statement to use and 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 any number of characters which can only include letters, digits, and underscore. Only the first 63 characters are used by MATLAB.

- MATLAB uses some built-in variable names and keywords. Avoid using built in variable names. For example the variable pi has the value of π or 3.14. It is probably a good idea to declare all your constants in the code.
- 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 (32-bit operating system). 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 (e.g., **1, −9, +9.0, 0.001, 99.9999**).
- Scientific notation is expressed with the letter **e**, (e.g., **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 subscripts must match)
−	Subtraction (same as above)
*	Multiplication (the size 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 only 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.

1.2.4 An Interactive Session

Start MATLAB. In the MATLAB Command window, there should be the MATLAB prompt followed by a blinking cursor. MATLAB is ready to go to work. Please hit return at the end of the line (or before the comment, as you do not need to type the comments in this exercise). Please do read the comments as they contain useful information. Please feel free to try your own variations and extend the exercise. In fact, to understand and reinforce the commands, it is recommended that you make up your own examples often in this exercise.

```
>> 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                  % echoes the value stored in a
>> A = 1.5;     % another variable A
>> a, A         % echoes a and A - Note case matters
>> aA = a*A     % multiplying a and A and a new variable

>> one = a; two = b; three = c;       % new variables
>> % assigns the values of a, b, c to new variables

>> four = d; five = e; six = pi % value of pi is 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 size information is assigned to AA1
>> size(AA1)   % AA1 is a one by two matrix

>> 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?

>> 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
>> % by default row vectors are created

>> C3 = [C1 D1] % C1 is augmented by an extra column
>> C3 = [C3 ; C2(3,:)]
>> % 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 column elements

>> 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 .* is represents the product of each element
>> % of C2 multiplied with the corresponding
>> % element of C3.  Such a multiplication is
>> % not defined mathematically

>> 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 must be terminated with "end"
>> % the prompt does not appear until "end" in entered

>> plot(sin(0.01*f),cos(0.03*f))
>> % plots an x-y plot in the Figure Window
>> % the Figure window is the third MATLAB window
>> % you will use often to display graphics
>> % it is also loaded with rich features
>> % that you can explore by yourself
>> % about the MATLAB command used above - plot
>> % all commands in MATLAB are between regular parenthesis
>> % first element of plot is the x information
>> % second element of plot is the y information
```

```
>> xlabel('sin(0.01*f)') % text appear in single quotes
>> % text is also called strings
>> 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
>> % you can exit using the exit button on the top right
```

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.

This session introduced

- The MATLAB Command Window and Workspace
- Variable assignment
- Basic Matrix operations
- Accessing rows and columns
- Suppressing echoes
- **Who, inverse** commands
- .* multiplication
- Figure window
- Basic plotting commands

In the next session, we will use the m-file editor to gain additional experience in using MATLAB. In this case we will not be interactively using MATLAB.

1.2.5 Using the Editor

In this section, we will use the MATLAB editor to create and run a script file. A script file is a collection of commands executed in sequence by MATLAB. This is also called batch execution. The file has to be saved before execution. Normally, the editor is used to generate two kinds of MATLAB files. These files are termed as **script files** and **function files**. Both these files contain MATLAB commands like the ones we have used. However, the second type of files has a specified format. Both file types should have the extension .m. Though these files are ASCII text files, and can be generated in any text editor, the generic .m extension should be used because MATLAB searches for this extension. The MATLAB editor will append this extension automatically as the file is saved. This extension is unique to MATLAB.

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 by just typing commands in the workspace window after the prompt. You can also work with the variables created by the script file.

The MATLAB editor uses color to identify MATLAB statements and elements. It also provides the current values of the variables (if/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 PCs. 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 need not be opened. The Editor provides its own window for entering the script statements.

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. We will save the commands in a file called *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. This will open an explorer type window where you can save the file, including creating a directory to save it in. If you want to break this session in segments, you can save your current work and continue where you left off, by traversing to the directory where you stored it and opening it in the editor. You can also open the file by typing line commands in the Command window. You will need the full **path** to this file. We plan to save this file in the sub-directory *Ch1* of the directory *Opt_book*, which is off the main file directory *C*. The path for the file is therefore **C:\Opt_book\Ch1\script1.m**. Note, the path here is specified as a PC path description. In UNIX, usually the slashes are forward slashes. The reason we need this information is to inform MATLAB where to find the file. We do this in the MATLAB Command window using an **addpath** command to inform MATLAB of the location of the directory to search for the file:

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

The simpler way to do this is to open the Editor window, find the file script1.m by traversing the directory, and open it. When you try and run the file, MATLAB will ask you if you want to add the directory to the path, and you can accept the choice. This way you will not need to know about the *addpath* command. The

script that was created in script1.m can be run by typing (note the extension is omitted).

```
>> script1
```

It can also be saved and run by clicking the icon that represents a *written page with a down arrow*, called the *RUN* icon, located at the right of the group of icons below the menu bar in the Editor window. If you just created the file and are clicking it for the first time a save file window will appear before the code is executed.

To understand the programming concepts in the script, particularly for users 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. You can run by clicking the RUN icon, 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. Open the MATLAB editor and type the following code. The code is in courier font.

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

```
% script1.m - (second edition)
clear  % clear all values - clean start
clc    % position curser at the top of the screen
format compact % print single spacing on the screen
               % default is double spacing

%   example of using script
A1 = [1 2 3];  % a row vector
A2 = [4 5 6];  % another row vector
% the commands not terminated with semi-colon will display
% information on the screen
A = [A1; A2]   % a 2X3 matrix
B = [A1' A2']  % a 3x2 matrix
C = A*B        % matrix multiplication

%  press the run icon and save file as script1.m in
% a directory.  Look at the output in command window
% you should see A, B, C

%  you can use blank lines to make the code readable

% now recreate 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          % print a1 on the screen

%  press the run icon and look at the output
%  in command window


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

%  press the run icon and look at the output
%  in command window

% creating matrix AA and BB
for i = 1:3
    AA(1,i) = a1(1,i);
    AA(2,i) = a2(1,i);
    BB(i,1) = a1(1,i);
    BB(i,2) = a2(1,i);
end

AA      % print the value of AA in the window
BB      % print the value of BB

%  press the run icon and look at the output
%  in command window

% instead of the for statement we could have
AAA(1,:) = a1;  % first row of AAA is the vector a1
AAA(2,:) = a2;  % second row of AAA is vector a2

BBB(:,1) = a1';% first column of BBB is transpose of a1
BBB(:,2) = a2';% second column of BBB is transpose of a2

AAA
BBB

%  press the run icon and look at the output
%  in command window
```

```
who     % list all the variable in the workspace
whos % list all variables with their type and storage

%  press the run icon and look at the output
%  in command window

% consider writing code for Matrix multiplication
% in most programming languages
% multiply two matrices (only if column of first matrix
% must match row of second matrix)

szAA = size(AA) %   size of AA
szBB = size(BB);%   size of BB
if (szAA(1,2) == szBB(1,1))% ONLY if column and row match
    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);
                %  add to the variable and replace it
            end     % end of k - loop
        end     % end of j - loop
    end % end of i - loop
end % end if - loop
CC

%  press the run icon and look at the output
%  in command window

% % Note the power of MATLAB derives from its ability to
% % handle matrices very easily

CCC = AA*BB
% % this completes the script session
```

If you have been running the commands as you have been typing the file is already saved. Otherwise 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 the following:

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

1.2.6 Creating a Code Snippet

In this section, we will examine the other type of m-file, which is called the **function *m-file***, which will define a function in MATLAB. For those familiar with other programming languages such as C, Java, or FORTRAN, these files represent functions or subroutines. They are primarily used to handle some specific set 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 the **calling program/code**. There are three essential parameters in developing the *function m-file*:

1. What input is necessary for the calculations (information passed to the function)
2. What are the specific calculations that must take place (code in the function)
3. What information must be returned to the calling program (information returned after processing in the function)

MATLAB requires the structure of the *function m-file* to follow a prescribed format. The variables used in the function files are created when the function is entered and dies when the value is returned from the function. Variables with the same names outside the function are unaffected.

We will use the editor to develop two files: the *script m-file* that will contain the command, which will call the *function m-file*. The *function m-file* will perform a polynomial curvefit. This exercise is called curve fitting. A couple of chapters later this problem will be identified as a problem in **unconstrained optimization**. The *function m-file* will require a set of *x,y data*, together with the *order of the polynomial* to be fit. The technique is based the method of least squares. 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. The responsibility of the *script m-file* is to acquire the information needed, and use the output to compare the curve against the data through a plot, and evaluate the quality of the fit.

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 values returned by the function [*returnval*]. Next, an *equal to* sign is followed by the **name** [*mypolyfit*] of the function, with the parameters **passed to** the function within parenthesis (*XY, N)*. The file is recommended to be saved with the same name as the name of the function [(*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 code:

```
function returnval = mypolyfit(XY, N)
%
% These comments will appear when the user types
% help myployfit in the Command window
% This space is intended to inform the user how to interact
% with the program, what it does and what are the input
% and output parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       mypolyfit performs the Least Square Error
%       fit of polynomial of order N
%
%       x vs y - data found is [:,2] matrix XY
%
%       It returns the vector of N + 1 coefficients
%       starting from the constant term
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Applied Optimization with MATLAB Programming
% P. Venkataraman
% Second Edition,  John Wiley and Sons.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%

for i = 1:N+1
    a(i) = 0.0; % initialize the coefficient to zero
    % this will be returned if there are
    % insufficient data points
end

sz = size(XY);  % find the number of data points
NDATA = sz(1,1);    % number of data points
if NDATA == 0
```

```
    fprintf('Error: There is no data to fit');
    returnval = a;  % zero value returned
    return;     % return back to calling program
end
% the fprintf prints formatted information to the screen

if NDATA < N
    fprintf('Too few data points for good fit');
    fprintf('\nError: Returned without execution');
    returnval = a; % zero value returned
    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
% Please consult any refernce on Numerical Analysis
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for i = 1:N+1;
    % set up the right hand side
    b(i) = 0.0;
    for m = 1:NDATA; % loop over all data points
        b(i) = b(i) + XY(m,2)*XY(m,1)^(i-1);
    end     % loop m

    % set up the normal matrix
    for j = 1:N+1;
        if j >= i
            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.  Otherwise debug the matrix [A]

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

Save the file as *mypolyfit.m*. To use the function we will use the following script file *ScriptPolyfit.m*.

```
%  ScriptPolyfit.m
%
% This is a script file for the running mypolyfit.m
% Chapter 1. Section 1.2.5 Creating a Code Snippet
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Applied Optimization with MATLAB Programming
% P. Venkataraman
% Second Edition,  John Wiley and Sons.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The script will:
%
% create x - y data (a polynomial of order 3)
%
% calls mypolyfit function to obtain coefficients
%
% Compares the original and fitted data on a plot
%
% calculates the error
%
% Also saves the data in an existing directory
%

clear  % clear all values - clean start
clc    % position cursos at the top of the screen
format compact % print single spacing on the screen
               % default is double spacing

% Examine the help available with mypolyfit that you set up
help mypolyfit

% create data points for fitting
for i = 1: 20;
      x = i/20;
      XY(i,1) = x;
      XY(i,2) = 2 + 3.0*x - x*x - 3*x^3;
end

coeff = mypolyfit(XY,3)'  % prints the coefficients on
                          % the screen
% 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

% MATLAB provides a function called polyval that
% will evaluate the polynomial.  However to use it we must
% reverse the coefficient vector.  We cam use the
% MATLAB function "fliplr" on the coefficient
% vector
cflip = fliplr(coeff)   % flip the coefficient vector
yfit = polyval(cflip,XY(:,1));  % values of y at the
                                % same x values

error = sum((yfit - XY(:,2)).^2);
% sum of the square of the difference between
% fitted value and original value
% NOTE: element by element operator .^

% Plot the original data and the fitted data
plot(XY(:,1),XY(:,2),'ro',XY(:,1),yfit,'b-');
% original data is red circles
% fitted data is the blue line
xlabel('x');  % label x axis
ylabel('y');  % label y axis
legend('Original data','fitted data'); % the legend
text(0.1,1.2,strcat('Error squared =',num2str(error)));
% drop a piece of text at the location x = 0.1, y = 1.2
% the text will combine the string'Error squared" and
% the error value after converting it into a string
title('Using function mypolyfit.m');

save C:\OptBook\Chapter1\XY.dat XY -ascii -double
% [] (line continued from above)
% this will save the values of XY in the file XY.dat
% as ascii text file in double precision values in the
% directory
% C:\OptBook\Chapter1\
% Be sure to create the directory before you run this
% script
```

This concludes the exercise where a function was written to calculate the coefficients of the polynomial used to fit a curve to some x-y 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 using data generated by a cubic polynomial so the same coefficients are obtained if the code is correct. A plot is used to compare the results. The error is printed on the plot.

1.2.7 Creating a Program

This section introduces some user interface elements that you can use as part of your programs. This section is optional and can be considered as advanced programming. If you were a C++ programmer you would be doing this after many months of experience. It is included here to illustrate that it is not difficult, even for those starting out programming in MATLAB, to think about incorporating advanced resources available in MATLAB. In this section, we will essentially be performing the same calculations as in the last section, including calling **mypolyfit.m**. We will develop a program that will read ***x-y* data**, curvefit the data using a polynomial, and compare the original and fitted data graphically. 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 then 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. The book will continue to use these elements throughout as appropriate. While the input elements used in this code are new commands, the program will mostly use commands that have been introduced earlier. In sequential order, the events in this program are as follows:

1. To read the ***x-y* 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
6. To report on the fitted accuracy on the figure itself. The new script file will be called **prog_pfit.m**.

Start the text editor to create the file called **prog_pfit.m**. In it, enter the following code:

```
% Program for fitting a polynomial curve to xy data
% Name:  prog_fit.m  ( a script file)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Applied Optimization with MATLAB Programming
% P. Venkataraman
% Second Edition,  John Wiley and Sons.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 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

clear  % clear all values - clean start
clc    % position cursos at the top of the screen
format compact % print single spacing on the screen
               % default is double spacing

[file,path] =uigetfile('*.dat','All Files');
% 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
% open the file saved in the last session

if isstr(file) % if a file is selected
    loadpathfile = ['load ',path file];
    % creates a string that will be evaluated using eval
    % 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 -
    % It executes Matlab load command. 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)
    % XY matrix is avaiable in the workspace

    NDATA = length(XY(:,1));   % number of data points
    clear path loadpathfile
    % we will not need these variables
    % get rid of these variables to free memory
end  % if statement

% 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

% lets get the value of order of polynomial to
% be fitted from the user
NS = inputdlg(PROMPT, TITLE, LINENO);
% the input dialog captures the user input in NS
% NS is a string Array
% check help input dialog for more information

N = str2num(NS{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(XY,N);

% generate the fitted curve and obtain the squared error
% Here is another (inefficient) way to evaluate
% the y values of fit and the error
err2 = 0.0;
for i = 1:NDATA   % for each data point
    for j = 1:N + 1
        a(1,j) = XY(i,1)^(j-1);
    end
    y(i) = a*coeff;   % the data for the fitted curve
    err2 = err2 + (XY(i,2) -y(i))*(XY(i,2)-y(i));
    % the square error
end

% plotting
plot (XY(:,1),XY(:,2),'ro',XY(:,1),y,'b-');
% original data are red o's
% fitted data is blue solid line
xlabel('x');
ylabel('y');

% create strings using available information
% for the title
strorder = setstr(num2str(N));
% convert the order of curve to a string
```

```
% setstr assigns the string to strorder
titlestr = ['Polynomial curvefit of 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');
% you should see the same plot as in the previous exercise

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 NS coeff XY file err2

% 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, require interactions, and display a figure similar to Figure 1.10. 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 the ability to debug and troubleshoot. Although this section has maintained a separate section on the use of MATLAB in different ways, subsequent chapters will see an integration of the use of various resources that MATLAB contains.

1.2.8 Application Bibliography

The following listing is a small sample of titles that explore in greater detail, the spread of the application of optimization to new areas. All of books appeared in the period between the two editions of this book.

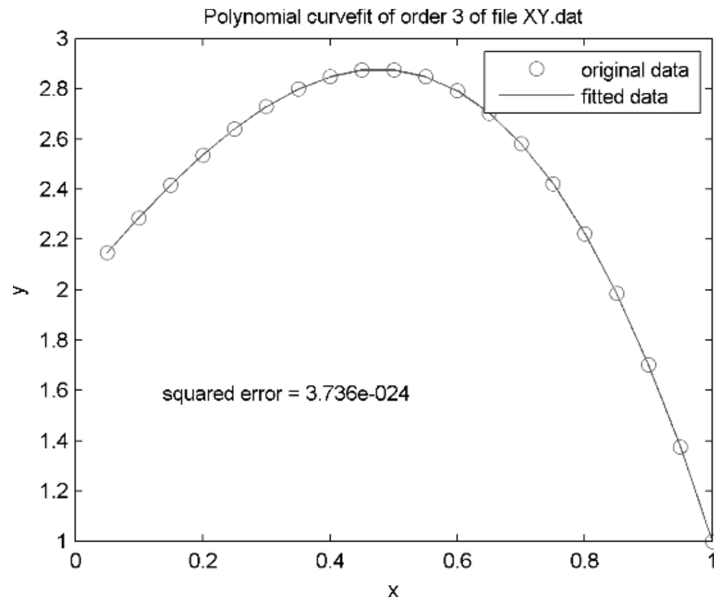

Figure 1.10 Original and fitted data.

Allstot, David James, Kiyong Choi, Lenn Schramm, and Jinho Park, *Parasitic-Aware Optimization of CMOS RF Circuits*. New York: Springer, 2003.

Baldick, Ross, *Applied Optimization: Formulation and Algorithms for Engineering Systems*. London: Cambridge University Press, 2006, 786 pages.

Bellingham, Richard and Russell J. Campanello, "HR Optimization: From Personnel Administration to Human and Organizational Capital Development," *HRD Products*, 2004.

Biegler, Lorentz T., *Large-Scale Pde-Constrained Optimization*. New York: Springer, 2003.

Craven, Bruce Desmond, M. Sardar, and N. Islam, *Optimization in Economics and Finance: Some Advances in Nonlinear, Dynamic, Multicriteria and Stochastic Methods*. New York: Springer, 2005.

Dorigo, Marco and Thomas Stutzle, *Ant Colony Optimization*. Boston: MIT Press, 2004.

Du, Ding-Zhu, *Combinatorial Optimization in Communication Networks*. New York: Springer, 2006.

Fullér, Róbert, and Christer Carlsson, *Fuzzy Reasoning in Decision Making and Optimization*. New York: Springer, 2002.

Hartmann, Alexander K. and Rieger Heiko, *New Optimization Algorithms in Physics*. Hoboken, NJ: Wiley-VCH, 2004.

Laporte, Emmanuel and Patrick Le Tallec, *Numerical Methods in Sensitivity Analysis and Shape Optimization*. New York: Springer, 2003, 216 pages.

Leondes, Cornelius T., "Methods in Diagnosis Optimization," *World Scientific*. 2005.

Lu, Bing, Sachin S. Sapatnekar, and Dingzhu Du, *Layout Optimization in VLSI Design*. New York: Springer, 2001.

Momoh, James A., *Electric Power System Applications of Optimization*. Marcel Dekker, 2001.

Narayan, S. Rau, *Optimization Principles: Practical Applications to the Operation and Markets of the Electric. Power Industry*. Hoboken, NJ: Wiley-IEEE, 2003, 360 pages

Pardalos, Panos M. and Christodoulos A. Floudas, *Optimization in Computational Chemistry and Molecular Biology: Local and Global Approaches*. New York: Springer, 2000.

Putman, Richard E., *Industrial Energy Systems: Analysis, Optimization, and Control*, ASME Press, 2004.

Ralf Korn, Elke, *Option Pricing and Portfolio Optimization: Modern Methods of Financial Mathematics*. American Mathematical Society, 2001.

Rhyder, Robert F., *Manufacturing Process Design and Optimization*. Marcel Dekker, 1997.

Torres, Nāestor V. and Eberhard O. Voit, *Pathway Analysis and Optimization in Metabolic Engineering*, London: Cambridge University Press, 2002.

Vofs, Stefan, Stefan Voss, and David L. Woodruff, *Introduction to Computational Optimization Models for a Production Planning in a Supply Chain*. New York: Springer, 2006.

PROBLEMS

Many problems here were defined by students in my course on optimization as part of their projects. **(For problems 1.1 to 1.5 please use your imagination for domain knowledge.)**

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. Assume a single beam in the middle of the basement parallel to the long side of the house.

1.7 Define the mathematical model for an optimum overhanging traffic light. Decide what you want to optimize. Define your parameters. (See Figure Problem 1.7.)

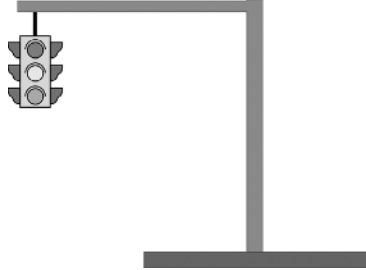

Figure Problem 1.7

1.8 Define the problem and identify a mathematical model for scheduling and optimization of your daily routine activity. Plan around six routine activities. Associate each with cost. Identify a couple of constraints. Keep the model linear.

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 based on ingredients of limited availability to meet specified demands.

1.11 Find the rectangle of the largest area that can fit within an ellipse of semi-major axis a and semi-minor axis b. Set up the optimization problem, Make any assumptions you need. Refer to Figure Problem 1.11.

1.12 Find the circle of the largest area in a semi-ellipse (right of the y-axis) with semi-major axis a and semi-minor axis b. Set up the optimization problem, Make any assumptions you need (Figure Problem 1.12.)

1.13 Find the quadratic polynomial between the limits $x = 0$ and $x = 3$, enclosing minimum area with x-axis, with a slope of 3 at $x = 0$ and a slope of -1 at $x = 3$. Set up the optimization problem.

1.14 Consider a beam with given material properties (E and I), known cross-section area (A), and of length L. Design the triangular load distribution spread over the entire length of the beam, with the total

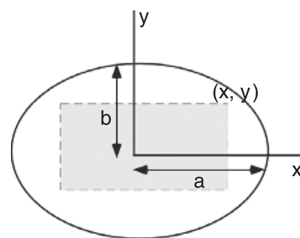

Figure Problem 1.11

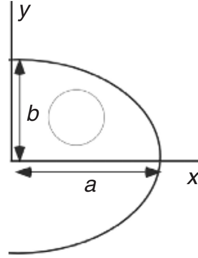

Figure Problem 1.12

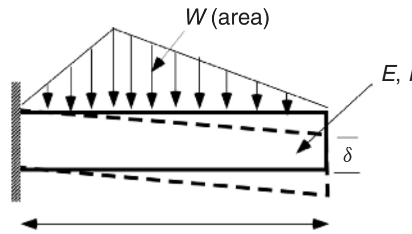

Figure Problem 1.14

load of W, which will cause the least deflection of the end. (See Figure Problem 1.14.)

1.15 The energy crisis of 2008 is forcing the gas company to consider ways to provide short time relief to consumers by trying to use the same infrastructure, with minor modification like a new pump, to increase the flow of gas from the wells. The gas flows from the wells to the utilities through large pipes with pumping stations placed at equidistant locations (L). The pressure difference between two pumping stations can be considered to be $p_1 - p_2$. Figure (Problem 1.15) illustrates the layout and the cross-section description. The volume flow rate Q (cubic feet/hr) can be calculated as:

$$Q = 1350D^{2.5}\left(\frac{p_1 - p_2}{LG}\right)^{0.5}$$

Where D is the diameter of the pipe; G is the specific gravity of the gas (dimensionless); L is the pipe length (yards); p_1 is the inlet pressure (in of H_2O); p_2 is the outlet pressure (in of H_2O); This simple equation is accredited to Dr. Pole in 1851 (Ref: www.psig.org/papers/2000/0012.pdf). However, for any cross section of the flow/pipe (see figure), if the outside pressure is ignored, the tangential and radial stresses are calculated

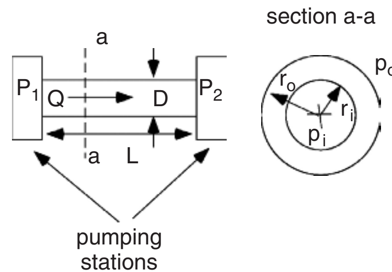

Figure Problem 1.15

(Shigley, Mischke, Budynas, *Mechanical Engineering Design*) as:

$$\sigma_t = \frac{p_i r_i^2}{r_o^2 - r_i^2}\left(1 + \frac{r_o^2}{r^2}\right)$$

$$\sigma_r = \frac{p_i r_i^2}{r_o^2 - r_i^2}\left(1 - \frac{r_o^2}{r^2}\right)$$

Note that $\boldsymbol{\sigma_t > \sigma_r}$ and $\boldsymbol{\sigma_t}$ is maximum at $\boldsymbol{r_i}$, set up an optimization problem for increasing the flow rate. Identify parameters for the problem based on existing infrastructure (use the library or internet) and make suitable assumptions.

1.16 A thin-walled spherical gas container (radius: $\boldsymbol{r}$, thickness: $\boldsymbol{t}$) is required to be designed for an internal pressure of 5 MPa. Find the smallest mass of the container if the factor of safety is 3 and the material used is structural steel. The stresses in the thin-walled structure can be evaluated as follows:

$$\sigma(normal) = \frac{pr}{2t};\ \tau(shear) = \frac{pr}{4t}$$

Set up the optimization problem in standard format.

1.17 A closed-end, thin-walled cylindrical pressure vessel is to be designed for minimum mass. It should contain at least 25 m^3 of gas at a pressure of 3.5 MPa. The circumferential/hoop stress is not to exceed 210 MPa, while the circumferential strain is not to exceed 0.001. The material is structural steel. The design variables are inside radius $\boldsymbol{r}$ and thickness $\boldsymbol{t}$. Draw the figure expressing the problem. Take the maximum hoop stress as follows:

$$\sigma_t|_{\max} = \frac{p(2r + t)}{2t}$$

Set up the optimization problem in standard format.

1.18 A chrome vanadium spring ($G = 79.3$ GPa; $\rho = 7{,}860$ kg/m^3), of minimum mass is needed to carry a uniform pressure load $p = 250$ kPa acting over a diaphragm of diameter of 0.1 m as shown in Figure Problem 1.18 giving a compressive load of $\boldsymbol{F}$. The maximum shearing stress is 1.5 GPa. The maximum compression of the spring is 0.1 m. The surge frequency should be 60 Hz or higher to avoid resonance. The solid length of the spring (compressed fully) must be larger than $L = 40$ mm because of the block. The spring is fully compressed when carrying the load. The design variables are $\boldsymbol{n}$ the number of coils; $\boldsymbol{d}$ the spring wire diameter, $\boldsymbol{D}$ the mean diameter of the spring. The outer limits on the spring are 0.1m. We define three relations before assembling the constraints.

$$C = \frac{D}{d}; \quad \text{spring index}$$

$$k = \frac{Gd^4}{8D^3(n-2)}; \quad \text{spring constant}$$

$$K = \frac{(4C+2)}{(4C-3)}; \quad \text{Bergsträsser factor}$$

The constraint equations are

$$nd \geq 0.04; \quad d + D \leq 0.1$$

$$\frac{KF^2D}{\pi kd^3} \leq 1.5 \text{ GPa (shear stress)}$$

$$\frac{F^2D^3(n-2)}{kd^4G} \leq 0.1\,m \text{ (maximum compression)}$$

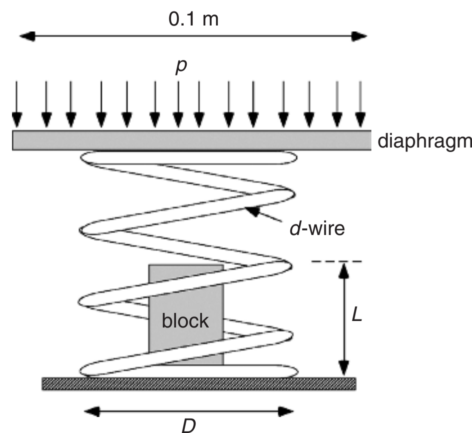

Figure Problem 1.18

$$\frac{d}{2\pi D^2(n-2)}\sqrt{\frac{G}{2\rho}} \geq 60 \text{ Hz; (surge frequency)}$$

And the side constraints:

$$4 \leq n \leq 12; \quad 0.06 \leq D \leq 0.09[m]; \quad 0.001 \leq d \leq 0.012[m]$$

Set up the optimization problem in the standard format.

1.19 Bellville springs, or conical disc springs, can handle large loads in tight spaces. Linear, regressive, and progressive load-deflection characteristics can be designed through stacking (*SAE Spring Design Manual*, Transactions SAE, 1990). They have nonlinear load-deflection and load-stress characteristics. A single spring is used in the following problem and the geometry is defined in the Figure (Problem 1.19). There are four design variables: $\boldsymbol{D_i}$, $\boldsymbol{D_o}$, $\boldsymbol{h}$, $\boldsymbol{t}$. The assumptions in the following development are: the angle α is small; cross-section do not warp throughout the range of deflection; fully elastic behavior during deflection; maximum stress at the inside joint as shown in the Figure Problem 1.19 (Almen and Lazlo, *The Uniform—Section Disc*, Transactions of ASME, 1936). The functions involved in the problem are:

$$W = \rho\pi t\left(\frac{D_o + D_i}{2}\right)\sqrt{h^2 + \left(\frac{D_o - D_i}{2}\right)^2} \quad \textit{(Weight)}$$

Almen and Lazlo define the load-deflection and stress-deflection formulas for the spring as

$$P = \frac{Ey}{(1-\nu^2)M(D_o/2)^2}[(h - y/2)(h - y)t + t^3]$$

$$S = \frac{Eh}{(1-\nu^2)M(D_o/2)^2}[C_1(h/2) + C_2 t]$$

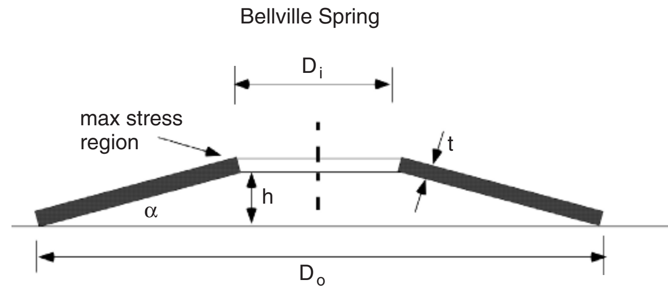

Figure Problem 1.19 Bellville spring optimization.

$\boldsymbol{P}$ is the axial load (lb); $\boldsymbol{y}$ is the deflection (inches); $\boldsymbol{E}$ is the modulus of elasticity (psi); $\boldsymbol{S}$ is the maximum tensile stress (psi); $\boldsymbol{\rho}$ is the specif weinght (lb/in^3); $\boldsymbol{\nu}$ is the Poisson's ratio. The values for $\boldsymbol{M}$, $\boldsymbol{C}_1$, $\boldsymbol{C}_2$ can be calculated by

$$M = \frac{6.0}{\pi \ln \dfrac{D_o}{D_i}} \left[\frac{D_o/D_i - 1}{D_o/D_i} \right]$$

$$C_1 = \frac{6.0}{\pi \ln \dfrac{D_o}{D_i}} \left[\frac{D_o/D_i - 1}{\ln(D_o/D_i)} - 1 \right]$$

$$C_2 = \frac{6.0}{\pi \ln \dfrac{D_o}{D_i}} \left[\frac{D_o/D_i - 1}{2} \right]$$

Set up an optimization problem for minimum weight, with a capacity of generating an axial load of at least 200 lb for a deflection of 0.08 in. The stress must be less than the tensile stress of the material. Take the factor of safety as 2.2. The range of $\boldsymbol{D}_o$ is between 3 in and 10 in. Use appropriate side constraints for the other variables. Choose a material too.

1.20 This is a variation on Example 1.2. Rectangular planks of any cross-section dimensions a and b can be nailed to construct an I-beam like that in Figure Problem 1.20. The nail spacing is 4 in. The wood is Douglas fir ($E = 1.6\ 10^6$ psi, $G = 0.6\ 10^6$ psi, $\nu = 0.33$, $\gamma = 0.016$ ib/in^3). Find the allowable tensile and shear stress for design. Consider the load of 500 lb being carried at the end of 48 in. It is cantilevered at the other end. Design the beam of minimum weight. The maximum load is at the fixed end. The shear in the nails is limited to 100 lb. Develop the mathematical model. You can follow Example 1.2, except that the shear stress constraint is replaced by the shear in the nail. The shear force in the nail is the shear flow at the junction, multiplied by the nail spacing.

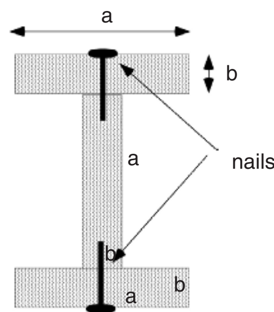

Figure Problem 1.20

1.21 What is the smallest gear ratio ($\boldsymbol{r_C/r_B}$) for the transmission shafts in Figure Problem 1.21 if shear stress in AB and AC are less than 1,200 psi, and the twist of the end D must be less than 0.16 rad. The length of the shaft AB is 12 inches and its diameter is 0.25 in. The length of the shaft CD is 24 inches and its diameter is 0.12 in. Use your own additional information if required.

1.22 In July, Venkat's Fruit Orchard usually sees a scheduling problem. Both cherries and blueberries ripen around the same time. A cartload of cherries sells for \$2,750 while the cartload of blue berries fetches \$1,750. The farm must deliver at least a half cartload of cherries and one cartload of blueberries to the local supermarket. Three persons can pick two cartloads of cherries in a day while one person will pick a cartload of blueberries in the same time. Only 46 people show up for work during the day. One cartload of cherries is contained in 4 pallets, while the same amount of blueberries occupies 6 pallets. There are 251 pallets. Each picker is paid \$45 per day. The packaging and storage can handle, at most, 30 cartloads per day. Set up the LP problem for the Orchard to maximize profits.

1.23 The energy crisis of 2008 is keeping the refineries very active trying to adjust the product mix due to demand and supply. Largely, the decision is about buying two varieties of crude oil and selling two kinds of product: gasoline and diesel. They can buy light crude at \$135 per barrel, while heavy crude is discounted at \$95 per barrel. Consider one barrel as 40 gallons. They can sell gasoline at \$4.25/gallon and diesel at \$4.85 a gallon. A barrel of light crude will yield 0.5 barrels of gasoline and 0.4 barrels of diesel. A barrel of heavy crude will yield 0.4 barrels of gasoline and 0.2 barrels of diesel. Totals supply of light crude is limited to 500,000 barrels, while heavy crude is unlimited in supply. In a day the refinery can only handle 2 million gallons of gasoline and 3 million gallons of diesel. Set up the LP programming problem to maximize daily profits.

1.24 Consider the standard bridge circuit shown in Figure Problem 1.24. The power delivered by the batteries is $\boldsymbol{P_b}$ and the power consumed by the resistors is $\boldsymbol{P_r}$. Matching the two sets up the constraint

$$P_b - P_r = 0$$

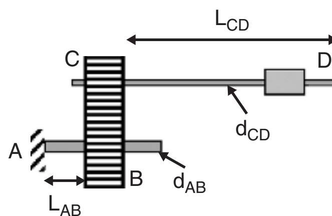

Figure Problem 1.21

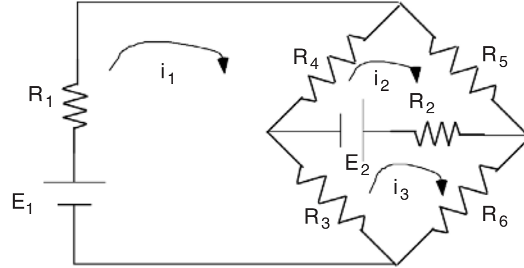

Figure Problem 1.24 A bridge circuit.

Show that the solution to the optimization problem of Minimizing $\boldsymbol{P}_r$ with respect to the currents $(\boldsymbol{i}_1, \boldsymbol{i}_2, \boldsymbol{i}_3)$ and subject to the constraint will yield the Kirchhoff's loop equation (Donald A. Pierre, *Optimization Theory with Applications*). The expression for the power is the following:

$$P_b = E_1 i_1 + E_2(i_3 - i_2)$$

$$P_r = i_1^2 R_1 + (i_2 - i_3)^2 R_2 + (i_1 - i_3)^2 R_3 + (i_1 - i_2)^2 R_4 + i_2^2 R_5 + i_3^2 R_6$$

1.25 A gating circuit using a switch and two diodes is shown in Figure Problem 1.25. When the switch $(\boldsymbol{S}_1)$ is open the load current $\boldsymbol{i}_L$ flows through the load $\boldsymbol{R}_L$. When the switched is closed the power $(\boldsymbol{P}_R)$ is dissipated through the resistor $\boldsymbol{R}$. It is necessary to limit this power dissipation to $\boldsymbol{P}_m$. The current and the dissipated power (Donald A. Pierre, *Optimization Theory with Applications*) are:

$$i_L = \frac{V}{R + R_L}; \quad P_R = \frac{V^2}{R}$$

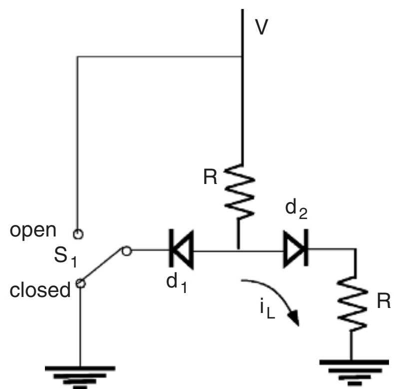

Figure Problem 1.25 A gate circuit.

Set up an optimization problem to maximize the load current with respect to $\boldsymbol{R}$ and $\boldsymbol{V}$ subject to the constraint on the dissipated power.

1.26 A clipping circuit with the given input-output characteristics is shown in the Figure Problem 1.26. For specified levels of $\boldsymbol{V}_{o1}$, $\boldsymbol{V}_{o2}$, *and* $\boldsymbol{R}_L$ the resistances $\boldsymbol{R}_1$, $\boldsymbol{R}_2$, and the voltage $\boldsymbol{E}$ are to be selected so that the desired input-output characteristics must be obtained. Furthermore, the power drain ($\boldsymbol{P}$) from the voltage supply ($\boldsymbol{E}$) must be as low as possible (Donald A. Pierre, *Optimization Theory with Applications*). The functional relations are:

$$P = \frac{(E - V_{o1})^2}{R_2} + \frac{V_{o1}^2}{R_1} + \frac{V_{o1}^2}{R_L}$$

$$V_{o1} = R_1 \left[\frac{E - V_{o1}}{R_2} - \frac{V_{o1}}{R_L} \right]; \quad V_{o2} = \frac{(E - V_{o2}) R_L}{R_2}$$

Obtain the relations above. Set up the optimization problem in standard format.

1.27 The ratio of the $|\boldsymbol{V}_2(\boldsymbol{s})/\boldsymbol{V}_1(\boldsymbol{s})|$ needs to be maximized for the circuit shown in the Figure Problem 1.27. Note that $\boldsymbol{s} = \boldsymbol{j\omega}$. However, the ratio of the voltages must match at two given frequencies. Hence the constraint:

$$\left| \frac{V_2(j\omega_1)}{V_1(j\omega_1)} \right| = \left| \frac{V_2(j\omega_2)}{V_1(j\omega_2)} \right|$$

must be satisfied. The resistances are fixed while the capacitors and inductors are variables. Set up the constrained optimization problem and express the voltage ratio as a real valued function.

1.28 Consider a plane in level flight where lift ($\boldsymbol{L}$) is equal to weight ($\boldsymbol{W}$) and thrust ($\boldsymbol{T}$) is equal drag ($\boldsymbol{D}$). The forces are shown in Figure Problem 1.28. The lift and drag can also be calculated from the nondimensional lift and

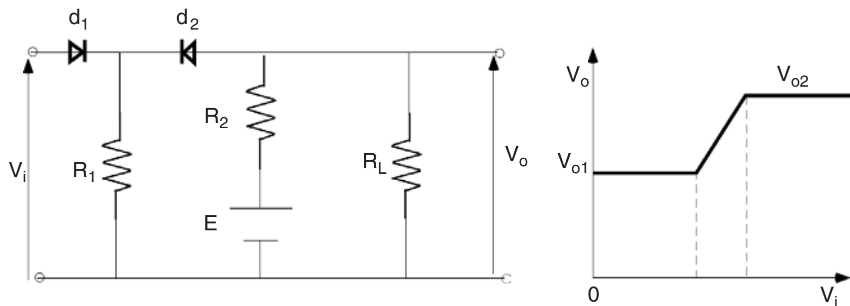

Figure Problem 1.26 A clipping circuit.

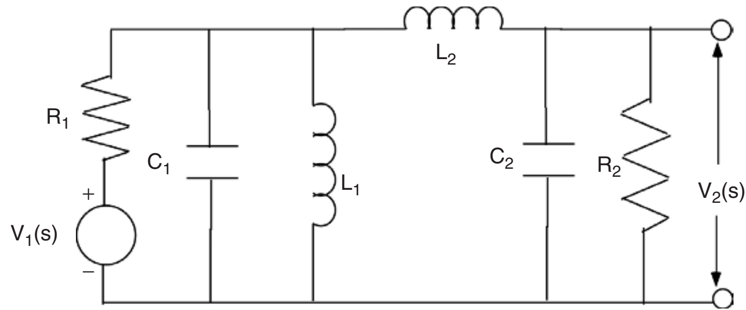

Figure Problem 1.27 Matching transfer function.

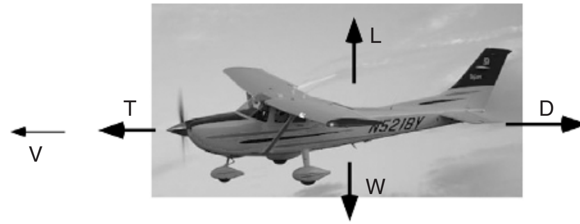

Figure Problem 1.28 Aircraft in level flight.

drag coefficients ($\boldsymbol{C_L}$ ***and*** $\boldsymbol{C_D}$) through

$$L = \frac{1}{2}\rho V^2 S C_L; \quad D = \frac{1}{2}\rho V^2 S C_D$$

In standard design, the relation between the coefficients can be expressed through the parabolic drag polar:

$$C_D = C_{D0} + K C_L^2$$

where $\boldsymbol{\rho}$ is the atmospheric density at the flight altitude; $\boldsymbol{V}$ is the speed of flight; $\boldsymbol{S}$ is the reference area of the wing; $\boldsymbol{C_{D0}}$ is the zero lift drag; and $\boldsymbol{K}$ is induced drag factor, all of which are constants for the flight. Find the speed for minimum drag. See Figure Problem 1.28.

1.29 Since the shown aircraft is powered by a piston propeller engine, maybe it is better to fly at the speed for minimum power, where the power is calculated as

$$P = DV$$

Find the speed for minimum power.

1.30 Another way of looking at Problem 1.28 is to work with the nondimensional relations. A measure of aircraft performance is aerodynamic efficiency $\boldsymbol{E}$, which is the ratio of $\boldsymbol{C_L}$ over $\boldsymbol{C_D}$. Find the maximum aerodynamic efficiency and the speed corresponding to this efficiency.

Problems 1.31 to 1.33 can be handled by assuming a series solution as in Example 1.4, or using Bezier functions (Chapter 9, Chapter 11), or any other type of parametric representation for the variables. These problems are more naturally defined through the calculus of variation. These problems can be postponed until those chapters if only Bezier functions are going to be used.

1.31 The problem involves designing a two-dimensional wing for minimum drag (Miele, *Theory of Optimum Aerodynamic Shapes*). The linearized theory for a 2D symmetrical wing, at zero angle of attack, at a given free stream Mach number ($\boldsymbol{M}$) leads to the pressure coefficient described by

$$C_p = \frac{2\dfrac{dy}{dx}}{\sqrt{M^2 - 1}}$$

The function $\boldsymbol{y(x)}$ is the description of the airfoil upper surface, as shown in Figure Problem 1.31. The aerodynamic drag, for a given free stream dynamic pressure ($\boldsymbol{q}$) can then be expressed as follows:

$$D = 2q \int_{x_i}^{x_f} C_p \frac{dy}{dx} dx = \frac{4q}{\sqrt{M^2 - 1}} \int_{x_i}^{x_f} \left(\frac{dy}{dx}\right)^2 dx$$

From the figure

$$x_i = 0; \quad x_f = L; \quad y_i = 0; \quad y_f = 0$$

Consider that the airfoil volume ($\boldsymbol{A}$) is given, leading to the constraint

$$\frac{A}{2} = \int_{x_i}^{x_f} y(x)\, dx$$

Figure Problem 1.31 Minimum pressure drag.

If the torsional stiffness ($\boldsymbol{I}_c$) of the thin-skin structure is also given, then

$$\frac{I_c}{2} = \int_{x_i}^{x_f} y^2(x)\, dx$$

If the bending stiffness is prescribed ($\boldsymbol{I}_b$) is prescribed then

$$\frac{3I_b}{2} = \int_{x_i}^{x_f} y^3(x)\, dx$$

Express the surface as a Bezier function and identify the optimization problem for minimum drag subject to

a. given volume
b. given torsional stiffness
c. given bending stiffness
d. given volume and torsional stiffness
e. given volume and bending stiffness

Note that this problem suggests that using a parametric definition for the function/trajectory we can convert problem from the calculus of variations to design optimization problems.

1.32 Here, we set up a problem in designing a body of revolution for minimum drag under the Newtonian pressure law as illustrated in the Figure Problem 1.32. The Newtonian pressure law in hypersonic aerodynamics is (Miele, *Theory of Optimum Aerodynamic Shapes*)

$$C_p = 2\sin^2\theta$$

where θ is the local inclination of the body shape. Nevertheless, the functional to be minimized is

$$I = \int_{x_i}^{x_f} \frac{y(y')^3}{1+(y')^2} dx + \frac{y_i^2}{2}; \quad x_i = 0; \quad x_f = L; \quad y_f = d/2$$

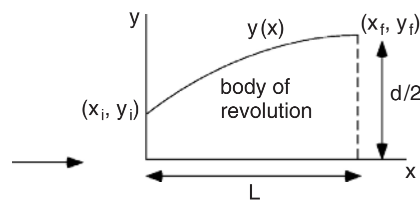

Figure Problem 1.32 Minimum pressure drag for body of revolution.

Express the surface as a Bezier function and identify the optimization problem for minimum drag.

1.33 Consider the RL circuit shown in Figure Problem 1.33. The initial time is 0. The final time T is specified. It is required to maximize the output voltage at the final time. There are two constraints. The first is the differential constraint due to the circuit equation. The second is that the energy delivered over the time interval $\boldsymbol{T}$ is specified as $\boldsymbol{E_s}$. The following functional equations can be easily derived:

$$v_o(T) = \int_0^T R\frac{di}{dt}dt; \quad \frac{di}{dt} = \frac{v_i}{L} - \frac{R}{L}i; \quad E_s = \int_0^T v_i i\, dt$$

For a fixed $\boldsymbol{R}$ and $\boldsymbol{L}$, find the functions $\boldsymbol{v_i(t)}$ and $\boldsymbol{i(t)}$ that will solve the optimization problem. Represent $\boldsymbol{v_i(t)}$ and $\boldsymbol{i(t)}$ as Bezier functions.

1.34 The $\Phi X \Psi$ fraternity is going to capitalize on the mouth-watering pizza sauce, accidentally created by one of its members, by organizing a weekly fundraiser for its charities. It plans two varieties of pizza based on two ingredients, sausage and cheese. Of course, each pizza will have the pie, base sauce, sausage, and cheese. The ingredients are in cups for each type of pizza. A cup of sausage costs \$0.75, a cup of cheese \$0.5, while a cup of sauce is \$1.00. The pie costs \$0.50 each.

Pizza	Sell Price	Sausage	Cheese	Sauce
Supergooey	\$ 7.50	1.5	2.75	2.0
Justgooey	\$ 7.50	2.5	1.5	1.5

The fraternity can obtain only 120 cups of sausage, 100 cups of cheese, and 120 cups of sauce, and any amount of pies. Find how many pizzas of the two kinds it must sell for maximum profit.

1.35 A two-dimensional aluminum truss is shown in Figure Problem 1.35. All of the members have the same annular cross-section. For the loading shown

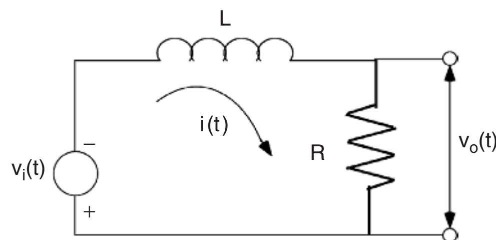

Figure Problem 1.33 A circuit problem.

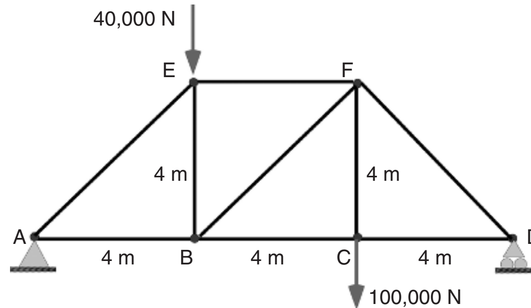

Figure Problem 1.35 Truss Problem.

determine the truss of minimum mass. Ensure that the normal stresses are within the elastic limit, while the truss does not fail in buckling.

1.36 A shell and tube heat exchanger is a popular device to transfer energy. The cross-section of one is shown in Figure Problem 1.36. The diameter of the shell (***D***) is specified as 0.75 m. The length of the exchanger (***L***) is 2.5 m. The shell is made from a plate that costs \$9.50 per unit area. The tubes are cut from stock and cost \$1.25 per meter. The energy exchanged by the device can be calculated as $\mathbf{0.5}*\boldsymbol{sqrt}(\boldsymbol{n})*\boldsymbol{L}^{0.75}/\boldsymbol{d}^{0.2}$ kW. Here ***n*** is the number of tubes and ***d*** is the diameter of the tube. To make sure the tubes fit in the shell we impose an area constraint that the total area of the tubes must be less than half the cross-sectional area of the shell. Just to be sure, we constrain the diameter by requiring $\boldsymbol{sqrt}(\boldsymbol{n})*\boldsymbol{d}$ must be less than the diameter of the shell. Set up an optimization problem for minimum cost with the understanding that the heat exchanger must transfer at least 10 kW of energy.

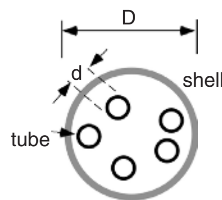

Figure Problem 1.36 Heat exchange design.

1.37 Fred wants to invest his \$100,000 in four mutual funds whose holdings and return is shown in the Table Problem 1.37. He wants his choice to reflect a growth strategy that requires that he has at least 60 percent invested in domestic stocks, at most 20 percent in international stock, and wants his exposure to real estate to be at most 10 percent. How should he invest his money for maximum return?

Table Problem 1.37

Mutual Fund	Domestic Stocks	International Stocks	Bonds	Real Estate	Expected Return
F1	0.2	0.6	0.2	0.0	6 %
F2	0.6	0.1	0.3	0.1	3 %
F3	0.4	0.5	0.0	0.1	4 %
F4	0.5	0.1	0.1	0.3	1 %

1.38 The four-bar link is shown in Figure Problem 1.38. Find the link length $\boldsymbol{L}_1$ and $\boldsymbol{L}_2$ to maximize the velocity at D when θ is 90° and ω is 50 rad/s. Use reasonable side constraints.

1.39 Solve Problem 1.38 with a constraint on acceleration while the angular velocity ω is increasing at a rate of 2 rad/sec^2.

1.40 The local Pizza Shoppe wants to create a template it can use to cut three different sizes of the pie base each time it rolls out the rectangular area of the dough as shown in Figure Problem 1.40. The Shoppe is experimenting with metric pizzas to combat obesity. The three sizes are grande ($\boldsymbol{R}_1 =$ 0.2 m), medium ($\boldsymbol{R}_2 = 0.15$ m), and personal ($\boldsymbol{R}_3 = 0.1$ m). Set up an optimization problem for locating the three bases (one each) such that there is the least wastage.

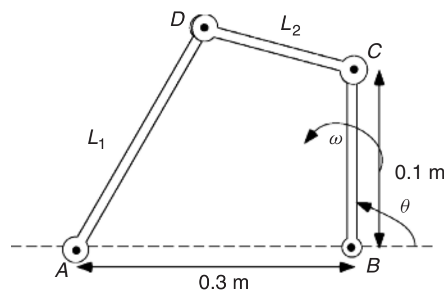

Figure Problem 1.38 Four bar linkage.

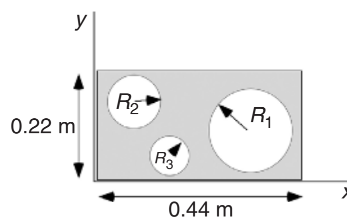

Figure Problem 1.40 Pizza base problem.