

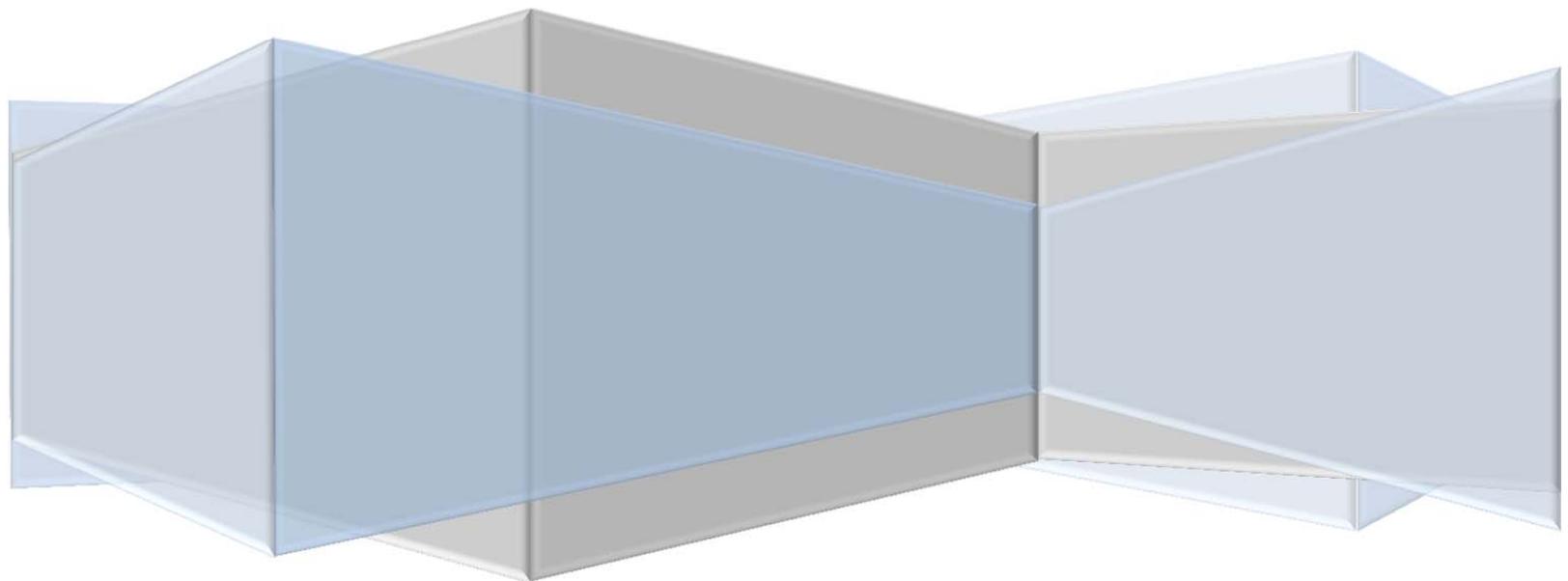
Optimization Methods for Engineering Design

Applications and Theory

Alan R. Parkinson

Richard J. Balling

John D. Hedengren



CHAPTER 1

INTRODUCTION TO OPTIMIZATION-BASED DESIGN

1. What is Optimization?

Engineering is a profession whereby principles of nature are applied to build useful objects. A mechanical engineer designs a new engine, or a car suspension or a robot. A civil engineer designs a bridge or a building. A chemical engineer designs a distillation tower or a chemical process. An electrical engineer designs a computer or an integrated circuit.

For many reasons, not the least of which is the competitive marketplace, an engineer might not only be interested in a design which works at some sort of nominal level, but is the *best* design in some way. The process of determining the best design is called *optimization*. Thus we may wish to design the smallest heat exchanger that accomplishes the desired heat transfer, or we may wish to design the lowest-cost bridge for the site, or we may wish to maximize the load a robot can lift.

Often engineering optimization is done implicitly. Using a combination of judgment, experience, modeling, opinions of others, etc. the engineer makes design decisions which, he or she hopes, lead to an optimal design. Some engineers are very good at this. However, if there are many variables to be adjusted with several conflicting objectives and/or constraints, this type of experience-based optimization can fall short of identifying the optimum design. The interactions are too complex and the variables too numerous to intuitively determine the optimum design.

In this text we discuss a computer-based approach to design optimization. With this approach, we use the computer to search for the best design according to criteria that we specify. The computer's enormous processing power allows us to evaluate many more design combinations than we could do manually. Further, we employ sophisticated algorithms that enable the computer to efficiently search for the optimum. Often we start the algorithms from the best design we have based on experience and intuition. We can then see if any improvement can be made.

In order to employ this type of optimization, several qualifications must be met. First, we must have a *quantitative model* available to compute the responses of interest. If we wish to maximize heat transfer, we must be able to calculate heat transfer for different design configurations. If we wish to minimize cost, we must be able to calculate cost. Sometimes obtaining such quantitative models is not easy. *Obtaining a valid, accurate model of the design problem is the most important step in optimization.* It is not uncommon for 90% of the effort in optimizing a design to be spent on developing and validating the quantitative model. Once a good model is obtained, optimization results can often be realized quickly.

Fortunately, in engineering we often do have good, predictive models (or at least partial models) for a design problem. For example, we have models to predict the heat transfer or pressure drop in an exchanger. We have models to predict stresses and deflections in a bridge. Although engineering models are usually physical in nature (based on physical

principles), we can also use empirical models (based on the results of experiments). It is also perfectly acceptable for models to be solved numerically (using, for example, the finite element method).

Besides a model, we must have some variables which are free to be adjusted—whose values can be set, within reason, by the designer. We will refer to these variables as *design variables*. In the case of a heat exchanger, the design variables might be the number of tubes, number of shells, the tube diameters, tube lengths, etc. Sometimes we also refer to the number of design variables as the *degrees of freedom* of the computer model.

The freedom we have to change the design variables leads to the concept of *design space*. If we have four design variables, then we have a four dimensional design space we can search to find the best design. Although humans typically have difficulty comprehending spaces which are more than three dimensional, computers have no problem searching higher order spaces. In some cases, problems with thousands of variables have been solved.

Besides design variables, we must also have criteria we wish to optimize. These criteria take two forms: *objectives* and *constraints*. Objectives represent goals we wish to maximize or minimize. Constraints represent limits we must stay within, if inequality constraints, or, in the case of equality constraints, target values we must satisfy. Collectively we call the objectives and constraints *design functions*.

Once we have developed a good computer-based analysis model, we must link the model to optimization software. Optimization methods are somewhat generic in nature in that many methods work for wide variety of problems. After the connection has been made such that the optimization software can “talk” to the engineering model, we specify the set of design variables and objectives and constraints. Optimization can then begin; the optimization software will call the model many times (sometimes thousands of times) as it searches for an optimum design.

Usually we are not satisfied with just one optimum—rather we wish to explore the design space. We often do this by changing the set of design variables and design functions and re-optimizing to see how the design changes. Instead of minimizing weight, for example, with a constraint on stress, we may wish to minimize stress with a constraint on weight. By exploring in this fashion, we can gain insight into the trade-offs and interactions that govern the design problem.

In summary, *computer-based optimization* refers to using computer algorithms to search the design space of a computer model. The design variables are adjusted by an algorithm in order to achieve objectives and satisfy constraints. Many of these concepts will be explained in further detail in the following sections.

2. Engineering Models in Optimization

2.1. Analysis Variables and Functions

As mentioned, engineering models play a key role in engineering optimization. In this section we will discuss some further aspects of engineering models. We refer to engineering models as *analysis models*.

In a very general sense, analysis models can be viewed as shown in Fig 1.1 below. A model requires some inputs in order to make calculations. These inputs are called *analysis variables*. Analysis variables include design variables (the variables we can change) plus other quantities such as material properties, boundary conditions, etc. which typically would not be design variables. When all values for all the analysis variables have been set, the analysis model can be evaluated. The analysis model computes outputs called *analysis functions*. These functions represent what we need to determine the “goodness” of a design. For example, analysis functions might be stresses, deflections, cost, efficiency, heat transfer, pressure drop, etc. It is from the analysis functions that we will select the design functions, i.e., the objectives and constraints.

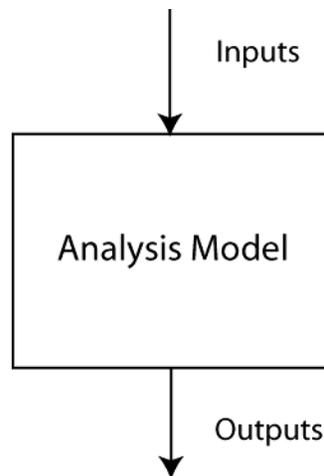


Fig. 1.1. The operation of analysis models

Thus from a very general viewpoint, analysis models require inputs—analysis variables—and compute outputs—analysis functions. Essentially all analysis models can be viewed this way.

2.2. An Example—the Two-bar Truss

At this point an example will be helpful.

Consider the design of a simple tubular symmetric truss shown in Fig. 1.2 below (problem originally from Fox¹). A *design* of the truss is specified by a unique set of values for the analysis variables: height (H), diameter, (d), thickness (t), separation distance (B), modulus

¹ R.L. Fox, *Optimization Methods in Engineering Design*, Addison Wesley, 1971

of elasticity (E), and material density (ρ). Suppose we are interested in designing a truss that has a minimum weight, will not yield, will not buckle, and does not deflect "excessively," and so we decide our model should calculate weight, stress, buckling stress and deflection—these are the analysis functions.

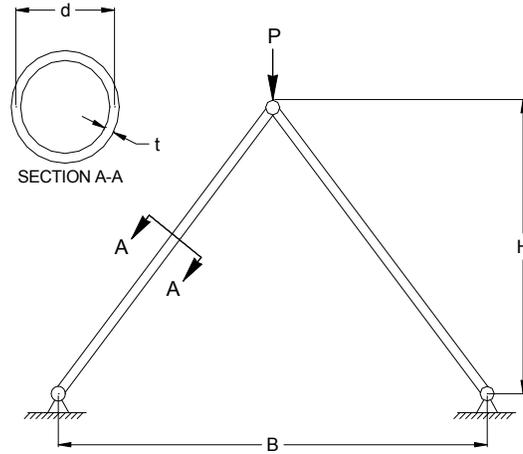


Fig. 1.2 - Layout for the Two-bar truss model.

In this case we can develop a model of the truss using explicit mathematical equations. These equations are:

$$Weight = \rho \cdot 2 \cdot \pi \cdot d \cdot t \cdot \sqrt{\left(\frac{B}{2}\right)^2 + H^2}$$

$$Stress = \frac{P \cdot \sqrt{\left(\frac{B}{2}\right)^2 + H^2}}{2 \cdot t \cdot \pi \cdot d \cdot H}$$

$$Buckling\ Stress = \frac{\pi^2 E (d^2 + t^2)}{8 \left[\left(\frac{B}{2}\right)^2 + H^2 \right]}$$

$$Deflection = \frac{P \cdot \left[\left(\frac{B}{2}\right)^2 + H^2 \right]^{(3/2)}}{2 \cdot t \cdot \pi \cdot d \cdot H^2 \cdot E}$$

The analysis variables and analysis functions for the truss are also summarized in Table 1.1. We note that the analysis variables represent all of the quantities on the right hand side of the

equations give above. When all of these are given specific values, we can *evaluate* the model, which refers to calculating the functions.

Table 1.1 - Analysis variables and analysis functions for the Two-bar truss.

<u><i>Analysis Variables</i></u>	<u><i>Analysis Functions</i></u>
<i>B, H, t, d, P, E, ρ</i>	Weight, Stress, Buckling Stress, Deflection

An example design for the truss is given as,

Analysis Variables	Value
Height, H (in)	30.
Diameter, d (in)	3.
Thickness, t (in)	0.15
Separation distance, B (inches)	60.
Modulus of elasticity (1000 lbs/in ²)	30,000
Density, ρ (lbs/in ³)	0.3
Load (1000 lbs)	66
Analysis Functions	Value
Weight (lbs)	35.98
Stress (ksi)	33.01
Buckling stress (ksi)	185.5
Deflection (in)	0.066

We can obtain a new design for the truss by changing one or all of the analysis variable values. For example, if we change thickness from 0.15 in to 0.10 in., we find that weight has decreased, but stress and deflection have increased, as given below,

Analysis Variables	Value
Height, H (in)	30.
Diameter, d (in)	3.
Thickness, t (in)	0.1
Separation distance, B (inches)	60.
Modulus of elasticity (1000 lbs/in ²)	30,000
Density, ρ (lbs/in ³)	0.3
Load (1000 lbs)	66
Analysis Functions	Value
Weight (lbs)	23.99
Stress (psi)	49.52

Buckling stress (psi)	185.3
Deflection (in)	0.099

3. Models and Optimization by Trial-and-Error

As discussed in the previous section, the “job,” so to speak, of the analysis model is to compute the values of analysis functions. The designer specifies values for analysis variables, and the model computes the corresponding functions.

Note that the analysis software does not make any kind of “judgment” regarding the goodness of the design. If an engineer is designing a bridge, for example, and has software to predict stresses and deflections, the analysis software merely reports those values—it does not suggest how to change the bridge design to reduce stresses in a particular location. Determining how to improve the design is the job of the designer.

To improve the design, the designer will often use the model in an iterative fashion, as shown in Fig. 1.3 below. The designer specifies a set of inputs, evaluates the model, and examines the outputs. Suppose, in some respect, the outputs are not satisfactory. Using intuition and experience, the designer proposes a new set of inputs which he or she feels will result in a better set of outputs. The model is evaluated again. This process may be repeated many times.

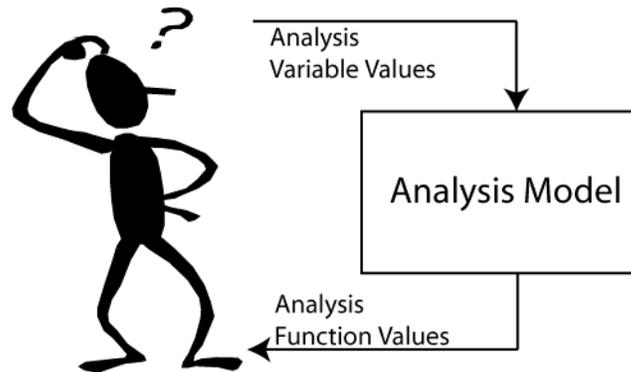


Fig. 1.3. Common “trial-and-error” iterative design process.

We refer to this process as “optimization by design trial-and-error.” This is the way most analysis software is used. Often the design process ends when time and/or money run out.

Note the mismatch of technology in Fig 1.3. On the right hand side, the model may be evaluated with sophisticated software and the latest high-speed computers. On the left hand side, design decisions are made by trial-and-error. The analysis is high tech; the decision making is low tech.

4. Optimization with Computer Algorithms

Computer-based optimization is an attempt to bring some high-tech help to the decision making side of Fig. 1.3. With this approach, the designer is taken out of the trial-and-error

loop. The computer is now used to both evaluate the model and search for a better design. This process is illustrated in Fig. 1.4.

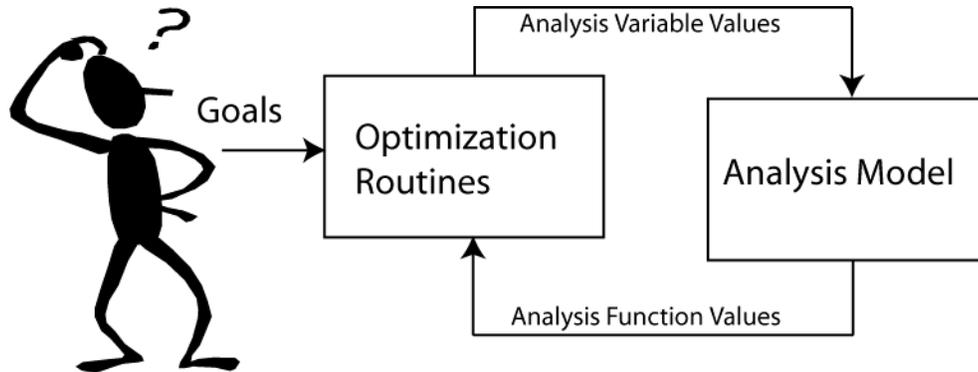


Fig. 1.4. Moving the designer out of the trial-and-error loop with computer-based optimization software.

The designer now operates at a higher level. Instead of adjusting variables and interpreting function values, the designer is specifying goals for the design problem and interpreting optimization results. The design space can be much more completely explored. Usually a better design can be found in a shorter time.

5. Specifying an Optimization Problem

5.1. Variables, Objectives, Constraints

Optimization problems are often specified using a particular form. That form is shown in Fig. 1.5. First the design variables are listed. Then the objectives are given, and finally the constraints are given. The abbreviation “s.t.” stands for “subject to.”

Find height and diameter to:

Minimize Weight

s. t.

Stress ≤ 100
(Stress-Buckling Stress) ≤ 0
Deflection ≤ 0.25

Fig. 1.5 Example specification of optimization problem for the Two-bar truss

Note that to define the buckling constraint, we have combined two analysis functions together. Thus we have *mapped* two analysis functions to become one design function.

We can specify several optimization problems using the same analysis model. For example, we can define a different optimization problem for the Two-bar truss to be:

Find thickness and diameter to:

Minimize Stress

s.t.

Weight ≤ 25.0

Deflection ≤ 0.25

Fig. 1.6 A second specification of an optimization problem for the Two-bar truss

The specifying of the optimization problem, i.e. the selection of the design variables and functions, is referred to as the *mapping between the analysis space and the design space*. For the problem defined in Fig. 1.5 above, the mapping looks like:

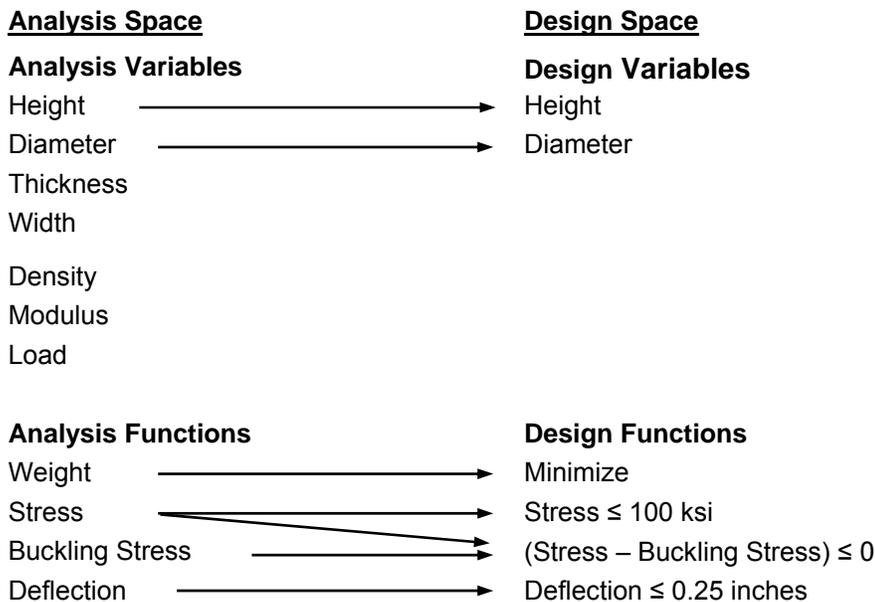


Fig. 1.7- Mapping Analysis Space to Design Space for the Two-bar Truss.

We see from Fig. 1.7 that the design variables are a subset of the analysis variables. This is always the case. In the truss example, it would never make sense to make load a design variable—otherwise the optimization software would just drive the load to zero, in which case the need for a truss disappears! Also, density and modulus would not be design variables unless the material we could use for the truss could change. In that case, the values these two variables could assume would be *linked* (the modulus for material A could only be matched with the density for material A) and would also be *discrete*. The solution of discrete variable optimization problems is discussed in Chapters 4 and 5. At this point, we will assume all design variables are continuous.

In like manner, the design functions are a subset of the analysis functions. In this example, all of the analysis functions appear in the design functions. However, sometimes analysis functions are computed which are helpful in understanding the design problem but which do

not become objectives or constraints. Note that the analysis function “stress” appears in two design functions. Thus it is possible for one analysis function to appear in two design functions.

In the examples given above, we only have one objective. This is the most common form of an optimization problem. It is possible to have multiple objectives. However, usually we are limited to a maximum of two or three objectives. This is because objectives usually are *conflicting* (one gets worse as another gets better) and if we specify too many, the algorithms can’t move. The solution of multiple objective problems is discussed in more detail in Chapter 5.

We also note that all of the constraints in the example are “less-than” inequality constraints. The limits to the constraints are appropriately called the *allowable values* or *right hand sides*. For an optimum to be valid, all constraints must be satisfied. In this case, stress must be less than or equal to 100; stress minus buckling stress must be less than or equal to 0, and deflection must be less than or equal to 0.25.

Most engineering constraints are inequality constraints. Besides less-than (\leq) constraints, we can have greater-than (\geq) constraints. Any less-than constraint can be converted to a greater-than constraint or vice versa by multiplying both sides of the equation by -1. It is possible in a problem to have dozens, hundreds or even thousands of inequality constraints. When an inequality constraint is equal to its right hand side value, it is said to be *active* or *binding*.

Besides inequality constraints, we can also have equality constraints. Equality constraints specify that the function value should equal the right hand side. Because equality constraints severely restrict the design space (each equality constraint absorbs one degree of freedom), they should be used carefully.

5.2. Example: Specifying the Optimization Set-up of a Steam Condenser

It takes some experience to be able to take the description of a design problem and abstract out of it the underlying optimization problem. In this example, a design problem for a steam condenser is given. Can you identify the appropriate analysis/design variables? Can you identify the appropriate analysis/design functions?

Description:

Fig. 1.8 below shows a steam condenser. The designer needs to design a condenser that will cost a minimum amount and condense a specified amount of steam, m_{\min} . Steam flow rate, m_s , steam condition, x , water temperature, T_w , water pressure P_w , and materials are specified. Variables under the designer’s control include the outside diameter of the shell, D ; tube wall thickness, t ; length of tubes, L ; number of passes, N ; number of tubes per pass, n ; water flow rate, m_w ; baffle spacing, B , tube diameter, d . The model calculates the actual steam condensed, m_{cond} , the corrosion potential, CP , condenser pressure drop, ΔP_{cond} , cost, C_{cond} , and overall size, V_{cond} . The overall size must be less than V_{max} and the designer would like to limit overall pressure drop to be less than ΔP_{max}

Discussion:

This problem description is somewhat typical of many design problems, in that we have to infer some aspects of the problem. The wording, “Steam flow rate, m_s , steam condition, x , water temperature, T_w , water pressure P_w , and materials are specified,” indicates these are either analysis variables that are not design variables (“unmapped analysis variables”) or constraint right hand sides. The key words, “Variables under the designer’s control...” indicate that what follows are design variables.

Statements such as “The model calculates...” and “The designer would also like to limit” indicate analysis or design functions. The phrase, “The overall size must be less than...” clearly indicates a constraint.

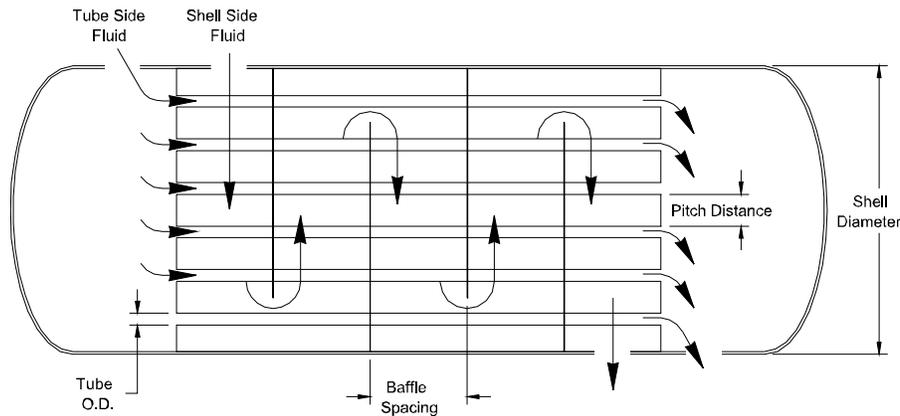


Fig. 1.8. Schematic of steam condenser.

Thus from this description it appears we have the following,

<p>Analysis Variables:</p> <ul style="list-style-type: none"> Outside diameter of the shell, D Tube wall thickness, t Length of tubes, L Number of passes, N Number of tubes per pass, n Water flow rate, m_w Baffle spacing, B, Tube diameter, d Steam flow rate, m_s Steam condition, x Water temperature, T_w Water pressure, P_w Material properties 	<p>Design Variables:</p> <ul style="list-style-type: none"> Outside diameter of the shell, D Tube wall thickness, t Length of tubes, L Number of passes, N Number of tubes per pass, n Water flow rate, m_w Baffle spacing, B, Tube diameter, d.
<p>Analysis Functions:</p> <ul style="list-style-type: none"> Cost, C_{cond} Overall size, V_{cond} Steam condensed, m_{cond} Condenser pressure drop, ΔP_{cond} corrosion potential, CP 	<p>Design Functions:</p> <ul style="list-style-type: none"> Minimize Cost s.t. $V_{cond} \leq V_{max}$ $m_{cond} \geq m_{min}$ $\Delta P_{cond} \leq \Delta P_{max}$

6. Concepts of Design Space

So far we have discussed the role of analysis software within optimization. We have mentioned that the first step in optimizing a problem is getting a good analysis model. We have also discussed the second step: setting up the design problem. Once we have defined the optimization problem, we are ready to start searching the design space. In this section we will discuss some concepts of design space. We will do this in terms of the familiar truss example.

For our truss problem, we defined the design variables to be height and diameter. In Fig. 1.9 we show a contour plot of the design space for these two variables. The plot is based on data created by meshing the space with a grid of values for height and diameter; other analysis variables were fixed at values shown in the legend on the left. The solid lines (without the triangular markers) represent contours of weight, the objective. We can see that many combinations of height and diameter result in the same weight. We can also see that weight decreases as we move from the upper right corner of the design space towards the lower left corner.

Constraint boundaries are also shown on the plot, marked 1-3. These boundaries are also contour lines—the constraint boundary is the contour of the function which is equal to the allowable value. The small triangular markers show the *feasible* side of the constraint. The space where all constraints are feasible is called the *feasible space*. The feasible space for this example is shown in Fig. 1.9. By definition, an optimum lies in the feasible space.

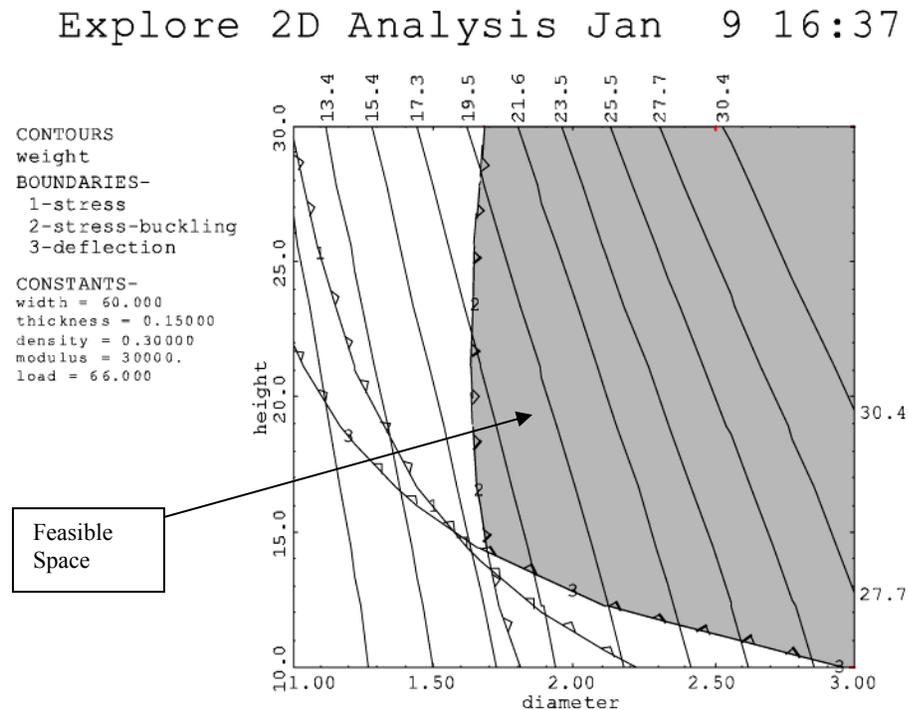


Fig. 1.9. Contour plot showing the design space for the Two-bar Truss.

We can easily see from the contours and the constraint boundaries that the optimum to this problem lies at the intersection of the buckling constraint and deflection constraint. We say that these are binding (or active) constraints. Several other aspects of this optimum should be noted.

First, this is a *constrained* optimum, since it lies on the constraint boundaries of buckling and deflection. These constraints are *binding* or *active*. If we relaxed the allowable value for these constraints, the optimum would improve. If we relaxed either of these constraints enough, stress would become a binding constraint.

It is possible to have an *unconstrained* optimum when we are optimizing, although not for this problem. Assuming we are minimizing, the contours for such an optimum would look like a valley. Contour plots showing unconstrained optimums are given in Fig. 1.10

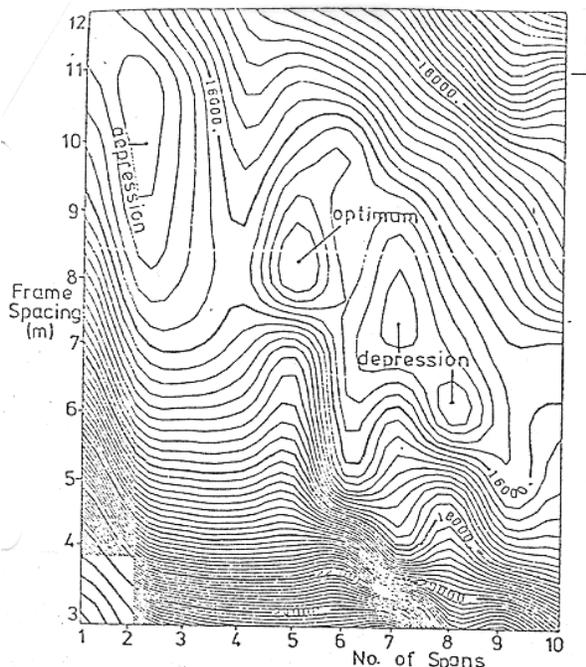
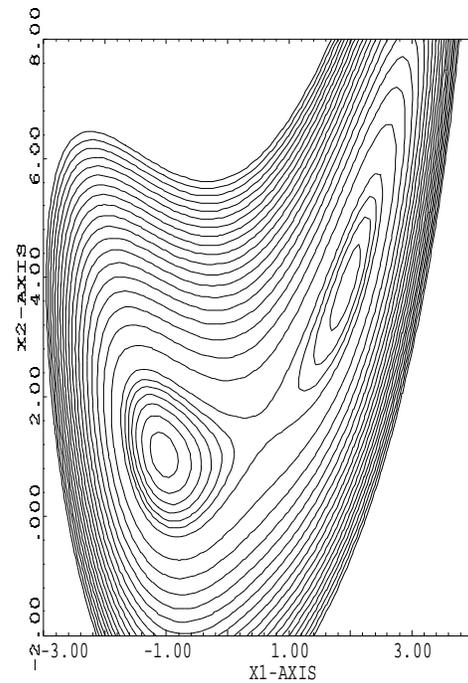


Figure 13. OPTIMAL RESPONSE FOR 2000 M² BUILDINGS



$$f(x_1, x_2) = 4 + 4.5x_1 - 4x_2 + x_1^2 + 2x_2^2 - 2x_1x_2 + x_1^4 - 2x_1^2x_2$$

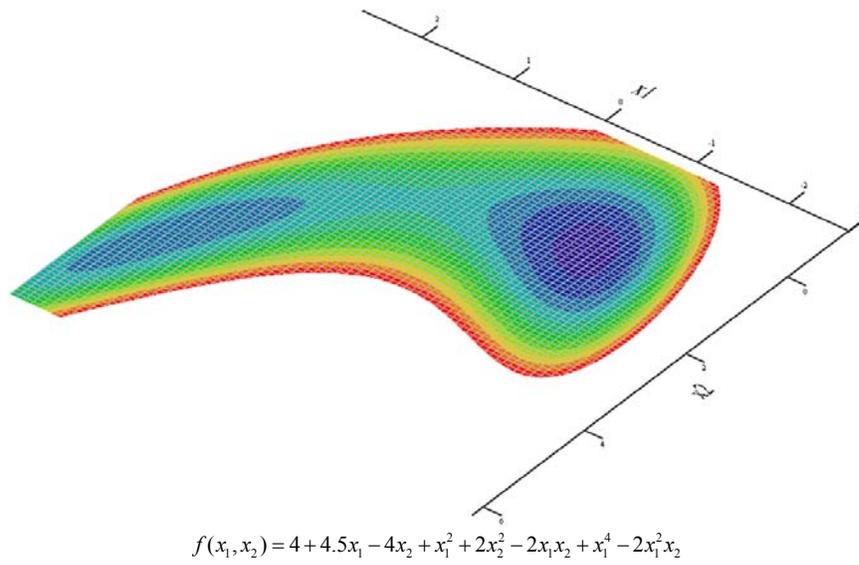


Fig. 1.10 Contour plots which show unconstrained optimums

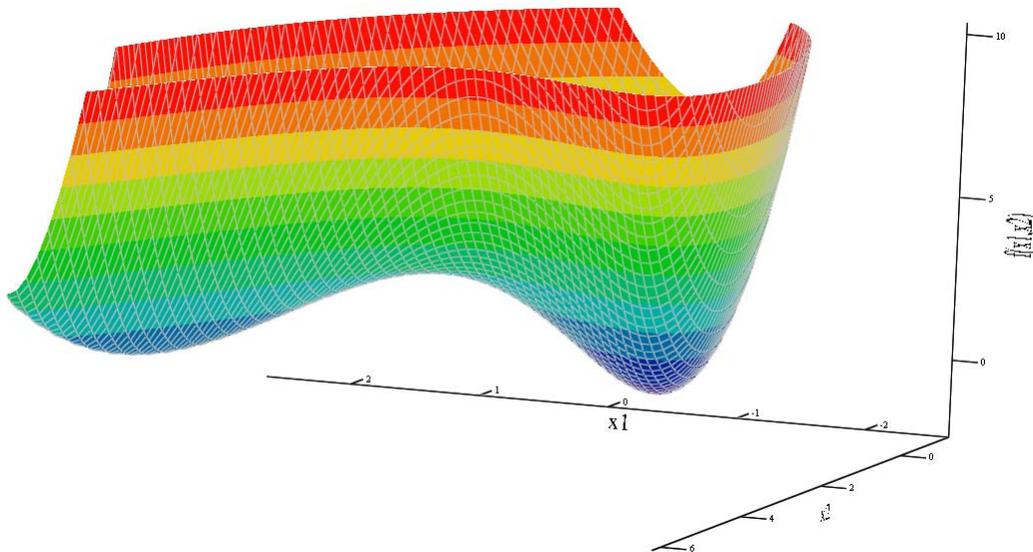


Fig. 1.11. Surface plot of the function:

$$f(x_1, x_2) := 4 + 4.5x_1 - 4x_2 + x_1^2 + 2x_2^2 - 2x_1x_2 + x_1^4 - 2x_1^2x_2$$

Second, this optimum is a *local* optimum. A local optimum is a point which has the best objective value of any feasible point in its neighborhood. We can see that no other near-by feasible point has a better value than this optimum.

Finally, this is also a *global* optimum. A global optimum is the best value over the entire design space. Obviously, a global optimum is also a local optimum. However, if we have several local optimums, the global optimum is the best of these.

It is not always easy to tell whether we have a global optimum. Most optimization algorithms only guarantee finding a local optimum. In the example of the truss we can tell the optimum is global because, with only two variables, we can graph the entire space, and we can see that this is the only optimum in this space. If we have more than two variables, however, it is difficult to graph the design space (and even if we could, it might be too expensive to do so) so we don't always know for certain that a particular optimum is global. Often, to try to determine if a particular solution is a global optimum, we start the optimization algorithms from several different design points. If the algorithms always converge to the same spot, we have some confidence that point is a global optimum.

7. How Algorithms Work

The most powerful optimization algorithms for nonlinear continuous problems use derivatives to determine a search direction in n dimensional space that improves the objective and satisfies the constraints. Usually these derivatives are obtained numerically. For the Two-bar truss, search paths for two different starting points are given in Fig. 1.11

Assuming derivatives are obtained numerically, the number of times the algorithms must evaluate the model is roughly 10-15 times the number of optimization variables. Thus a problem with 3 variables will require roughly 30-45 evaluations of the model; a problem with 10 variables will require 100-150 model evaluations.

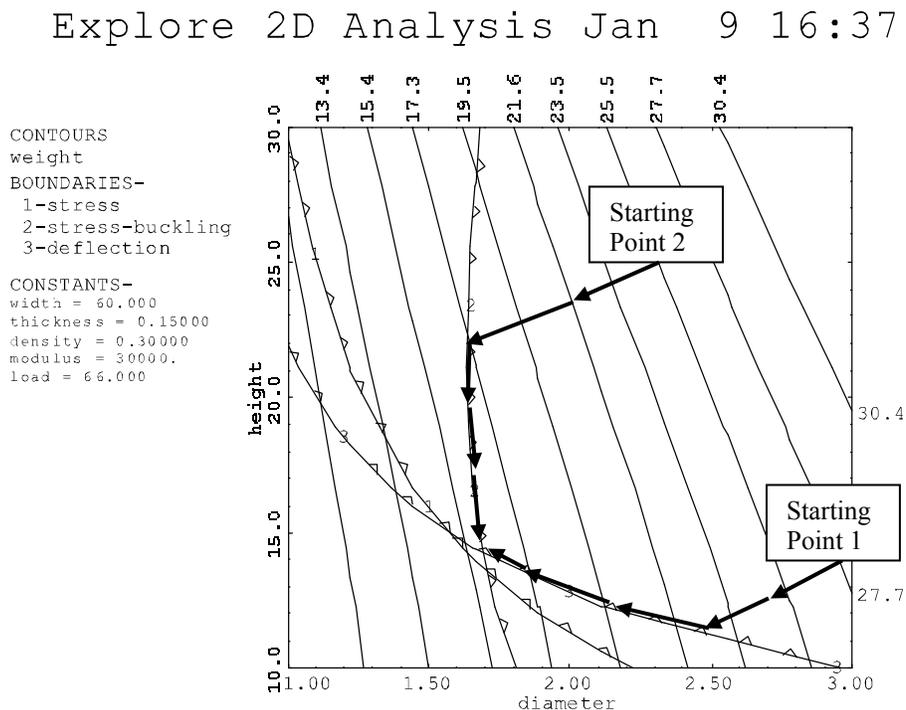


Fig. 1.12. Two paths algorithms might follow for two different starting points.

8. Cautions Regarding Optimization

Optimization algorithms can powerfully assist the designer in quantitative design synthesis. However,

1. The designer should always carefully and thoroughly validate the engineering model. Optimization of an inaccurate model is modestly illuminating at best and misleading and a waste of time at worst. Often optimization algorithms will exploit weaknesses in the model if they exist. As an example, regarding thermal systems, you can get some very high performance designs if a model does not enforce the second law of thermodynamics!
2. The algorithms help a designer optimize a particular design concept. At no point will the algorithms suggest that a different concept would be more appropriate. Achieving an overall optimum is dependent upon selection of the best concept as well as quantitative optimization.
3. Most engineering designs represent a compromise among conflicting objectives. Usually the designer will want to explore a number of alternative problem definitions to obtain insight and understanding into the design space. Sometimes non-quantitative considerations will drive the design in important ways.
4. We need to make sure the optimization problem represents the real problem we need to solve. Fox makes an important statement along these lines,

“In engineering design we often feel that the objective functions are self evident. But there are pitfalls. Care must be taken to optimize with respect to the objective function which most nearly reflects the *true* goals of the design problem. Some examples of common errors...should help focus this point. In static structures, the fully utilized (fully stressed) design is not always the lightest weight; the lightest weight design is not always the cheapest; in mechanisms, the design with optimum transmission angle does not necessarily have the lowest force levels; minimization of acceleration level at the expense of jerk (third derivative) may result in inadequate dynamic response.”

5. We need to be careful to optimize the *system* and not just individual parts. As an example, the following poem by Oliver Wendell Holmes describes a “one horse” shay (a wagon) so well built that it lasted 100 years, after which it went completely to pieces all at once.

But the Deacon swore (as Deacons do),
With an “I dew vum,” or an “I tell yeou,”
He would build one shay to beat the taown
‘N’ the keountry ‘n’ all the kentry raoun’;
It should be so built that it *couldn’t* break daown:
--”Fur,” said the Deacon, “‘t’s mighty plain
That the weakes’ place mus’ stan’ the strain;
‘N’ the way t’ fix it, uz I maintain,
Is only jest

T' make that place uz strong uz the rest.”

....

You see, of course, if you're not a dunce,
How it went to pieces all at once,--
All at once, and nothing first,--
Just as bubbles do when they burst.
End of the wonderful one-hoss shay.
Logic is logic. That's all I say.

Almost always in optimization we must take a system view, rather than an individual component view.

CHAPTER 2

MODELING CONCEPTS

1 Introduction

As was discussed in the previous chapter, in order to apply optimization methods we must have a model to optimize. As we also mentioned, obtaining a good model of the design problem is the most important step in optimization. In this chapter we discuss some modeling concepts that can help you develop models which can successfully be optimized. We also discuss the formulation of objectives and constraints for some special situations. We look at how graphics can help us understand the nature of the design space and the model. We end with an example optimization of a heat pump.

2 Physical Models vs. Experimental Models

Two types of models are often used with optimization methods: physical models and experimental models. Physical models are based on the underlying physical principles that govern the problem. Experimental models are based on models of experimental data. Some models contain both physical and experimental elements. We will discuss both types of models briefly.

2.1 Physical Models

Physical models can be either analytical or numerical in nature. For example, the Two-bar truss is an analytical, physical model. The equations are based on modeling the physical phenomena of stress, buckling stress and deflection. The equations are all closed form, analytical expressions. If we used numerical methods, such as the finite element method, to solve for the solution to the model, we would have a numerical, physical model.

2.2 Experimental Models

Experimental models are based on experimental data. A functional relationship for the data is proposed and fit to the data. If the fit is good, the model is retained; if not, a new relationship is used. For example if we wish to find the friction factor for a pipe, we could refer to the Moody chart, or use expressions based on a curve fit of the data.

3 Modeling Considerations

3.1 Making the Model Robust

During optimization, the algorithms will move through the design space seeking an optimum. Sometimes the algorithms will generate unanticipated designs during this process. To have a successful optimization, the model must be able to handle these designs, i.e. it must be *robust*. Before optimizing, you should consider if there are there any designs for which the model is undefined. If such designs exist, you need to take steps to insure either that 1) the optimization problem is defined such that the model will never see these designs, or 2) the model is made “bulletproof” so it can survive these designs.

For example, for the Two-bar truss, the truss becomes undefined if the diameter or thickness go to zero. Although unlikely to ever happen, we can insure this does not occur by defining non-zero lower bounds for these variables—the algorithms should never violate upper or lower bounds. (This is not true of constraints, however, which the algorithms sometimes will violate during the search.)

The above case seems a little farfetched, in that truss with a zero thickness or diameter would not make any sense. However, sometimes models can crash for designs that do make sense. Example 3.1.1 presents such a case.

3.1.1 Example: Log mean temperature difference

In the design of a shell and tube heat exchanger, the average temperature difference across the exchanger is given by the log mean temperature difference:

$$\Delta T_{LMTD} = \frac{(T_{h2} - T_{c2}) - (T_{h1} - T_{c1})}{\ln[(T_{h2} - T_{c2}) / (T_{h1} - T_{c1})]} \quad (2.1)$$

where $(T_{h2} - T_{c2})$ is, for example, the difference in the temperature of the hot and cold fluids at one end of the exchanger, denoted by subscript 2. This difference is sometimes called the “temperature of approach.” A perfectly legitimate case is to have the temperature of approach at both ends of the exchanger be the same, i.e.,

$$(T_{h2} - T_{c2}) = (T_{h1} - T_{c1}) \quad (2.2)$$

in which case the denominator becomes $\ln[1] = 0$, and the expression is undefined. It turns out in this case the appropriate expression for the temperature difference is the “arithmetic mean temperature difference”:

$$\Delta T_{AMTD} = [(T_{h2} - T_{c2}) + (T_{h1} - T_{c1})] / 2. \quad (2.3)$$

How would you handle this as part of your computer model? You would implement this as an IF ELSE statement. If the absolute value of the difference of the two temperatures of approach were below some tolerance, the arithmetic temperature difference would be used instead of the log mean temperature difference. This would prevent the model from crashing. The example of a heat pump at the end of this chapter implements this strategy.

Sometimes the values passed to the model, even though within the bounds, do not make sense. For example, if the wire diameter of a spring were more than ½ the coil diameter, as illustrated in the figure below, the spring could not be physically realized. Since it is unlikely this would represent an optimum design, the model just needs to be able to compute some sort of values without crashing and thereby halting execution. The algorithms will move away from this design as the optimization proceeds.

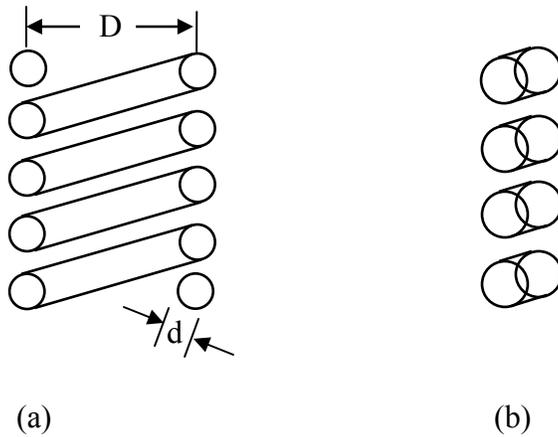


Fig. 2.1 Cross section of a spring. (a) Wire diameter, d , is reasonable relative to coil diameter, D . (b) Wire diameter is not reasonable relative to coil diameter.

One apparently obvious way to prevent something like this from happening is to include a constraint that restricts the relative sizes of the wire (d) and coil diameters (D), e.g.

$$2d < 0.3D \quad (2.4)$$

Often including such constraints will suffice. However it should be noted that some optimization algorithms will violate constraints during a search, so this may not always work.

Finally, some models are just inherently fragile. For certain combinations of variables, the models do not converge or in some manner fail. If the model cannot easily be made more robust, then the following strategy can be used.

If a design is passed in from the optimization software for which the model cannot compute some functions, that design is intercepted at some appropriate point inside the model, and a *penalty* function is passed back in place of the actual objective (with reasonable values given to constraints). The penalty function value is high enough (assuming we are minimizing) that the algorithms will naturally move away from the point that is causing trouble.

If we are using an unconstrained algorithm on a constrained problem, the penalty function often is composed of the objective and a penalty proportional to the violation of the constraints:

$$P = f + \sum_{i=1}^{viol} K_i g_i \quad (2.5)$$

where K_i is a large constant, g_i is the value of the constraint violation, and the subscript i only goes over the violated constraints.

The important point to note is that the model needs to be able to survive such designs, so the optimization can proceed.

3.2 Making Sure the Model is Adequate

Engineering models always involve assumptions and simplifications. An engineering model should be as simple as possible, but no simpler. If the model is not *adequate*, i.e., does not contain all of the necessary physics, the optimization routines will often exploit the inadequacy in order to achieve a better optimum. Obviously such an optimum does not represent reality.

For example, in the past, students have optimized the design of a bicycle frame. The objective was to minimize weight subject to constraints on stress and column buckling. The optimized design looked like it was made of pop cans: the frame consisted of large diameter, very thin wall tubes. The design satisfied all the given constraints.

The problem, however, was that the model was not adequate, in that it did not consider local buckling (the type of buckling which occurs when a pop can is crushed). For the design to be realistic, it needed to include local buckling.

Another example concerns thermal systems. If the Second Law of Thermodynamics is not included in the model, the algorithms might take advantage of this to find a better optimum (you can achieve some wonderful results if the Second Law doesn't have to be obeyed!). For example, I have seen optimal solutions which were excellent because heat could flow "uphill," i.e. against a temperature gradient.

3.3 Testing the Model Thoroughly

The previous two sections highlight the need for a model which will be optimized to be tested thoroughly. The model should first be tested at several points for which solutions (at least "ballpark solutions") are known. The model should then be exercised and pushed more to its limits to see where possible problems might lie.

3.4 Reducing Numerical Noise

Many engineering models are *noisy*. Noise refers to a lack of significant figures in the function values. Noise often results when approximation techniques or numerical methods are used to obtain a solution. For example, the accuracy of finite element methods depends on the "goodness" of the underlying mesh. Models that involve finite difference methods, numerical integration, or solution of sets of nonlinear equations all contain noise to some degree. It is important that you recognize that noise may exist. Noise can cause a number of problems, *but the most serious is that it may introduce large errors into numerical derivatives*. I have seen cases where the noise in the model was so large that the derivatives were not even of the right sign.

A discussion of the effect of noise and error on derivatives is given in Section 7.

Sometimes you can reduce the effect of noise by tightening convergence tolerances of numerical methods.

4 Proper Scaling of the Model Variables and Functions

Optimization algorithms can perform much better when functions and variables have been scaled to all be on the same order of magnitude. OptdesX uses the minimum and maximum bounds on variables to scale them to be between -1 and +1, using the equations,

$$(v_i)_{scaled} = \frac{(v_i)_{unscaled} - C_{1i}}{C_{2i}}, \text{ where } C_{1i} = \frac{Max_i + Min_i}{2} \text{ and } C_{2i} = \frac{Max_i - Min_i}{2} \quad (2.6)$$

In general, you should pick bounds which are as tight as possible. If during an optimization a lower or upper bound is reached (and it was set somewhat arbitrarily), it can always be relaxed.

Likewise, functions are also scaled to be on the order of 1 using the allowable and indifference values,

$$(f_i)_{scaled} = \frac{(f_i)_{unscaled} - Allowable_i}{Allowable_i - Indifference_i} \quad (2.7)$$

Since a derivative involves both functions and variables, it is affected by the scaling of both, according to the relationship,

$$\left(\frac{\partial f_i}{\partial x_j} \right)_{scaled} = \left(\frac{\partial f_i}{\partial x_j} \right)_{unscaled} \frac{(Max_j - Min_j)}{2 * (Allowable_i - Indifference_i)} \quad (2.8)$$

The OptdesX algorithms work with scaled values only. Scaling is important! Improperly scaled functions or variables may cause premature algorithm termination. When the problem is properly scaled, the gradients should all be roughly the same order of magnitude.

As shown in Fig. 2.2, we can view gradients using the Gradients window.

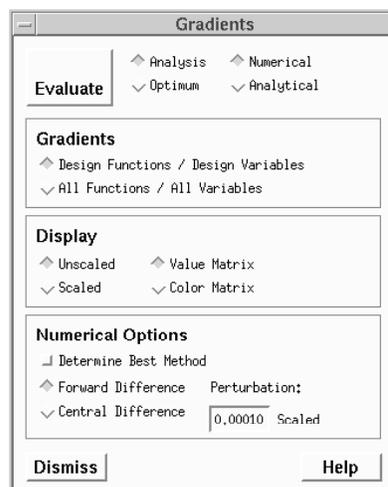


Fig. 2.2 Gradients Window

The unscaled gradients for the Two-bar Truss are given in Fig. 2.3:

	weight	stress	stress-buckling	deflection
height	0,5997942	-0,5501704	5,633651	-0,001100304
diameter	11,99578	-11,00350	-134,3756	-0,02200700
width	0,2998971	0,2750989	3,367010	0,001650621
thickness	239,9157	-220,0443	-226,2133	-0,4400886

Fig. 2.3. Unscaled gradients for the Two-bar Truss

Note the unscaled gradients differ by about five orders of magnitude. In terms of optimization, this may cause some variables and/or functions to overly dominate the optimization search with the end result the optimum is not found.

The scaled gradients are shown in Fig. 2.4,

	weight	stress	stress-buckling	deflection
height	0,1999314	-0,1100341	1,126730	-0,05501520
diameter	0,3998595	-0,2200700	-2,687512	-0,1100350
width	0,1999314	0,1100396	1,346804	0,1650621
thickness	1,799368	-0,9901994	-1,017960	-0,4950997

Fig. 2.4. Scaled gradients for the Two-bar Truss.

We see that the gradients are now all the same order of magnitude. This will facilitate the optimization search.

Scaling of functions, variables and derivatives does not change the solution to the optimization problem. Rather what we are doing is picking a different set of units (obviously we could calculate weight in grams or pounds and it shouldn't make a difference to the solution). In reality we have non-dimensionalized the problem so that all functions, variables and derivatives have similar magnitudes.

5 Formulating Objectives

5.1 Single Objective Problems

Most optimization problems, by convention, have one objective. This is at least partially because the mathematics of optimization were developed for single objective problems. Multiple objectives are treated briefly in this chapter and more extensively in Chap. 5.

Often in optimization the selection of the objective will be obvious: we wish to minimize pressure drop or maximize heat transfer, etc. However, sometimes we end up with a surrogate objective because we can't quantify or easily compute the real objective.

For example, it is common in structures to minimize weight, with the assumption the minimum weight structure will also be the minimum cost structure. This may not always be true, however. (In particular, if minimum weight is achieved by having every member be a different size, the optimum could be very expensive!) Thus the designer should always keep in mind the assumptions and limitations associated with the objective of the optimization problem.

Often in design problems there are other objectives or constraints for the problem which we can't include. For example, aesthetics or comfort are objectives which are often difficult to quantify. For some products, it might also be difficult to estimate cost as a function of the design variables.

These other objectives or constraints must be factored in at some point in the design process. The presence of these other considerations means that the optimization problem only partially captures the scope of the design problem. Thus the results of an optimization should be considered as one piece of a larger puzzle. Usually a designer will explore the design space and develop a spectrum of designs which can be considered as final decisions are made.

5.2 Special Objectives: Error Functions

Consider the four bar linkage shown in the Fig. 2.5a below (from *Optimization Methods for Engineering Design*, R.L. Fox, Addison Wesley, 1971). We would like the linkage to follow a particular path as given by the solid in line in Fig. 2.5b. The actual output of a particular design is given by the dashed line. In this case we would like to design a linkage such that the generated output and the desired output match as closely as possible.

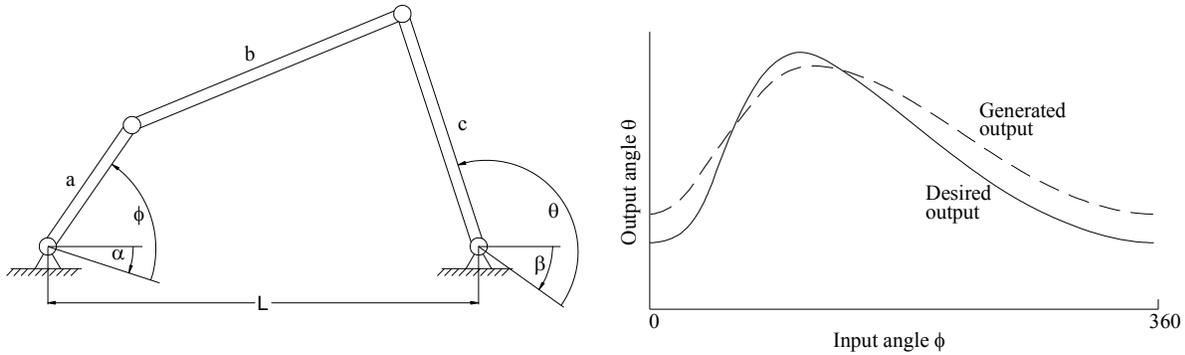


Fig 2.5 a) A four bar linkage. After Fox. b) Desired output vs. generated output

A question arises as to what the objective function should be. We need an objective which provides a measure of the error between the desired and actual output. The optimization algorithms would then try to reduce this error as much as possible. One such error function would be,

$$Error = \sum_{\phi=0}^{360} (\theta_{actual} - \theta_{desired})^2 \tag{2.9}$$

Squaring the difference in the above expression insures that the error accumulates whether positive or negative. Other error functions are also possible, such as an actual integration of the area between the two curves. If it is more important that the error be small for some parts of the range, we can add a weighting function,

$$Error = \sum_{\phi=0}^{360} w(\phi)(\theta_{actual} - \theta_{desired})^2 \tag{2.10}$$

Where $w(\phi)$ has higher values for the more important parts of the range. (Often weighting functions are constructed so the weights sum to 1.)

Error functions such as this are relatively common in engineering optimization. You should pay particular attention to scaling of an error function, as it is possible for a function such as this to vary over several orders of magnitude from the starting point to the optimum. Sometimes scaling which is appropriate at the start needs to be revised closer to the optimum.

5.3 Special Objectives: Economic Objective Functions

One type of optimization looks at the trade-off between capital costs made now, and savings resulting from the capital cost which occur later. For example, we could purchase a heat pump now (the capital cost) in order to save energy for a number of years. Typically the more expensive the heat pump is, the more energy we will save. We are interested in determining the optimal size of the heat pump that results in the best economic results.

To do this properly we must take into account the time value of money. The fact that money earns interest means that one dollar today is worth more than one dollar tomorrow. If the interest rate is 5%, compounded annually, \$1.00 today is the same as \$1.05 in one year. If expenses and/or savings are made at different times, we must convert them to the same time to make a proper comparison. This is the subject for engineering economics, so we will only briefly treat the subject here.

One approach to examining the economics of projects is called *net present value (NPV)*. For this approach, all economic transactions are converted to equivalent values in the present. In general, outflows (expenses) are considered negative; inflows (savings or income) are considered positive. There are two formulas which will be useful to us.

The present value of one future payment or receipt, F , at period n with interest rate i is,

$$P = \frac{F_n}{(1+i)^n} \quad (2.11)$$

This equation allows us to convert any future value to the present. This might not be very convenient, however, if we have many future values to convert, particularly if we have a series of uniform payments. The present value of a series of uniform amounts, A , where the first payment is made at the end of the first period is given by,

$$P = A \frac{(1+i)^n - 1}{i(1+i)^n} \quad (2.12)$$

5.3.1 Example: Net Present Value of Heat Pump

Suppose you can buy a heat pump for \$50,000. The pump is estimated to save \$12,000 per year for the five year life of the pump. An interest rate of 10% is assumed. Is the heat pump a good investment?

A time line showing the money flow rates is given below (size of arrows not to scale).

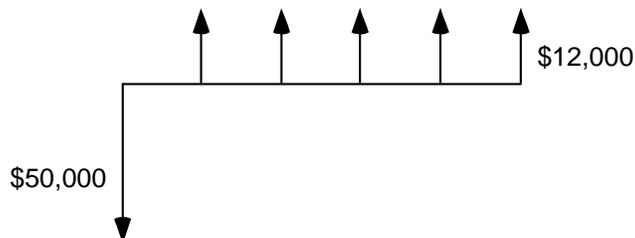


Fig. 2.6. Money flow rates for heat pump.

The initial expense of the pump, \$50,000, is already in the present, so this does not need to be changed. It will be considered negative, however, since it is money paid out.

The present value of the savings is given by,

$$P = \$12,000 \frac{(1 + 0.1)^5 - 1}{0.1(1 + 0.1)^5} = \$45,489 \quad (2.13)$$

$$NPV = -\$50,000 + \$45,489 = -\$4510$$

Since the Net Present Value is negative, this is not a good investment. This result might be a little surprising, since without taking into account the time value of money, the total savings is $5 * \$12,000 = \$60,000$. However, at 10% interest, the value of the savings, when brought back to the present, does not equal the value of the \$50,000 investment. This means we would be better off investing the \$50,000 at 10% than using it to buy the heat pump.

Note that (2.13) above assumes the first \$12,000 of savings is realized at the end of the first period, as shown by the timeline for the example.

When used in optimization, we would like to maximize net present value. In terms of the example, as we change the size of the heat pump (by changing, for example, the size of the heat exchangers and compressor), we change the initial cost and the annual savings.

Question: How would the example change if the heat pump had some salvage value at the end of the five years?

6 Formulating Constraints

6.1 Inequality and Equality Constraints

Almost all engineering problems are constrained. Constraints can be either inequality constraints ($\leq b_i$ or $\geq b_i$) or equality constraints ($= b_i$). The feasible region for inequality constraints represents the entire area on the feasible side of the allowable value; for equality constraints, the feasible region is only where the constraint is equal to the allowable value itself.

Because equality constraints are so restrictive, they should be avoided in optimization models whenever possible (an exception would be linear constraints, which are easily handled). If the constraint is simple, this can often be done by solving for the value of a variable explicitly. This eliminates a design variable and the equality constraint from the problem.

6.1.1 Example: Eliminating an Equality Constraint

Suppose we have the following equality constraint in our model,

$$x_1^2 + x_2^2 = 10 \quad (2.14)$$

Where x_1 and x_2 are design variables. How would we include this constraint in our optimization model?

We have two choices. The choices are mathematically equivalent but one choice is much easier to optimize than the other.

First choice: Keep x_1 and x_2 as design variables. Calculate the function,

$$g = x_1^2 + x_2^2 \quad (2.15)$$

Send this analysis function to the optimizer, and define the function as an equality constraint which is set equal to 10.

Second choice: Solve for x_1 explicitly in the analysis model,

$$x_1 = \sqrt{10 - x_2^2} \quad (2.16)$$

This eliminates x_1 as a design variable (since its value is calculated, it cannot be adjusted independently by the optimizer) and eliminates the equality constraint. The equality constraint is now implicitly embedded in the analysis model.

Which choice is better for optimization? The second choice is a better way to go hands down: it eliminates a design variable and a highly restrictive equality constraint. The second choice results in a simpler optimization model.

In one respect, however, the second choice is not as good: we can no longer explicitly put bounds on x_1 like we could with the first choice. Now the value of x_1 is set by (2.16); we can only control the range of values indirectly by the upper and lower bounds set for x_2 . (Also, we note we are only taking the positive roots of x_1 . We could modify this, however.)

6.2 Constraints Where the Allowable Value is a Function

It is very common to have constraints of the form:

$$\text{Stress} \leq \text{Buckling Stress} \quad (2.17)$$

Most optimization routines, however, cannot handle a constraint that has a function for the right hand side. This is easily remedied by rewriting the function to be,

$$\text{Stress} - \text{Buckling Stress} \leq 0 \quad (2.18)$$

In OptdesX, this mapping can be done two ways: in your analysis routine you can create a function which computes stress minus buckling stress, or you can make the mapping directly in the OptdesX interface:

stress-buckling	-37.26021	≤	2	0.000000	-50.00000
Sum stress	64.18629				
buckling	101.4465				

Fig. 2.7. Mapping two analysis functions to become one design function in OptdesX

6.3 Constraint with Two Allowable Values

Constraints sometimes also have upper and lower allowable values:

$$4 \leq \text{DiameterRatio} \leq 16 \tag{2.19}$$

We will break this into two constraints:

$$\begin{aligned} \text{DiameterRatio} &\leq 16 \\ \text{DiameterRatio} &\geq 4 \end{aligned}$$

In OptdesX this is accomplished by mapping the analysis function twice:

Diameter RatioL	7.684895	≤	1	16.00000	4.000000
Diameter RatioG	7.684895	≥	1	4.000000	16.00000

Fig. 2.8. Mapping one analysis function to become two constraints.

6.4 Taking the Maximum of a Group of Functions

Sometimes we would like to reduce the size of a problem by combining several constraints together. For example, in a finite element model, it is expensive to place a stress constraint on every element. One possibility is to group a block of elements together and take the maximum stress over the group. In OptdesX:

stress	38.21160	≤	3	39.60000	26.40000
Max stress1	33.01160				
stress2	38.21160				
stress3	36.11160				

Fig. 2.9. Taking the maximum of several functions to become one constraint.

However, the Max function has a drawback in that it will not be differentiable when the max value changes from one analysis function to another. Usually, however, the optimization algorithms can skip over this discontinuity.

6.5 Parametric Constraints

Suppose we have a truss where the load acts within some envelope, as shown in Fig. 2.10 below (from *Optimization Methods for Engineering Design*, R.L. Fox, Addison Wesley, 1971).

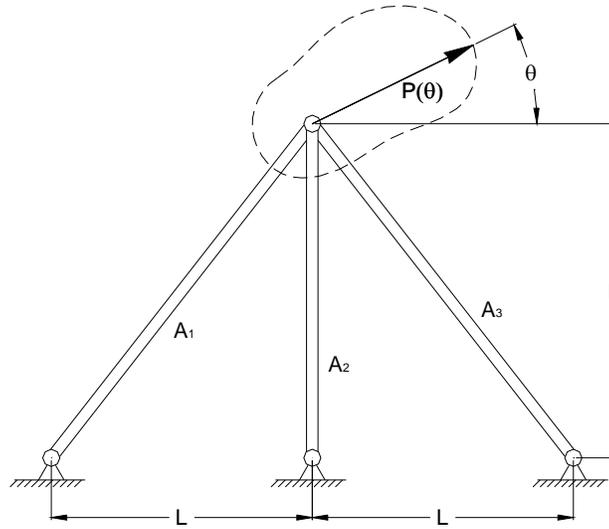


Fig. 2.10. A three-bar truss with a parametric load, after Fox.

The load can act in any direction; the magnitude is a function of θ :

$$P_1(\theta) = \frac{10.0}{\left(1 - \frac{1}{2} \cdot \cos\left(\theta - \frac{\pi}{6}\right)\right)} \text{ (kips)} \quad 0 \leq \theta \leq 2\pi \quad (2.20)$$

Thus the stress constraint becomes:

$$\text{Stress}(\theta) < S_y \quad 0 \leq \theta \leq 2\pi \quad (2.21)$$

Fox refers to this as a *parametric constraint*. How would we evaluate this type of constraint?

As you may have guessed, we would use a loop inside our analysis routine. We would loop through values of theta and keep track of the highest value of stress. This is the value which would be sent to the optimizer. Pseudo-code for the loop would be something like (where we assume theta is in degrees and is incremented by two degrees each time through the loop),

```

maxstress = 0.0;
For (theta = 0; theta <= 360; theta += 2.0;) {
    stress = (calculate stress);
    If (stress >> maxstress ) maxstress = stress;
}

```

7 Accurate Numerical Derivatives

In Section 2 we looked at how scaling can affect the values of derivatives. We noted it is advantageous for derivatives to be the same order of magnitude when optimizing. This is equivalent to selecting an optimum set of units for our variables. We also mentioned that

noise in the model can introduce error into derivatives. In this section we will look carefully at errors in numerical derivatives and learn how to reduce them. This is crucial if we expect a successful optimization.

7.1 Truncation Error

Most of the time we obtain derivatives numerically. An exception is structural optimization, where some packages support the calculation of analytical derivatives directly from the finite element results. The most common numerical method is forward difference. The error associated with this method can be derived from a Taylor Expansion. For a function of only one variable:

$$f(x + \Delta x) = f(x) + \frac{df}{dx} \Delta x + \frac{1}{2} \frac{d^2 f}{dx^2} \Delta x^2 + \dots \quad (2.22)$$

Solving for the derivative:

$$\frac{df}{dx} = \frac{f(x + \Delta x) - f(x)}{\Delta x} - \frac{1}{2} \frac{d^2 f}{dx^2} \Delta x + \dots \quad (2.23)$$

If we approximate the derivative as,

$$\frac{df}{dx} = \frac{f(x + \Delta x) - f(x)}{(\Delta x)} \quad (2.24)$$

then we see we have a *truncation error* of $\frac{1}{2} \frac{d^2 f}{dx^2} \Delta x + \dots$ which is proportional to Δx . Thus to reduce this error, we should make Δx small.

We can also derive a central difference derivative:

$$f(x + \Delta x) = f(x) + \frac{df}{dx} \Delta x + \frac{1}{2} \frac{d^2 f}{dx^2} \Delta x^2 + \frac{1}{6} \frac{d^3 f}{dx^3} \Delta x^3 + \dots \quad (2.25)$$

$$f(x - \Delta x) = f(x) - \frac{df}{dx} \Delta x + \frac{1}{2} \frac{d^2 f}{dx^2} \Delta x^2 - \frac{1}{6} \frac{d^3 f}{dx^3} \Delta x^3 + \dots \quad (2.26)$$

Subtracting the second expression from the first,

$$f(x + \Delta x) - f(x - \Delta x) = 2 \frac{df}{dx} \Delta x + \frac{1}{3} \frac{d^3 f}{dx^3} \Delta x^3 + \dots \quad (2.27)$$

Solving for the derivative,

$$\frac{df}{dx} = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} - \frac{1}{6} \frac{d^3 f}{dx^3} \Delta x^2 + \dots \quad (2.28)$$

If we approximate the derivative as,

$$\frac{df}{dx} = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} \quad (2.29)$$

then the truncation error is $\frac{1}{6} \frac{d^3 f}{dx^3} \Delta x^2 + \dots$, which is proportional to Δx^2 .

Assuming $\Delta x < 1.0$, the central difference method should have less error than the forward difference method. For example, if $\Delta x = 0.01$, the truncation error for the central difference derivative should be proportional to $(0.01)^2 = 0.0001$.

If the error of the central difference method is better, why isn't it the default? Because there is no free lunch! The central difference method requires two functions calls per derivative instead of one for the forward difference method. If we have 10 design variables, this means we have to call the analysis routine 20 times (twice for each variable) to get the derivatives instead of ten times. This can be prohibitively expensive.

7.2 Round-off Error

Besides truncation error, we also have *round-off* error. This is error associated with the number of significant figures in the function values. Returning to our expression for a forward difference derivative,

$$\frac{df}{dx} = \frac{f(x + \Delta x) + |\varepsilon| - f(x) + |\varepsilon|}{\Delta x} - \frac{1}{2} \frac{d^2 f}{dx^2} \Delta x + \dots \quad (2.30)$$

$$\frac{df}{dx} = \frac{f(x + \Delta x) - f(x)}{\Delta x} + \frac{2\varepsilon}{\Delta x} - \frac{1}{2} \frac{d^2 f}{dx^2} \Delta x + \dots \quad (2.31)$$

where $|\varepsilon|$ represents the error in the true function value caused by a lack of significant figures.

Recall that the true derivative is,

$$\frac{df}{dx} = \frac{f(x + \Delta x) - f(x)}{\Delta x} - \frac{1}{2} \frac{d^2 f}{dx^2} \Delta x + \dots \quad (2.32)$$

but we are estimating it as,

$$\frac{df}{dx} = \frac{f(x + \Delta x) - f(x)}{\Delta x} + \frac{2\varepsilon}{\Delta x} \quad (2.33)$$

where this last term is the round-off error. The total error is therefore,

$$\frac{2\varepsilon}{\Delta x} - \frac{1}{2} \frac{d^2 f}{dx^2} \Delta x + \dots \quad (2.34)$$

Thus we have two competing errors, truncation error and round-off error. To make truncation error small, Δx should be small; to reduce round-off error, Δx should be large. We will have to compromise.

7.3 Example of Truncation Error

We wish to compute the derivatives of the function $f(x) = x^3 + x^{1/2}$ at the point $x = 3$.

The true derivative at this point is 27.2886751.

With a forward difference derivative and $\Delta x = 0.01$,

$$\frac{df}{dx} = \frac{f(3+0.01) - f(3)}{0.01} = \frac{29.0058362 - 28.7320508}{0.01} = 27.37385$$

The absolute value of error is 0.08512.

Now suppose we use $\Delta x = 0.01$ with a central difference derivative:

$$\frac{df}{dx} = \frac{f(3+0.01) - f(3-0.01)}{2*0.01} = \frac{29.0058362 - 28.4600606}{0.02} = 27.28878$$

The absolute error has decreased to 0.000105

7.4 Example of Truncation and Round-off Error

Suppose, because of the type of model we have, we only have five significant figures for our data. Thus instead of 29.0058362, we really only have 29.006-----, where even though other digits are reported, they are only numerical noise. (Note we have rounded here—we might also specify *chopping*.)

With $\Delta x = 0.01$,

$$\frac{df}{dx} = \frac{f(3+0.01) - f(3)}{0.01} = \frac{29.006 - 28.732}{0.01} = 27.4000000$$

The absolute error has increased to 0.1113

Lets try a whole range of Δx 's:

Δx	Error
1.0	9.98
0.1	0.911
0.01	0.1113

0.001	0.2887
0.0001	2.7113
0.00001	27.728

If we plot this data,

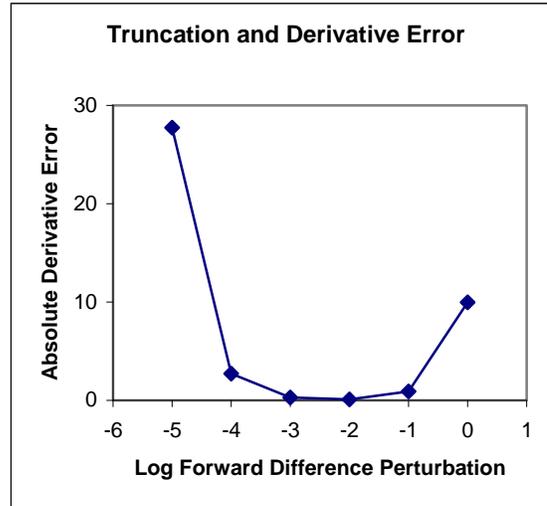


Fig. 2.11. Plot of total error for example

As we expected, we see that the data have a “U” shape. At small perturbations, round-off error dominates. At large perturbations, truncation error dominates. There is an optimal perturbation that gives us the minimum error in the numerical derivative.

Usually we don’t know the true value of the derivative so we can’t easily determine the optimal perturbation. However, understanding the source and control of errors in these derivatives can be very helpful. If we suspect we have a noisy model, i.e. we don’t have very many significant figures in our functions, *we use a central difference method with a large perturbation*. The large perturbation helps reduce the effects of noise, and the central difference method helps control the truncation error associated with a large perturbation. In OptdesX, where derivatives are scaled, we recommend a perturbation of 0.01 for noisy models instead of 0.0001, the default, along with the central difference method.

7.5 Partial Derivatives

The concepts of the preceding sections on derivatives extend directly to partial derivatives. If we had, for example, two variables, x_1 and x_2 , a finite forward difference partial derivative,

$\frac{\partial f}{\partial x_1}$, would be given by,

$$\frac{\partial f}{\partial x_1} \approx \frac{f(x_1 + \Delta x_1, x_2) - f(x_1, x_2)}{\Delta x_1} \quad (2.35)$$

Note that only x_1 is perturbed to evaluate the derivative. This variable would be set back to its base value and x_2 perturbed to find $\frac{\partial f}{\partial x_2}$.

In a similar manner, a central difference partial derivative for $\frac{\partial f}{\partial x_1}$ would be given by,

$$\frac{\partial f}{\partial x_1} \approx \frac{f(x_1 + \Delta x_1, x_2) - f(x_1 - \Delta x_1, x_2)}{2\Delta x_1} \quad (2.36)$$

8 Interpreting Graphics

This section may seem a disconnected subject—interpreting graphs of the design space. However graphs of the design space can tell us a lot about our model. If we can afford the computation, it can be very helpful to look at graphs of design space.

OptdesX supports two basic types of plots: sensitivity plots (where functions are plotted with respect to one variable) and contour plots (where functions are plotted with respect to two variables). To create a sensitivity plot, one variable is changed at equal intervals, according to the number of points specified, and the analysis routine is called at each point. If ten points are used, the analysis routine is called ten times. The function values are stored in a file—this file is then graphed.

To create a contour plot, two variables are changed at equal intervals (creating a mesh in the space) and the values stored in a file. If ten points are used for each variable, the analysis routine is called 100 times.

Some points to remember:

- Use graphs to help you verify your model. Do the plots make sense?
- Graphs can provide additional evidence that you have a global optimum. If the contours and constraint boundaries are smooth and relatively linear, then there are probably not a lot of local optima. Remember, though, you are usually only looking at a slice of design space.
- When you generate a contour plot, remember that all variables except the two being changed are held constant at their current values. Thus if you wish to see what the design space looks like around the optimum, you must be at an optimum.
- Graphs can often show why the algorithms may be having trouble, i.e. they can show that the design space is highly nonlinear, that the feasible region is disjoint, that functions are not smooth, etc.
- Graphs are interpolated from an underlying mesh. Make sure the mesh is adequate. The mesh must be fine enough to capture the features of the space (see example in manual pg 10-9 to 10-10). Also, you need to make sure that the ranges for the variables, set in the

Explore window, are such that the designs are meaningful. In some cases, if the ranges are not appropriate, you may generate designs that are undefined, and values such as “Infinity” or “NaN” will be written into the Explore file. OptdesX cannot graph such a file and will give you an error message.

- Optimal explore graphs show how the optimum changes as we change two *analysis* variables. (We cannot change two design variables—these are adjusted by the software to achieve an optimum.) These types of plots, where an optimization is performed at each mesh point, can be expensive. You must be very careful to set ranges so that an optimum can be found at each mesh point.

In the figures below we show some examples of contour plots which can help us gain insight into an analysis model.

The first plot, shown in Fig. 2.12 below, illustrates several features of interest. We note that that we have two optimums in the middle of the graph. We also have a step in the middle of the graph where the contours are steep—this would likely mean the algorithms would “fall” down the step and find the second optimum. Also, in the upper left corner the contours are very close together. The “diamond” shapes along the right side are probably not real—these can occur when the underlying mesh is not fine enough.

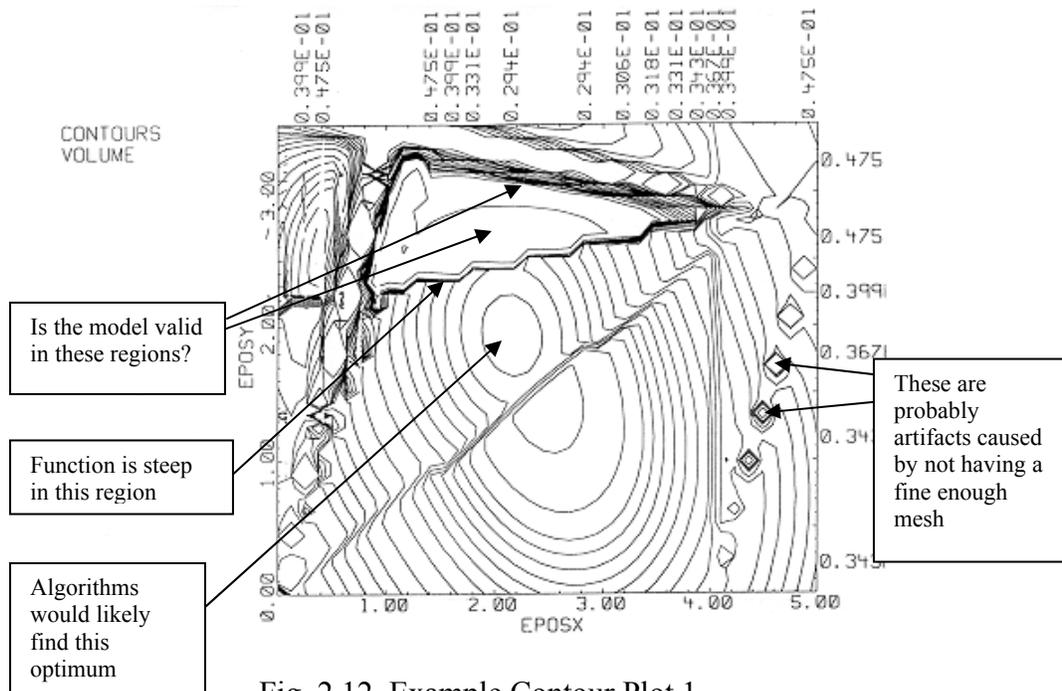


Fig. 2.12. Example Contour Plot 1

Our next example is shown in Fig. 2.13. In this example, which represents the design space for a spring, we have a lot of constraint boundaries. As we examine constraint boundaries, we see that the feasible region appears to be a triangular area, as shown. In the lower right hand corner, however, we have an area of suspicious looking constraint boundaries (constraint 2), which, if valid, would mean no feasible region exists. When we consider the

values of variables in this area, however, we understand that this boundary is bogus. In this part of the plot, the wire diameter is greater than the coil diameter. Since this is impossible with a real spring, we know this boundary is bogus and can be ignored.

Explore 2D Analysis Jan 15 10:34

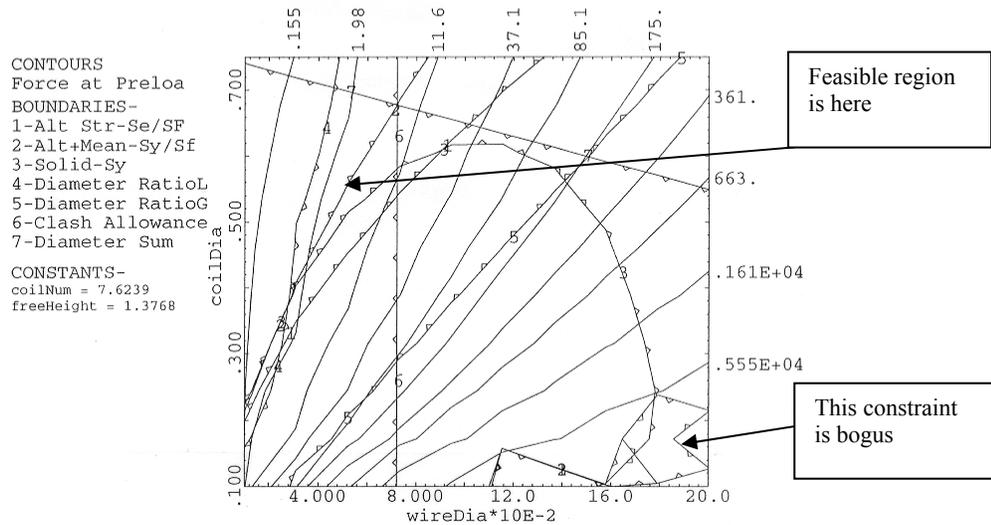


Fig. 2.13. Example Contour Plot 2

Our third and final example is given in Fig. 2.14. This is a sensitivity plot. Plots 2,3,4,5 look pretty reasonable. Plot 1, however, exhibits some noise. We would need to take this into account when numerical derivatives are computed. (Recall that we would want to use a central difference scheme with a larger perturbation.)

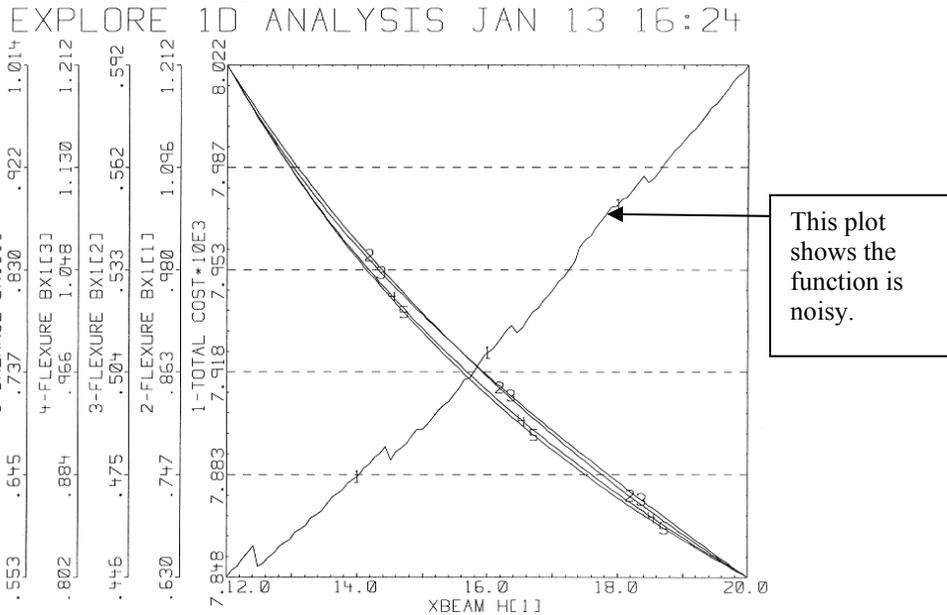


Fig. 2.14. Example Sensitivity Plot

9 Modeling and Optimization Example: Optimization of a Heat Pump

9.1 Problem Description

To save energy, the installation of a heat pump is proposed as shown in Fig. 2.15. The heat pump recovers heat from exhaust air and transfers it to incoming outdoor air. The building is electrically heated, so that if the heat pump does not bring the temperature of the incoming air up to 35°C , electric-resistance heat is used to supplement the heat pump.¹

Determine the size of compressor, condenser, and evaporator that provides a minimum total cost (capital costs and operating costs). The temperatures of approach in the evaporator and condenser should not be less than four degrees C.

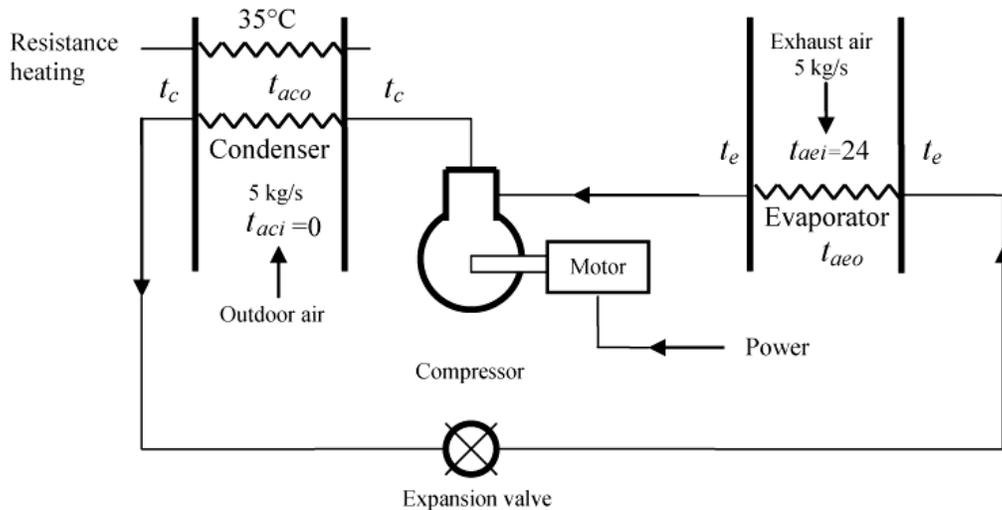


Fig. 2.15. Schematic of Heat Pump (After Stoecker)

Data

U Values of condenser and evaporator coils, $25 \text{ W} / (\text{m}^2 \cdot \text{K})$ based on air-side area.

Cost of coils, \$100 per square meter of air-side area

Compressor cost, including motor, \$220 per motor kilowatt

Power cost, 6 cents per kilowatt-hour

Interest rate, 10 percent

Economic life, 10 years

Number of hours of operation per year, 4000

Average outdoor temperature during the 4000 h, 0°C

Air flow rate through both coils, 5 kg/s

Temperatures of approach in the exchangers should be at least 4°C .

¹ This problem is a modified version taken from W. F. Stoecker, *Design of Thermal Systems, 3rd Edition*, McGraw Hill.

The performance characteristics of the compressor can be expressed as a coefficient of performance (COP), where

$$\text{COP} = \frac{\text{refrigeration rate, kW}}{\text{electric power to compressor motor, kW}}$$

(Refrigeration rate is the heat absorbed at the evaporator.) The COP is a function of the evaporating and condensing temperatures, t_e °C, and t_c °C, and can be represented by

$$\text{COP} = 7.24 + 0.352t_e - 0.096 t_c - 0.0055 t_e t_c \quad (2.37)$$

9.2 Initial Problem Discussion

This problem involves determining the optimal trade-off between first costs and operating costs. If we are willing to put in a more expensive heat pump (larger heat exchangers and compressor) we can recover more heat and energy costs drop. Conversely, we can reduce initial costs by installing a smaller heat pump (or no heat pump at all) but energy costs are then higher. This is a relatively common situation with engineering systems.

We have obviously made a number of simplifications to this problem. For example, we are not considering any benefit which might accrue from running the heat pump as an air conditioner during the summer. We are also using a relatively simple model for the condenser, evaporator and compressor. The optimization is also conducted on the basis of a constant outdoor air temperature of 0°C.

Although not considered here, during actual operation the COP of the system would vary with the outdoor air temperature. A more complex analysis would analyze the savings assuming different outdoor air temperatures for certain times of the year, month or day.

From consideration of the problem we define the following:

Q_c = heat transfer, condenser

Q_e = heat transfer, evaporator

Q_r = heat transfer, resistance heater

A_c = area of condenser

A_e = area of evaporator

W = work input to compressor

\dot{m} = air mass flow rate (5 kg / s)

C_p = specific heat, air (1.0 kJ / kg)

t_e = working fluid temperature, evaporator

t_c = working fluid temperature, condenser

t_{aci} = inlet air temperature, condenser (0°C)
 t_{aco} = outlet air temperature, condenser
 t_{aei} = inlet air temperature, evaporator (24°C)
 t_{aeo} = outlet air temperature, evaporator
 t_{appe} = temperature of approach, evaporator
 t_{appc} = temperature of approach, condenser
 COP = coefficient of performance
 C_{hp} = cost of heat pump
 C_{ehp} = cost of electricity for heat pump
 C_{erh} = cost of electricity for resistance heater
 C_{total} = total cost

Some of these will be design variables and some will be design functions. In order to better understand these relationships, it will be helpful to write the equations for the model.

9.3 Definition of Optimization Problem

The definition of the optimization problem is not always obvious. In this case it is clear what the objective should be: to minimize overall costs (cost of heat pump and cost of energy over ten years). We also note we will have constraints on the temperature of approach in the evaporator and condenser. However, it is not clear what the design variables should be. For example, we need to determine $t_{aco}, t_{aeo}, t_e, t_c, Q_e, Q_c, A_e, A_c$ and W .

Not all of these can be design variables, however, because once some of these are set, others can be calculated from the modeling equations. Some modeling equations essentially act as equality constraints and can be used to eliminate variables (as discussed in Section 6.1.1)

For now we will define the optimization problem as follows,

Find $t_{aco}, t_{aeo}, t_e, t_c, Q_e, Q_c, A_e, A_c$ and W

To Minimize C_{total}

s.t. $t_{appe} > 4$
 $t_{appc} > 4$

In the next sections, where we define the model and equation sequence, we will be able to refine this definition further.

9.4 Model Development

The following equations will apply to the heat pump model,

Equation	Explanation
$Q_c = Q_e + W$	This equation comes from an energy balance across the heat pump. During steady state operation, the heat transferred at the condenser is equal to the heat absorbed at the evaporator and the energy input to the compressor.
$Q_c = UA_c \Delta T_c$	This gives the heat transferred at the condenser in terms of the heat transfer coefficient, area and temperature difference.
ΔT_c	This is the temperature difference across the condenser and is calculated as the log mean temperature difference: $\Delta T_c = \frac{\{(t_c - t_{aci}) - (t_c - t_{aco})\}}{\ln \{(t_c - t_{aci}) / (t_c - t_{aco})\}}$
$Q_c = \dot{m}C_p(t_{aco} - t_{aci})$	The heat transferred at the condenser must equal the sensible heat gain of the air across the condenser.
$Q_e = UA_e \Delta T_e$	The heat transferred at the evaporator.
ΔT_e	The temperature difference across the evaporator: $\Delta T_e = \frac{\{(t_{aei} - t_e) - (t_{aeo} - t_e)\}}{\ln \{(t_{aei} - t_e) / (t_{aeo} - t_e)\}}$
$Q_e = \dot{m}C_p(t_{aei} - t_{aeo})$	The heat transferred at the evaporator must equal the sensible heat loss of the air across the evaporator.
$Q_r = \dot{m}C_p(35 - t_{aco})$	Heat transfer to air by resistance heater.
$COP = \frac{Q_e}{W} = f(t_e, t_c)$	Definition of coefficient of performance (note for this problem it specifies the COP in terms of “refrigeration rate,” which implies Q_e instead of Q_c . The actual equation for the COP is given in (2.37)
$C_{total} = C_{hp} + C_{ehp} + C_{erh}$	We must bring the energy costs for the heat pump and the resistance heater into the present by calculating their present value. Other costs are already in the present. We will use (2.12) to do this: $P = A \frac{(1+i)^n - 1}{i(1+i)^n}$
$t_{appe} = (t_{aeo} - t_e)$	Temperature of approach for the evaporator. This is the difference between the air exit temperature and the working fluid temperature.
$t_{appc} = (t_{aco} - t_c)$	Temperature of approach for the condenser.

It pays to be careful as you develop model equations. An error here can cause lots of grief downstream. One recommendation is to check the units of all equations to make sure the units are correct.

9.5 Equation Sequence

In this step we will determine the order in which equations will be calculated. This will help us determine the set of design variables. It turns out, as we lay out the model equations, we find we only need three design variables. Once these three are set, all the others can be calculated.

We will guess that we have design variables: t_{ace}, t_e, t_c (this guess is not obvious—it sometimes takes some iterations of this step to decide what the most convenient set of variables will be).

Get values of design variables from optimizer: t_{ace}, t_e, t_c ↓	The optimization code will set the values of these variables.
Compute $COP = f(t_e, t_c)$ ↓	From (2.37)
$Q_e = \dot{m}C_p(t_{aei} - t_{aeo})$ $= (5\text{kg/s})(1.00\text{kJ/kg}^\circ\text{C})(24 - t_{aeo})$ ↓	Calculate heat transferred to air from evaporator
$W = Q_e / COP$ ↓	Knowing Q_e and COP the Work can be calculated
$Q_c = Q_e + W$ ↓	Heat at condenser
$t_{aco} = \frac{Q_c}{\dot{m}C_p} + t_{aci}$ ↓	Outlet air temperature of condenser
$A_c = Q_c / U\Delta T_c$ ↓	Area of condenser
$A_e = Q_e / U\Delta T_e$ ↓	Area of evaporator
$C_{hp} = A_e * 100 + A_c * 100 + W * 220$ ↓	Cost of heat pump
$C_{ehp} = W * 0.06 * 4000 * \frac{(1+i)^n - 1}{i(1+i)^n}$ ↓	Present worth of energy cost to run heat pump

$C_{erh} = \dot{m}C_p(35 - t_{aco}) * 0.06 * 4000 * \frac{(1+i)^n - 1}{i(1+i)^n}$ <p style="text-align: center;">↓</p>	Present worth of energy cost to run resistance heater
$C_{total} = C_{hp} + C_{ehp} + C_{erh}$ <p style="text-align: center;">↓</p>	Find total cost
$t_{appe} = (t_{aao} - t_e)$ $t_{appc} = (t_{aco} - t_c)$	Find temperatures of approach

9.6 Model Coding

We are now ready to code the model. Although the code below is for OptdesX, similar coding would need to be done to connect with any optimizer. Note that even though we only have a few design functions, we send a lot of function values back to the optimizer (the calls to `afdsca`) so we can see them in the interface and verify the model is working properly.

```
#include "supportC.h"
#include <math.h>

/*=====
   Function anapreC
   Preprocessing Function
   -----*/
void anapreC( char *modelName )
{
    /* set model name (16 chars max) */
    strcpy( modelName, "Heat Pump" );
}

/*=====
   Function anafunC
   Analysis Function
   -----*/
void anafunC( void )
{
    /* don't forget to declare your variables and functions to be
       double precision */
    double taei, taci, taco, tao, te, tc, Ae, Ac, U, W, Qc, Qe;
    double mdot, cp, delte, deltc, Costelec, Costpump, Costtot;
    double Costresist, COP, appte, apptc, appte2, apptc2;

    /* get AV values from OptdesX (Variable names 16 chars max) */
    avdscaC( &taeo, "evap air temp out" );
    avdscaC( &te, "temp evaporator" );
    avdscaC( &tc, "temp condenser" );

    U = 0.025;
    mdot = 5.;
    cp = 1.00;
    taci = 0.;
    taei = 24.;
```

```

COP = 7.24 + 0.352*te - 0.096*tc - 0.0055*te*tc;

Qe = mdot*cp*(taei - tao);
W = Qe/COP;
Qc = Qe + W;
taco = Qc/(mdot*cp) + taci;

if (fabs ((taei - te) - (taeo - te)) > 0.0001) {
delte = ((taei - te)-(taeo - te))/log((taei - te)/(taeo - te));
}
else {
delte = ((taei - te)+(taeo - te))/2.;
}
if (fabs ((tc - taco) - (tc - taci)) > 0.0001) {
deltc = ((tc - taco)-(tc - taci))/log((tc - taco)/(tc - taci));
}
else {
deltc = ((tc - taco)+(tc - taci))/2.;
}

Ac = Qc/(U*deltc);
Ae = Qe/(U*delte);

Costelec = W*6.144*0.06*4000.;
Costpump = Ae*100. + Ac*100. + W*220.;
Costresist = 0.;

if (taco < 35.) {
Costresist = mdot*cp*(35 - taco)*6.144*0.06*4000.;
}
Costtot = Costresist + Costelec + Costpump;

appte = tao - te;
apptc = tc - taco;
appte2 = taei - te;
apptc2 = tc - taci;

afdscaC( COP, "COP" );
afdscaC( Qe, "Q evaporator" );
afdscaC( Qc, "Q condenser" );
afdscaC( W, "Work" );
afdscaC( delte, "delte" );
afdscaC( deltc, "deltc" );
afdscaC( Ac, "Area condenser" );
afdscaC( Ae, "Area evaporator" );
afdscaC( Costelec, "Cost electricity" );
afdscaC( Costpump, "Cost heat pump" );
afdscaC( Costresist, "Cost resist heat" );
afdscaC( Costtot, "Cost total" );
afdscaC( apptc, "temp app cond" );
afdscaC( appte, "temp app evap" );
afdscaC( appte2, "temp app evap2" );
afdscaC( apptc2, "temp app cond2" );
}
/*=====
Function anaposC
Postprocessing Function
-----*/
void anaposC( void )
{

```

}

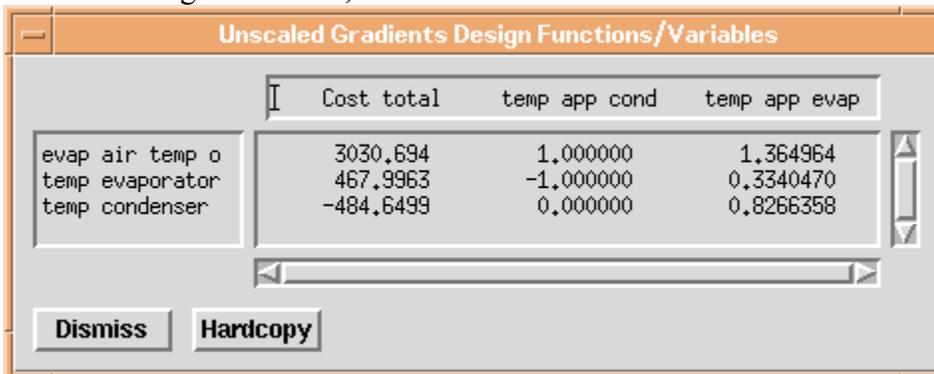
9.7 Linking and Verifying the Model

After we have coded and debugged the model, we link it to the optimizer and begin verification. (In some situations we might be able to verify the model before we link to the optimizer. In such a case we would be able to execute the model separately.) Verification involves checking the model at various points to make sure it is working properly. The values of the design variables are changed manually, and the compute functions button is pushed (causing OptdesX to call the model) to look at the corresponding function values. In this case, I spent over two hours debugging the model. Initially I noticed that no matter what the starting design was, the cost would evaluate to be higher for small perturbations of the design variable, t_{aeo} , regardless of whether I perturbed it smaller or larger. The problem was finally traced down to the fact I was using “abs” for absolute value instead of “fabs” to get the absolute value of a real expression. This was causing erratic results in the model (“abs” is used to get the absolute value of integer expressions). I also discovered I had accidentally switched t_{appe} with t_{appc} . I did some “brute force” debugging to find these errors: I put in several `printf` statements and printed out lots of values until I could track them down.

9.8 Optimizing the Model

After debugging and checking the model at various points, we are finally ready to optimize. The first step is to make sure the scaled gradients (derivatives) look good.

The unscaled gradients are,



	Cost total	temp app cond	temp app evap
evap air temp o	3030.694	1.000000	1.364964
temp evaporator	467.9963	-1.000000	0.3340470
temp condenser	-484.6499	0.000000	0.8266358

Fig. 2.16 Unscaled derivatives.

We see that the derivatives vary by almost five orders of magnitude, from 0.33 to 3030. This can have the effect of slowing the search for an optimum or even causing premature termination. As discussed previously, OptdesX scales the gradients according to the Min, Max, Allowable and Indifference values which are provided by the user. We check the scaled gradients,

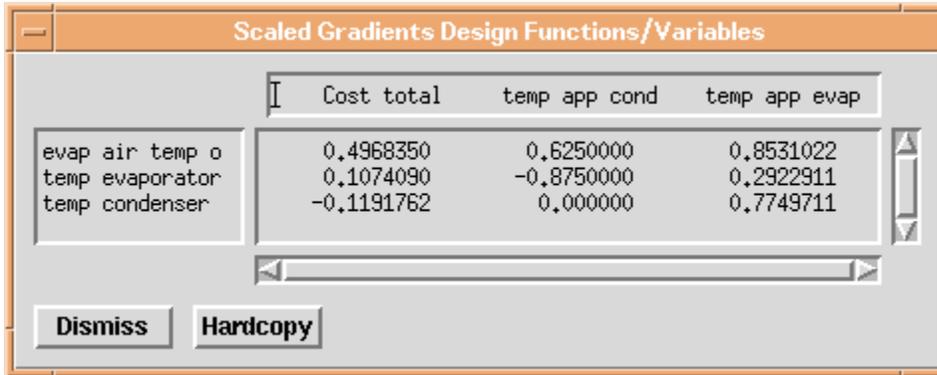


Fig. 2.17 Scaled derivatives.

The scaling is much more uniform.

The starting design is shown in Figs. 2.18 and 2.19 below:

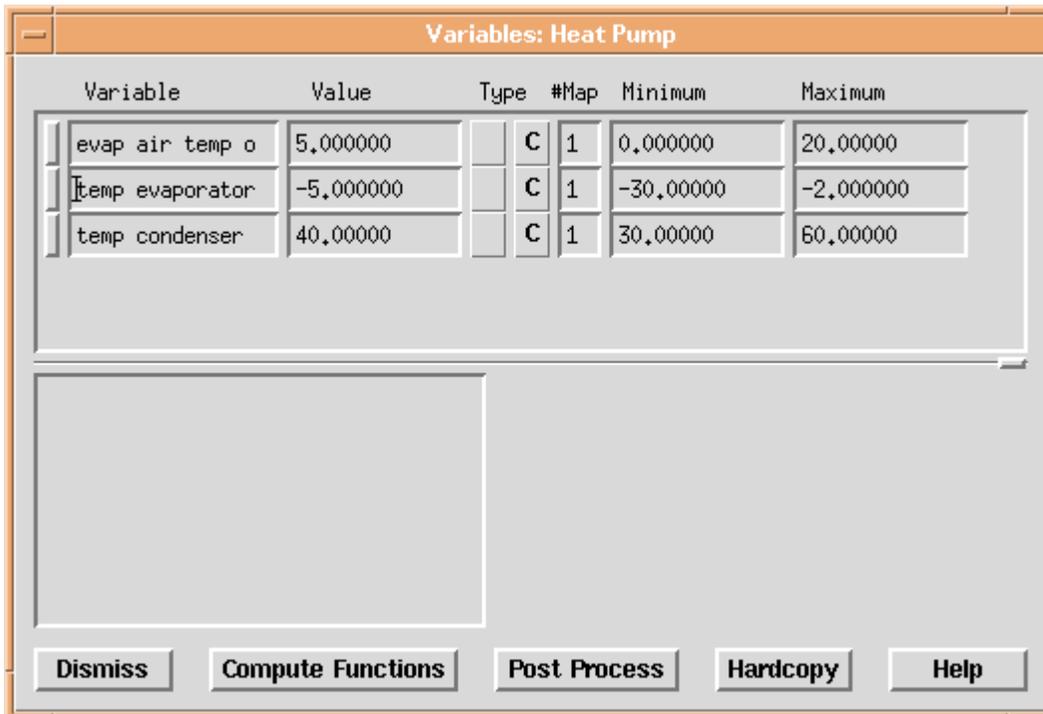


Fig. 2.18 Starting values of design variables.

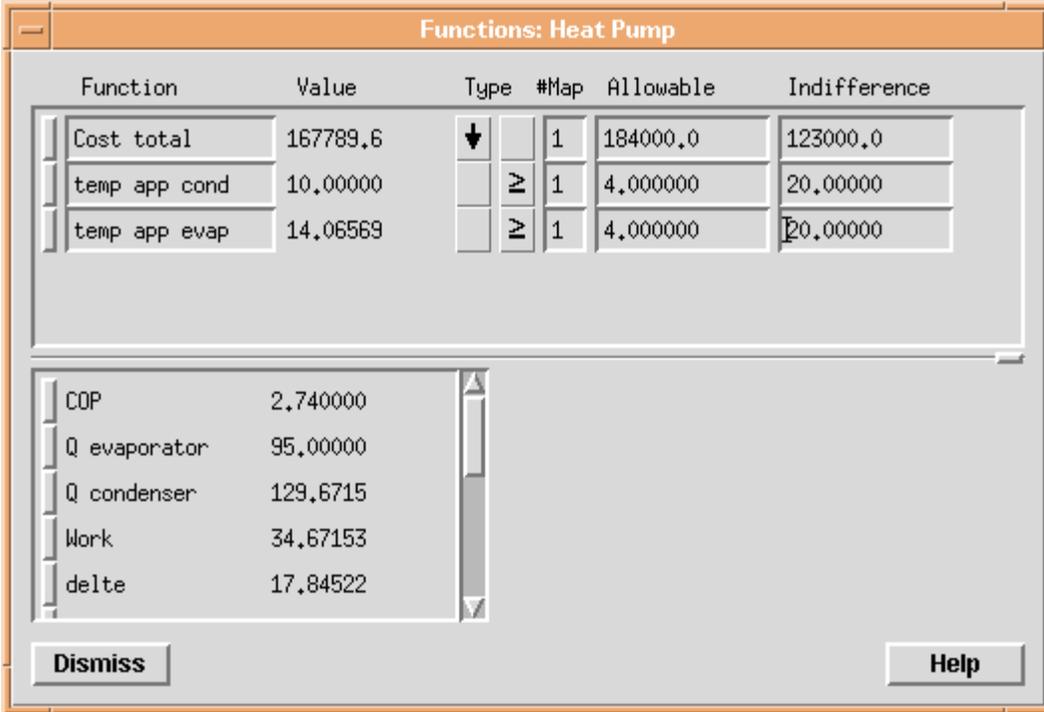


Fig. 2.19. Starting values for functions.

We are ready to optimize. We run the default algorithm GRG. The variables change to:

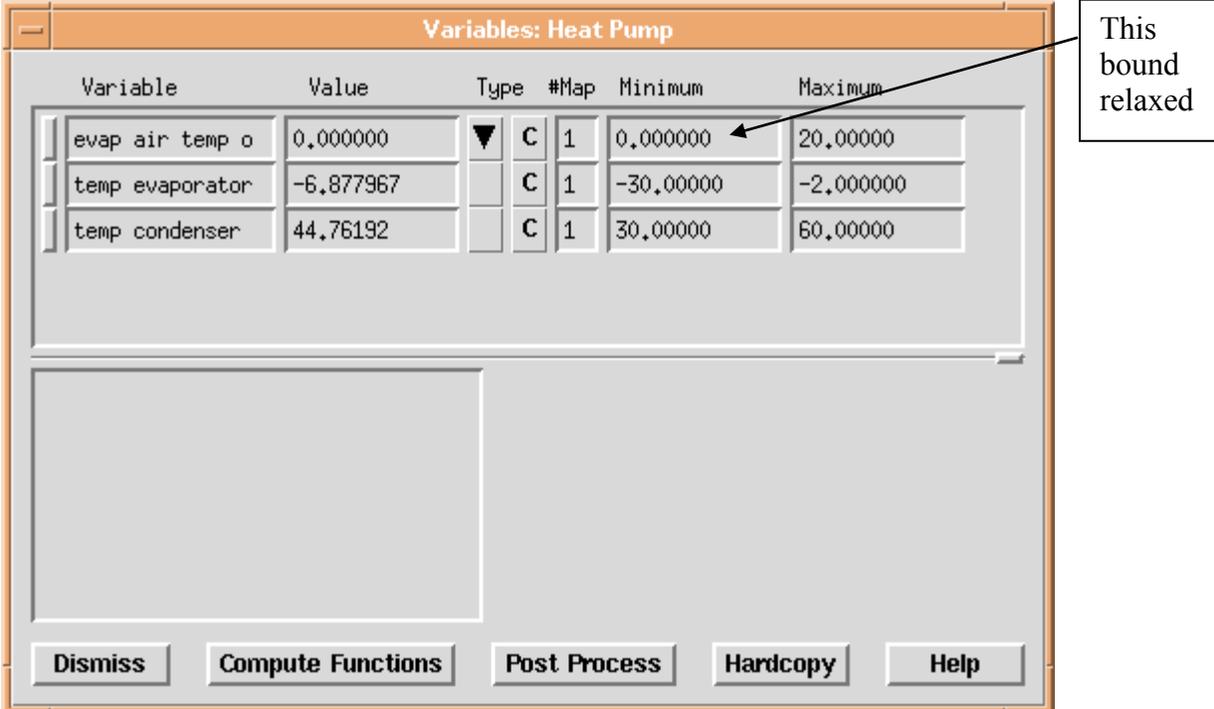


Fig. 2.20 Optimum values for variables.

We see that a design variable, evaporator air outlet temperature, has hit its lower bound of zero. We relax this bound somewhat to see if we can do better:

Variable	Value	Type	#Map	Minimum	Maximum
evap air temp o	-0,06417466	C	1	-5,000000	20,00000
temp evaporator	-6,936127	C	1	-30,00000	-2,000000
temp condenser	44,85551	C	1	30,00000	60,00000

Buttons: Dismiss, Compute Functions, Post Process, Hardcopy, Help

Fig. 2.21 Optimum values for variables with relaxed bound.

The functions at the optimum are given in Fig. 2.22.

Function	Value	Type	#Map	Allowable	Indifference
Cost total	153007,7	↓	1	184000,0	123000,0
temp app cond	6,871953	IV	1	4,000000	20,00000
temp app evap	9,870615	IV	1	4,000000	20,00000

COP	2,203534
Q evaporator	120,3209
Q condenser	174,9245
Work	54,60360
delte	15,99505

Buttons: Dismiss, Help

Fig. 2.22 Function values at optimum.

We see that the total cost has decreased from \$167,800 to \$153,000. Although not shown in Fig. 2.22, the cost of the resistance heating has dropped to nearly zero. Does this make

sense? We see that the COP at the optimum is 2.2, meaning for every unit of energy input to the compressor, 2.2 units of heat are delivered at the condenser. So we would expect the heat pump to be a cost effective means of delivering heat. We might continue to explore this solution to make sure we understand it.

A contour of the space around the optimum is given in Fig 2.23. We see that for this problem the solution is a relatively shallow unconstrained optimum.

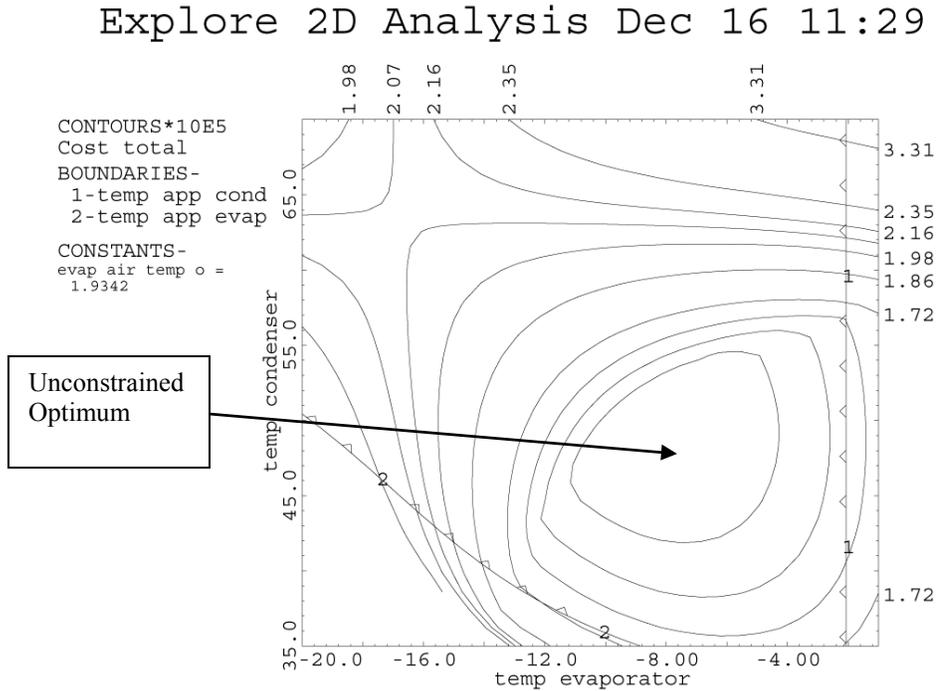


Fig 2.23 Contour plot for the heat pump.

CHAPTER 3

UNCONSTRAINED OPTIMIZATION

1. Preliminaries

1.1. Introduction

In this chapter we will examine some theory for the optimization of unconstrained functions. We will assume all functions are continuous and differentiable. Although most engineering problems are constrained, much of constrained optimization theory is built upon the concepts and theory presented in this chapter.

1.2. Notation

We will use lower case italics, e.g., x , to represent a scalar quantity. Vectors will be represented by lower case bold, e.g., \mathbf{x} , and matrices by upper case bold, e.g., \mathbf{H} .

The set of n design variables will be represented by the n -dimensional vector \mathbf{x} . For example, previously we considered the design variables for the Two-bar truss to be represented by scalars such as diameter, d , thickness, t , height, h ; now we consider diameter to be the first element, x_1 , of the vector \mathbf{x} , thickness to be the second element, x_2 , and so forth. Thus for any problem the set of design variables is given by \mathbf{x} .

Elements of a vector are denoted by subscripts. Values of a vector at specific points are denoted by superscripts. Typically \mathbf{x}^0 will be the starting vector of values for the design variables. We will then move to \mathbf{x}^1 , \mathbf{x}^2 , until we reach the optimum, which will be \mathbf{x}^* . A summary of notation used in this chapter is given in Table 1.

Table 1 Notation

\mathbf{A}	Matrix \mathbf{A}	$\mathbf{x}, \mathbf{x}^k, \mathbf{x}^*$	Vector of design variables, vector at iteration k , vector at the optimum
\mathbf{I}	Identity matrix	$x_1, x_2 \dots x_n$	Elements of vector \mathbf{x}
\mathbf{a}	Column vector	\mathbf{s}, \mathbf{s}^k	Search direction, search direction at iteration k
$\mathbf{a}_i \quad i = 1, 2, \dots$	Columns of \mathbf{A}	$\alpha, \alpha^k, \alpha^*$	Step length, step length at iteration k , step length at minimum along search direction
$\mathbf{e}_i \quad i = 1, 2, \dots$	Coordinate vectors (columns of \mathbf{I})	$f(\mathbf{x}), f(\mathbf{x}^k), f^k$	Objective function, objective evaluated at \mathbf{x}^k
$\mathbf{A}^T, \mathbf{a}^T$	transpose	$\nabla^2 f(\mathbf{x}^k), \nabla^2 f^k$ $\mathbf{H}(\mathbf{x}^k), \mathbf{H}^k$	Hessian matrix at \mathbf{x}^k
$\nabla f(\mathbf{x}), \nabla f(\mathbf{x}^k), \nabla f^k$	Gradient of $f(\mathbf{x})$, gradient evaluated at \mathbf{x}^k	$ \mathbf{A} $	Determinant of \mathbf{A}

$\Delta \mathbf{x}^k = \mathbf{x}^{k+1} - \mathbf{x}^k$	Difference in \mathbf{x} vectors	$\mathbf{x} \in R^n$	All vectors which are in n-dimensional Euclidean space
$\boldsymbol{\gamma}^k = \nabla f^{k+1} - \nabla f^k$	Difference in gradients at $\mathbf{x}^{k+1}, \mathbf{x}^k$	\mathbf{N}^k	Direction matrix at \mathbf{x}^k

1.3. Statement of Problem

The problem we are trying to solve in this chapter can be stated as,

$$\begin{aligned} &\text{Find } \mathbf{x}, \quad \mathbf{x} \in R^n \\ &\text{To Minimize } f(\mathbf{x}) \end{aligned}$$

1.4. Gradient Vector

1.4.1. Definition

The gradient of $f(\mathbf{x})$ is denoted $\nabla f(\mathbf{x})$. The gradient is defined as a column vector of the first partial derivatives of $f(\mathbf{x})$:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \quad (3.1)$$

1.4.2. Example: Gradient of a Function

Evaluate the gradient of the function $f(\mathbf{x}) = 6 - 2x_1 + x_2 + 2x_1^2 + 3x_1x_2 + x_2^2$

$$\nabla f = \begin{bmatrix} -2 + 4x_1 + 3x_2 \\ 1 + 3x_1 + 2x_2 \end{bmatrix} \text{ If evaluated at } \mathbf{x}^0 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}, \nabla f^0 = \begin{bmatrix} -4 \\ -1 \end{bmatrix}$$

A very important property of the gradient vector is that it is *orthogonal to the function contours and points in the direction of greatest increase of a function*. The negative gradient points in the direction of greatest decrease. Any vector \mathbf{v} which is orthogonal to $\nabla f(\mathbf{x})$ will satisfy $\mathbf{v}^T \nabla f(\mathbf{x}) = 0$.

1.5. Vectors That Point "Downhill" or "Uphill"

If we have some search direction \mathbf{s} , then $\mathbf{s}^T \nabla f$ is proportional to the projection of \mathbf{s} onto the gradient vector. We can see this better in Fig. 3.1:

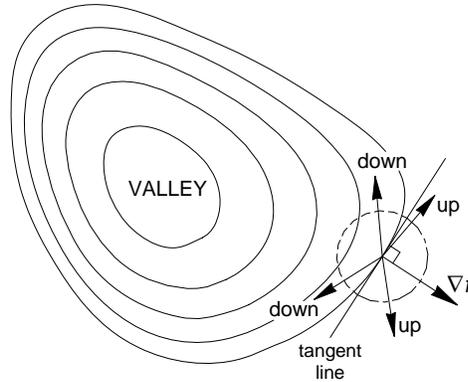


Fig. 3.1. Vectors that point uphill or downhill.

As long as $\mathbf{s}^T \nabla f > 0$, then \mathbf{s} points, at least for some small distance, in a direction that increases the function (points uphill). In like manner, if $\mathbf{s}^T \nabla f < 0$, then \mathbf{s} points downhill. As an example, suppose at the current point in space the gradient vector is $\nabla f(\mathbf{x}^k)^T = [6, 1, -2]$. We propose to move from this point in a search direction $\mathbf{s}^T = [-1, 1, 0]$.

Does this direction go downhill? We evaluate $\mathbf{s}^T \nabla f = [-1, -1, 0] \begin{bmatrix} 6 \\ 1 \\ -2 \end{bmatrix} = -7$

So this direction would take us downhill, at least for a short step.

1.6. Hessian Matrix

1.6.1. Definition

The Hessian Matrix, $\mathbf{H}(\mathbf{x})$ or $\nabla^2 f(\mathbf{x})$ is defined to be the square matrix of second partial derivatives:

$$\mathbf{H}(\mathbf{x}) = \nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_n} & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \dots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \dots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \dots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \quad (3.2)$$

We can also obtain the Hessian by applying the gradient operator on the gradient transpose,

$$\mathbf{H}(\mathbf{x}) = \nabla^2 f(\mathbf{x}) = \nabla(\nabla f(\mathbf{x})^T) = \begin{bmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \vdots \\ \frac{\partial}{\partial x_n} \end{bmatrix} \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \dots & \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_n} & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \dots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \dots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \dots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

The Hessian is a symmetric matrix. The Hessian matrix gives us information about the curvature of a function, and tells us how the gradient is changing.

For simplicity, we will sometimes write \mathbf{H}^k instead of $\mathbf{H}(\mathbf{x}^k)$.

1.6.2. Example: Hessian Matrix

Find the Hessian matrix for the function, $f(x) = 6 - 2x_1 + x_2 + 2x_1^2 + 3x_1x_2 + x_2^2$

$$\nabla f = \begin{bmatrix} -2 + 4x_1 + 3x_2 \\ 1 + 3x_1 + 2x_2 \end{bmatrix}, \quad \begin{aligned} \frac{\partial^2 f}{\partial x_1^2} &= 4 & \frac{\partial^2 f}{\partial x_1 \partial x_2} &= 3 \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} &= 3 & \frac{\partial^2 f}{\partial x_2^2} &= 2 \end{aligned}$$

and the Hessian is:

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}$$

1.7. Positive and Negative Definiteness

1.7.1. Definitions

If for any vector, \mathbf{x} , the following is true for a symmetric matrix \mathbf{B} ,

$$\begin{aligned} \mathbf{x}^T \mathbf{B} \mathbf{x} > 0 & \text{ then } B \text{ is positive definite} \\ \mathbf{x}^T \mathbf{B} \mathbf{x} < 0 & \text{ then } B \text{ is negative definite} \end{aligned} \tag{3.3}$$

1.7.2. Checking Positive Definiteness

The above definition is not very useful in terms of checking if a matrix is positive definite, because it would require that we examine every possible vector \mathbf{x} to see if the condition given in (3.3) is true. So, how can we tell if a matrix is positive definite? There are three ways we will mention,

1. A symmetric matrix \mathbf{B} is positive definite if all eigenvalues of \mathbf{B} are positive.
2. A symmetric matrix is positive definite if and only if the determinant of each of its principal minor matrices is positive.
3. A $n \times n$ matrix \mathbf{B} is symmetric and positive definite if and only if it can be written as $\mathbf{B} = \mathbf{L}\mathbf{L}^T$ where \mathbf{L} is a lower triangular matrix with positive diagonal elements. The \mathbf{L} matrix can be developed through Choleski decomposition.

The matrix we will be most interested in checking is the Hessian matrix, $\mathbf{H}(\mathbf{x})$

What does it mean for the Hessian to be positive or negative definite? If positive definite, it means curvature of the function is everywhere positive. This will be an important condition for checking if we have a minimum. If negative definite, curvature is everywhere negative. This will be a condition for verifying we have a maximum.

1.7.3. Example: Checking if a Matrix is Positive Definite Using Principal Minor Matrices

Is the matrix given below positive definite? We need to check the determinants of the principal minor matrices, found by taking the determinant of a 1x1 matrix along the diagonal, the determinant of a 2x2 matrix along the diagonal, and finally the determinant of the entire matrix. If any one of these determinants is not positive, the matrix is not positive definite.

$$\begin{bmatrix} 2 & 3 & -2 \\ 3 & 5 & -1 \\ -2 & -1 & 5 \end{bmatrix}$$

$$|[2]| = 2 > 0 \quad \left| \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix} \right| = 1 > 0 \quad \left| \begin{bmatrix} 2 & 3 & -2 \\ 3 & 5 & -1 \\ -2 & -1 & 5 \end{bmatrix} \right| = -5 < 0$$

The determinants of the first two principal minors are positive. However, because the determinant of the matrix as a whole is negative, this matrix is not positive definite.

We also note that the eigenvalues are -0.15, 4.06, 8.09. That these are not all positive also indicates the matrix is not positive definite.

1.7.4. Checking Negative Definiteness

How can we check to see if a matrix is negative definite? There are two ways we will mention,

1. A symmetric matrix \mathbf{B} is negative definite if all eigenvalues of \mathbf{B} are negative.

2. A symmetric matrix is negative definite if we reverse the sign of each element and the resulting matrix is positive definite.

Note: A symmetric matrix is not negative definite if the determinant of each of its principal minor matrices is negative. Rather, in the negative definite case, the signs of the determinants alternate minus and plus, so the easiest way to check for negative definiteness using principal minor matrices is to reverse all signs and see if the resulting matrix is positive definite.

It is also possible for a matrix to be positive *semi*-definite, or negative *semi*-definite. This occurs when one or more of the determinants or eigenvalues are equal to zero, and the others are all positive (or negative, as the case may be). These are special cases we won't worry about here.

If a matrix is neither positive definite nor negative definite (nor semi-definite) then it is *indefinite*. If using principal minor matrices, note that we need to check both cases before we reach a conclusion that a matrix is indefinite.

1.7.5. Example: Checking if a Matrix is Negative Definite Using Principal Minor Matrices

Is the matrix given above negative definite? We reverse the signs and see if the resulting matrix is positive definite:

$$\begin{bmatrix} -2 & -3 & 2 \\ -3 & -5 & 1 \\ 2 & 1 & -5 \end{bmatrix}$$

$$|[-2]| = -2 < 0$$

Because the first determinant is negative there is no reason to go further. We also note that the eigenvalues of the “reversed sign” matrix are not all positive.

Because this matrix is neither positive nor negative definite, it is indefinite.

1.8. Taylor Expansion

1.8.1. Definition

The Taylor expansion is an approximation to a function at a point \mathbf{x}^k and can be written in vector notation as:

$$f(\mathbf{x}) = f(\mathbf{x}^k) + (\nabla f(\mathbf{x}^k))^T (\mathbf{x} - \mathbf{x}^k) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^k)^T \nabla^2 f(\mathbf{x}^k) (\mathbf{x} - \mathbf{x}^k) + \dots \quad (3.4)$$

If we note that $\mathbf{x} - \mathbf{x}^k$ can be written as $\Delta \mathbf{x}^k$, and using notation $f(\mathbf{x}^k) = f^k$, we can write (3.4) more compactly as,

$$f^{k+1} = f^k + (\nabla f^k)^T \Delta \mathbf{x}^k + \frac{1}{2} (\Delta \mathbf{x}^k)^T \nabla^2 f^k \Delta \mathbf{x}^k + \dots \quad (3.5)$$

The Taylor expansion allows us to approximate any continuous function as a polynomial in terms of its derivatives at the point \mathbf{x}^k . We can make a linear approximation by taking the first two terms of the expansion. We can make a quadratic approximation by taking the first three terms of the expansion.

1.8.2. Example: Quadratic Approximation of a Transcendental Function

Suppose $f(\mathbf{x}) = 2(x_1)^{1/2} + 3\ln(x_2)$

$$\text{at } (\mathbf{x}^k)^T = [5, 4] \quad \nabla f^T = \left[x_1^{(-1/2)}, \frac{3}{x_2} \right] \quad (\nabla f^k)^T = [0.447, 0.750]$$

$$\frac{\partial^2 f}{\partial x_1^2} = -\frac{1}{2} x_1^{(-3/2)} \quad \frac{\partial^2 f}{\partial x_1 \partial x_2} = 0 \quad \frac{\partial^2 f}{\partial x_2 \partial x_1} = 0 \quad \frac{\partial^2 f}{\partial x_2^2} = \frac{-3}{x_2^2}$$

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} -\frac{1}{2} x_1^{(-3/2)} & 0 \\ 0 & \frac{-3}{x_2^2} \end{bmatrix} \text{ at } \begin{bmatrix} 5 \\ 4 \end{bmatrix} = \begin{bmatrix} -0.045 & 0.0 \\ 0.0 & -0.188 \end{bmatrix}$$

$$f(\mathbf{x}) \approx 8.631 + [0.447, 0.750] \begin{bmatrix} x_1 - 5 \\ x_2 - 4 \end{bmatrix} + \frac{1}{2} [x_1 - 5, x_2 - 4] \begin{bmatrix} -0.045 & 0.0 \\ 0.0 & -0.188 \end{bmatrix} \begin{bmatrix} x_1 - 5 \\ x_2 - 4 \end{bmatrix}$$

If we wish, we can stop here with the equation in vector form. To see the equation in scalar form we can carry out the vector multiplications and combine similar terms:

$$f(\mathbf{x}) \approx 8.631 + 0.447x_1 - 2.235 + 0.750x_2 - 3.000 +$$

$$\frac{1}{2} [(-0.045x_1 + 0.225), (-0.188x_2 + 0.752)] \begin{bmatrix} x_1 - 5 \\ x_2 - 4 \end{bmatrix}$$

$$f(\mathbf{x}) \approx 3.396 + 0.447x_1 + 0.750x_2 +$$

$$\frac{1}{2} (-0.045x_1^2 + 0.450x_1 - 1.125 - 0.188x_2^2 + 1.504x_2 - 3.008)$$

$$f(\mathbf{x}) \approx 1.300 + 0.672x_1 + 1.502x_2 - 0.023x_1^2 - 0.094x_2^2$$

Evaluating and comparing this approximation to the original:

$[\mathbf{x}]^T$	Quadratic	Actual	Error
[5,4]	8.63	8.63	0.00
[5,5]	9.28	9.3	0.02
[6,4]	9.05	9.06	0.01
[7,6]	10.55	10.67	0.12
[2,1]	3.98	2.83	-1.15
[9,2]	8.19	8.08	-0.11

We notice that the further the point gets from the expansion point, the greater the error that is introduced. We also see that at the point of expansion the approximation is exact.

2. Properties and Characteristics of Quadratic Functions

A lot of optimization theory is based on optimizing quadratic functions. It is therefore helpful to investigate some of the properties of these functions.

2.1. Representation

We can represent a quadratic function three ways—as a scalar equation, a general vector equation, and as a Taylor expansion. Although these representations look different, they give exactly the same results. For example, consider the equation,

$$f(x) = 6 - 2x_1 + x_2 + 2x_1^2 + 3x_1x_2 + x_2^2 \quad (3.6)$$

This is a scalar representation of a quadratic.

As a another representation, we can write a quadratic in general vector form,

$$f(\mathbf{x}) = a + \mathbf{b}^T \mathbf{x} + \frac{1}{2} \mathbf{x}^T \mathbf{C} \mathbf{x} \quad (3.7)$$

By inspection, the example given in (3.6), in the form of (3.7), is:

$$f(\mathbf{x}) = 6 + [-2, 1] \mathbf{x} + \frac{1}{2} \mathbf{x}^T \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix} \mathbf{x} \quad (3.8)$$

where,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

We also observe that \mathbf{C} in (3.7) ends up being \mathbf{H} .

A third form is a Taylor representation,

$$f(\mathbf{x}) = f^k + (\nabla f^k)^T \Delta \mathbf{x}^k + \frac{1}{2} (\Delta \mathbf{x}^k)^T \mathbf{H} \Delta \mathbf{x}^k \quad (3.9)$$

We note for (3.6), $\nabla f = \begin{bmatrix} -2 + 4x_1 + 3x_2 \\ 1 + 3x_1 + 2x_2 \end{bmatrix}$ and $\mathbf{H} = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}$

We will assume a point of expansion, $\mathbf{x}^k = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$, $\nabla f^k = \begin{bmatrix} -4 \\ -1 \end{bmatrix}$. (It may not be apparent, but if we are approximating a quadratic, it doesn't matter what point of expansion we assume. The Taylor expansion will be exact.)

The example in (3.6), as a Taylor representation, becomes,

$$f(\mathbf{x}) = 12 + [-4, -1] \Delta \mathbf{x} + \frac{1}{2} \Delta \mathbf{x}^T \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix} \Delta \mathbf{x} \quad (3.10)$$

where,

$$\Delta \mathbf{x} = \begin{bmatrix} x_1 + 2 \\ x_2 - 2 \end{bmatrix}$$

These three representations are equivalent. If we pick the point $\mathbf{x}^T = [1.0, 2.0]$, all three representations give $f = 18$ at this point, as you can verify by substitution.

2.2. Characteristics of Quadratic Functions

It is useful to note the following characteristics of quadratic equations:

- The equations for the gradient vector of a quadratic function are linear. This makes it easy to solve for where the gradient is equal to zero.
- The Hessian for a quadratic function is a matrix of constants (so we will write as \mathbf{H} or $\nabla^2 f$ instead of $\mathbf{H}(\mathbf{x})$ or $\nabla^2 f(\mathbf{x})$). Thus the curvature of a quadratic is everywhere the same.
- Excluding the cases where we have a semi-definite Hessian, quadratic functions have only one *stationary point*, i.e. only one point where the gradient is zero.
- Given the gradient and Hessian at some point \mathbf{x}^k , the gradient at some other point, \mathbf{x}^{k+1} , is given by,

$$\nabla f^{k+1} = \nabla f^k + \mathbf{H}(\mathbf{x}^{k+1} - \mathbf{x}^k) \quad (3.11)$$

This expression is developed in Section 9.1 of the Appendix by differentiating a Taylor expansion in vector form.

- Given the gradient some point \mathbf{x}^k , Hessian, \mathbf{H} , and a search direction, \mathbf{s} , the optimal step length, α^* , in the direction \mathbf{s} is given by,

$$\alpha^* = -\frac{(\nabla f^k)^T \mathbf{s}}{\mathbf{s}^T \mathbf{H} \mathbf{s}} \quad (3.12)$$

This expression is derived in Section 9.2 of the Appendix.

- The best methods of optimization are *methods of conjugate directions*. A method of conjugate directions will solve for the optimum of a quadratic function of n variables in n steps, providing minimizing steps are taken in each search direction. We will learn more about these methods in sections which follow.

2.3. Examples

We start with the example,

$$f(\mathbf{x}) = 4x_1 + 2x_2 + x_1^2 - 4x_1x_2 + x_2^2 \quad (3.13)$$

Since this is a quadratic, we know it only has one stationary point. We note that the Hessian,

$$\mathbf{H} = \begin{bmatrix} 2 & -4 \\ -4 & 2 \end{bmatrix}$$

is indefinite (eigenvalues are -0.16 and 6.1). This means we should have a saddle point. The contour plots in Fig 3.2 and Fig. 3.3 confirm this.

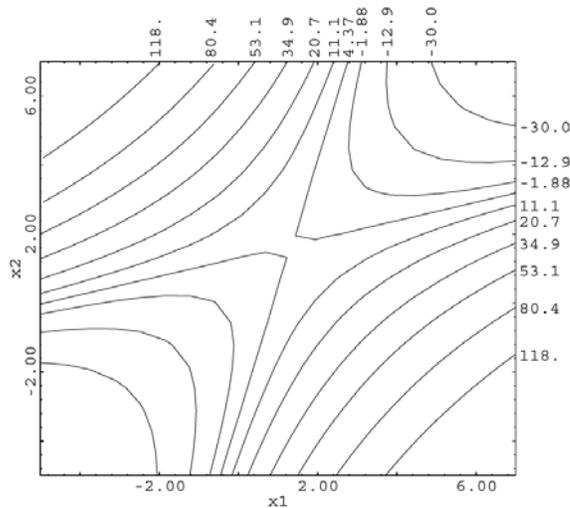


Fig. 3.2 Contour plot of Eq (3.13).

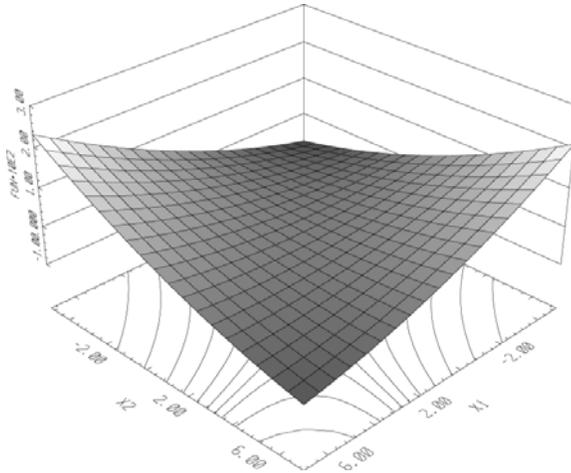


Fig. 3.3. 3D contour plot of (3.13).

We will do a second example. Suppose we have the function,

$$f(\mathbf{x}) = x_1 + 2x_2 + 4x_1^2 - x_1x_2 + 2x_2^2 \quad (3.14)$$

$$\nabla f = \begin{bmatrix} 1 + 8x_1 - x_2 \\ 2 - x_1 + 4x_2 \end{bmatrix} \text{ and } \mathbf{H} = \begin{bmatrix} 8 & -1 \\ -1 & 4 \end{bmatrix}$$

By inspection, we see that the determinants of the principal minor matrices are all positive. Thus this function should have a min and look like a bowl. The contour plots follow.

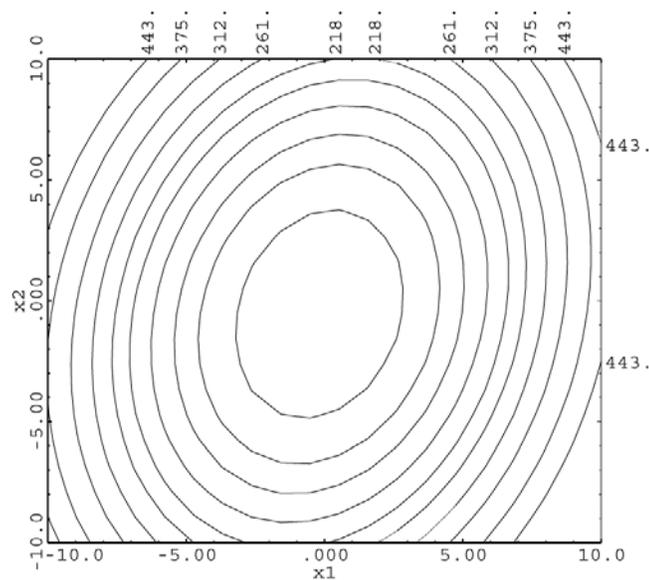


Fig. 3.4. Contour plot for (3.14)

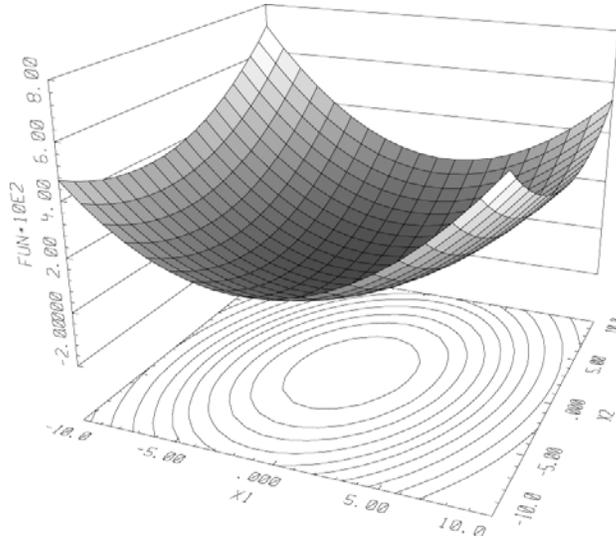


Fig. 3.5 3D contour plot for (3.14)

3. Necessary and Sufficient Conditions for an Unconstrained Optimum

With some preliminaries out of the way, we are now ready to begin discussing the theory of unconstrained optimization of differentiable functions. We start with the mathematical conditions which must hold at an unconstrained, local optimum.

3.1. Definitions

3.1.1. Necessary Conditions for an Unconstrained Optimum

The necessary conditions for an unconstrained optimum at \mathbf{x}^* are,

$$\nabla f(\mathbf{x}^*) = 0 \text{ and } f(\mathbf{x}) \text{ be differentiable at } \mathbf{x}^* \quad (3.15)$$

These conditions are necessary but not *sufficient*, inasmuch as $\nabla f(\mathbf{x}) = 0$ can apply at a max, min or a saddle point. However, if at a point $\nabla f(\mathbf{x}) \neq 0$, then that point *cannot* be an optimum.

3.1.2. Sufficient Conditions for a Minimum

The sufficient conditions include the necessary conditions but add other conditions such that when satisfied we *know* we have an optimum. For a minimum,

$$\nabla f(\mathbf{x}^*) = 0, f(\mathbf{x}) \text{ differentiable at } \mathbf{x}^*, \text{ plus } \nabla^2 f(\mathbf{x}^*) \text{ is positive definite.} \quad (3.16)$$

3.1.3. Sufficient Conditions for a Maximum

For a maximum,

$\nabla f(\mathbf{x}^*) = 0$, $f(\mathbf{x})$ differentiable at \mathbf{x}^* , plus $\nabla^2 f(\mathbf{x}^*)$ is negative definite. (3.17)

3.2. Examples: Applying the Necessary, Sufficient Conditions

Apply the necessary and sufficient conditions to find the optimum for the quadratic function,

$$f(\mathbf{x}) = x_1^2 - 2x_1x_2 + 4x_2^2$$

Since this is a quadratic function, the partial derivatives will be linear equations. We can solve these equations directly for a point that satisfies the necessary conditions. The gradient vector is,

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 - 2x_2 \\ -2x_1 + 8x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

When we solve these two equations, we have a solution, $x_1 = 0$, $x_2 = 0$ --this a point where the gradient is equal to zero. This represents a minimum, a maximum, or a saddle point. At this point, the Hessian is,

$$\mathbf{H} = \begin{bmatrix} 2 & -2 \\ -2 & 8 \end{bmatrix}$$

Since this Hessian is positive definite (eigenvalues are 1.4, 8.6), this must be a minimum.

As a second example, apply the necessary and sufficient conditions to find the optimum for the quadratic function,

$$f(\mathbf{x}) = 4x_1 + 2x_2 + x_1^2 - 4x_1x_2 + x_2^2$$

As in example 1, we will solve the gradient equations directly for a point that satisfies the necessary conditions. The gradient vector is,

$$\begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 - 4x_2 + 4 \\ -4x_1 + 2x_2 + 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

When we solve these two equations, we have a solution, $x_1 = 1.333$, $x_2 = 1.667$. The Hessian is,

$$\mathbf{H} = \begin{bmatrix} 2 & -4 \\ -4 & 2 \end{bmatrix}$$

The eigenvalues are -2, 6. The Hessian is indefinite. This means this is neither a max nor a min—it is a saddle point.

Comments: As mentioned, the equations for the gradient for a quadratic function are linear, so they are easy to solve. Obviously we don't usually have a quadratic objective, so the equations are usually not linear. Often we will use the necessary conditions to *check* a point to see if we are at an optimum. Some algorithms, however, solve for an optimum by solving directly where the gradient is equal to zero. Sequential Quadratic Programming (SQP) is this type of algorithm.

Other algorithms search for the optimum by taking downhill steps and continuing until they can go no further. The GRG (Generalized Reduced Gradient) algorithm is an example of this type of algorithm. In the next section we will study one of the simplest unconstrained algorithms that steps downhill: steepest descent.

4. Steepest Descent with a Quadratic Line Search

4.1. Description

One of the simplest unconstrained optimization methods is steepest descent. Given an initial starting point, the algorithm moves downhill until it can go no further.

The search can be broken down into stages. For any algorithm, at each stage (or iteration) we must determine two things:

1. What should the search direction be?
2. How far should we go in that direction?

Answer to question 1: For the method of steepest descent, the search direction is $-\nabla f(\mathbf{x})$

Answer to question 2: A *line search* is performed. "Line" in this case means we search along a direction vector. The line search strategy presented here, bracketing the function with quadratic fit, is one of many that have been proposed, and is one of the most common.

General Approach for each step:

Given some starting point, \mathbf{x}^k , we wish to determine,

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha \mathbf{s} \tag{3.18}$$

where \mathbf{s} is the search direction vector, usually normalized, and α is the step length, a scalar.

We will step in direction \mathbf{s} with increasing values of α until the function starts to get worse. Then we will curve fit the data with a parabola, and step to the minimum of the parabola.

4.2. Example: Steepest Descent with Line Search

$$\begin{aligned} \text{Min } f(\mathbf{x}) &= x_1^2 - 2x_1x_2 + 4x_2^2 & f^0 &= 19 \\ \text{starting at } \mathbf{x}^0 &= \begin{bmatrix} -3 \\ 1 \end{bmatrix} & -\nabla f^0 &= \begin{bmatrix} 8 \\ -14 \end{bmatrix} & \mathbf{s}^0 &= \begin{bmatrix} 8 \\ -14 \end{bmatrix} \\ \text{normalized } \mathbf{s}^0 &= \begin{bmatrix} 0.50 \\ -0.86 \end{bmatrix} & \mathbf{x}^1 &= \begin{bmatrix} -3 \\ 1 \end{bmatrix} + \alpha \begin{bmatrix} 0.50 \\ -0.86 \end{bmatrix} \end{aligned}$$

We will find α^* , which is the optimal step length, by trial and error.

Guess $\alpha^* = .4$ for step number 1:

Line Search Step	α	$\mathbf{x}^1 = \mathbf{x}^0 + \alpha \mathbf{s}^0$	$f(\mathbf{x})$
1	0.4	$\mathbf{x}^1 = \begin{bmatrix} -3.0 \\ 1.0 \end{bmatrix} + .4 \begin{bmatrix} 0.50 \\ -0.86 \end{bmatrix} = \begin{bmatrix} -2.80 \\ 0.66 \end{bmatrix}$	13.3

We see that the function has decreased; we decide to double the step length and continue doubling until the function begins to increase:

Line Search Step	α	$\mathbf{x}^1 = \mathbf{x}^0 + \alpha \mathbf{s}^0$	$f(\mathbf{x})$
2	0.8	$\mathbf{x}^1 = \begin{bmatrix} -3.0 \\ 1.0 \end{bmatrix} + .8 \begin{bmatrix} 0.50 \\ -0.86 \end{bmatrix} = \begin{bmatrix} -2.60 \\ 0.31 \end{bmatrix}$	8.75
3	1.6	$\mathbf{x}^1 = \begin{bmatrix} -3.0 \\ 1.0 \end{bmatrix} + 1.6 \begin{bmatrix} 0.50 \\ -0.86 \end{bmatrix} = \begin{bmatrix} -2.20 \\ -0.38 \end{bmatrix}$	3.74
4	3.2	$\mathbf{x}^1 = \begin{bmatrix} -3.0 \\ 1.0 \end{bmatrix} + 3.2 \begin{bmatrix} 0.50 \\ -0.86 \end{bmatrix} = \begin{bmatrix} -1.40 \\ -1.75 \end{bmatrix}$	9.31

The objective function has started to increase; therefore we have gone too far.

We will cut the change in the last step by half:

5	2.4	$\mathbf{x}^1 = \begin{bmatrix} -3.0 \\ 1.0 \end{bmatrix} + 2.4 \begin{bmatrix} 0.50 \\ -0.86 \end{bmatrix} = \begin{bmatrix} -1.80 \\ -1.06 \end{bmatrix}$	3.91
---	-----	---	------

A graph of our progress is shown in Fig. 3.6:

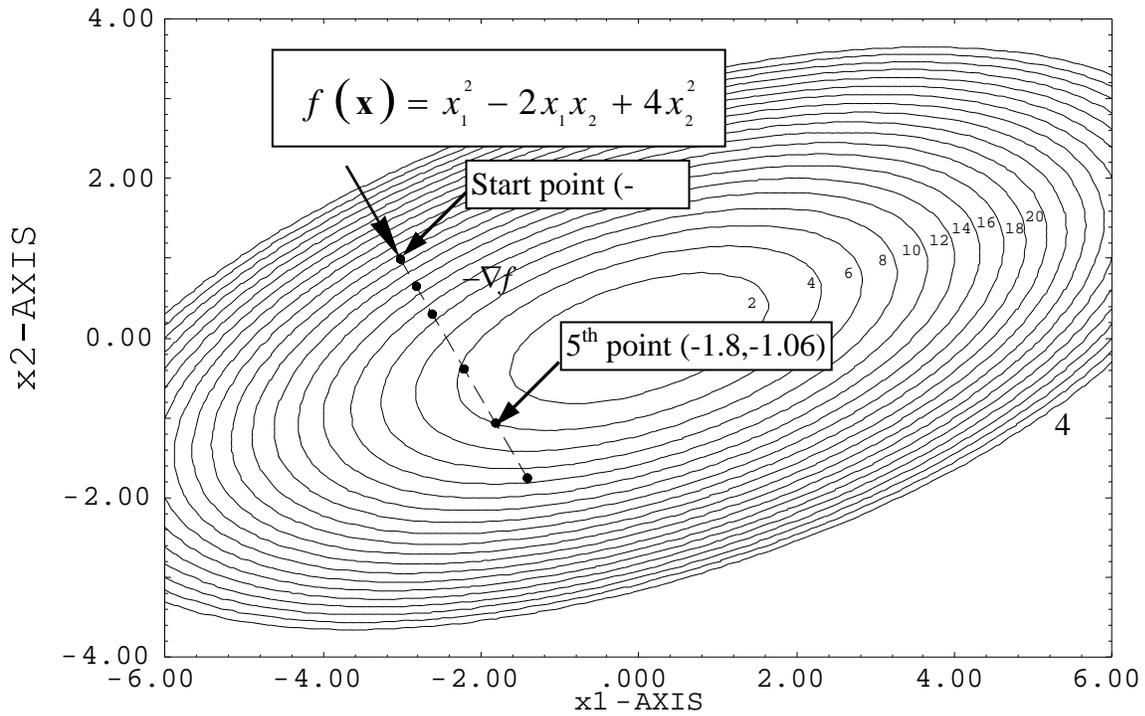


Fig. 3.6 Progress in the line search shown on a contour plot.

If we plot the objective value as a function of step length as shown in Fig 3.7:

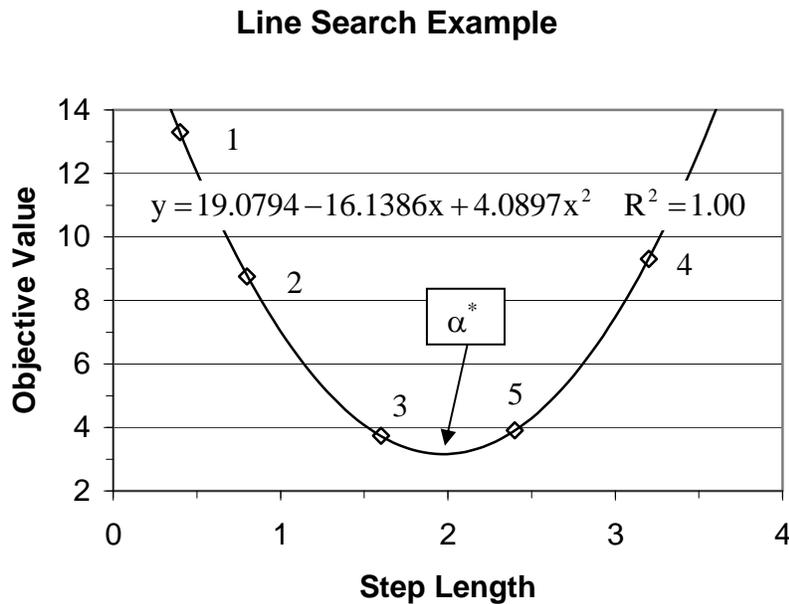


Fig. 3.7 The objective value vs. step length for the line search.

We see that the data plot up to be a parabola. We would like to estimate the minimum of this curve. We will curve fit points 2, 5, 3. These points are equally spaced and bracket the minimum.

$$\|2 \quad \|3 \quad \|5$$

Renumbering these points as $\alpha_1, \alpha_2, \alpha_3$ the minimum of the parabola is given by

$$\begin{aligned}\alpha^* &= \alpha_2 + \frac{\Delta\alpha [f(\alpha_1) - f(\alpha_3)]}{2[f(\alpha_1) - 2f(\alpha_2) + f(\alpha_3)]} \\ \alpha^* &= 1.60 + \frac{(0.8)[8.75 - 3.91]}{2[8.75 - 2(3.74) + 3.91]} \\ \alpha^* &= 1.97\end{aligned}\tag{3.19}$$

where $f(\mathbf{x}) = 3.2$

When we step back, after the function has become worse, we have four points to choose from (points 2, 3, 5, 4). How do we know which three to pick to make sure we don't lose the bracket on the minimum? The rule is this: take the point with the lowest function value (point 3) and the two points to either side (points 2 and 5).

In summary, the line search consists of stepping along the search direction until the minimum of the function in this direction is bracketed, fitting three points which bracket the minimum with a parabola, and calculating the minimum of the parabola. If necessary the parabolic fit can be carried out several times until the change in the minimum is very small (although the α are then no longer equally spaced, so the following formula must be used):

$$\alpha^* = \frac{f(\alpha_1)(\alpha_2^2 - \alpha_3^2) + f(\alpha_2)(\alpha_3^2 - \alpha_1^2) + f(\alpha_3)(\alpha_1^2 - \alpha_2^2)}{2[f(\alpha_1)(\alpha_2 - \alpha_3) + f(\alpha_2)(\alpha_3 - \alpha_1) + f(\alpha_3)(\alpha_1 - \alpha_2)]}\tag{3.20}$$

Each sequence of obtaining the gradient and moving along the negative gradient direction until a minimum is found (i.e. executing a line search) is called an *iteration*. The algorithm consists of executing iterations until the norm of the gradient drops below a specified tolerance, indicating the necessary conditions have been met.

As shown in Fig. 3.7, at α^* , $\frac{df}{d\alpha} = 0$. The process of determining α^* will be referred to as *taking a minimizing step*, or, *executing an exact line search*.

4.3. Pros and Cons of Steepest Descent

Steepest descent has several advantages. It usually makes good progress when far from the optimum (in the above example the objective decreased from 19 to 3 in the first iteration), and it is very simple to implement. It always goes downhill. It is also guaranteed to converge to a local optimum if enough steps are taken.

However, if the function to be minimized is *eccentric*, convergence of steepest descent can be very slow, as indicated by the following theorem from Luenberger.¹

THEOREM. Convergence of Steepest Descent. For a quadratic function, if we take enough steps, the method of steepest descent converges to the unique minimum point \mathbf{x}^* of f . If we define the error in the objective function at the current value of \mathbf{x} as,

$$E(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{x}^*)^T \mathbf{H}(\mathbf{x} - \mathbf{x}^*) \quad (3.21)$$

there holds at every step k ,

$$E(\mathbf{x}^{k+1}) \leq \left(\frac{A-a}{A+a} \right)^2 E(\mathbf{x}^k) \quad (3.22)$$

where

A = Largest eigenvalue of \mathbf{H}

a = Smallest eigenvalue of \mathbf{H}

Thus if $A=50$ and $a=1$, we have that the error at the $k+1$ step is only guaranteed to be less than the error at the k step by,

$$E^{k+1} \leq \left(\frac{49}{51} \right)^2 E^k$$

and thus the error may be reduced very slowly.

“Roughly speaking, the above theorem says that the convergence rate of steepest descent is slowed as the contours of f become more eccentric. If $a = A$, corresponding to circular contours, convergence occurs in a single step. Note, however, that even if $n-1$ of the n eigenvalues are equal and the remaining one is a great distance from these, convergence will be slow, and hence a single abnormal eigenvalue can destroy the effectiveness of steepest descent.”

The above theorem is based on a quadratic function. If we have a quadratic, and we do rotation and translation of the axes, we can eliminate all of the linear and cross product terms. We then have only the pure second order terms left. The eigenvalues of the resulting Hessian are equal to twice the coefficients of the pure second order terms. Thus the function,

$$f = x_1^2 + x_2^2$$

would have equal eigenvalues of (2, 2) and would represent the circular contours as mentioned above, shown in Fig. 3.8. Steepest descent would converge in one step. Conversely the function,

¹ Luenberger and Ye, *Linear and Nonlinear Programming, Third Edition*, 2008

$$f = 50x_1^2 + x_2^2$$

has eigenvalues of (100, 2). The contours would be highly eccentric and convergence of steepest descent would be very slow. A contour plot of this function is given in Fig 3.9,

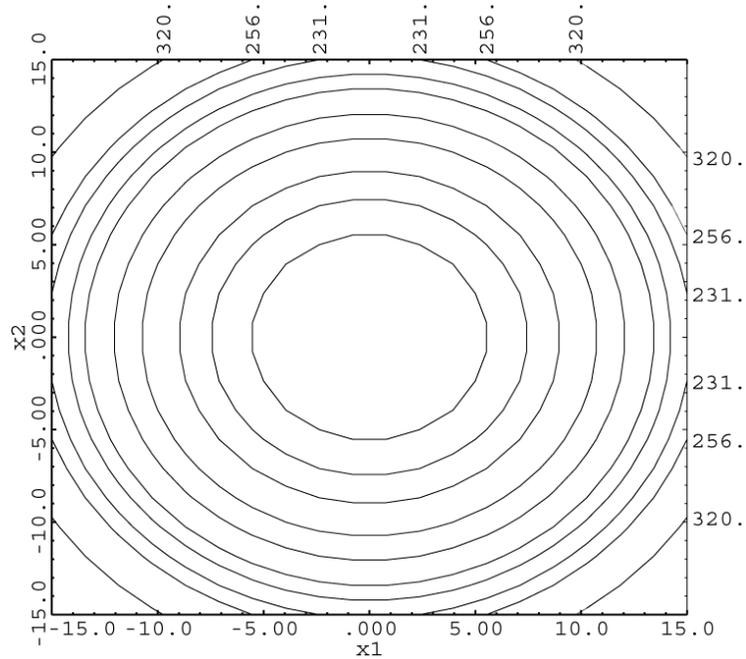


Fig. 3.8. Contours of the function, $f = x_1^2 + x_2^2$.

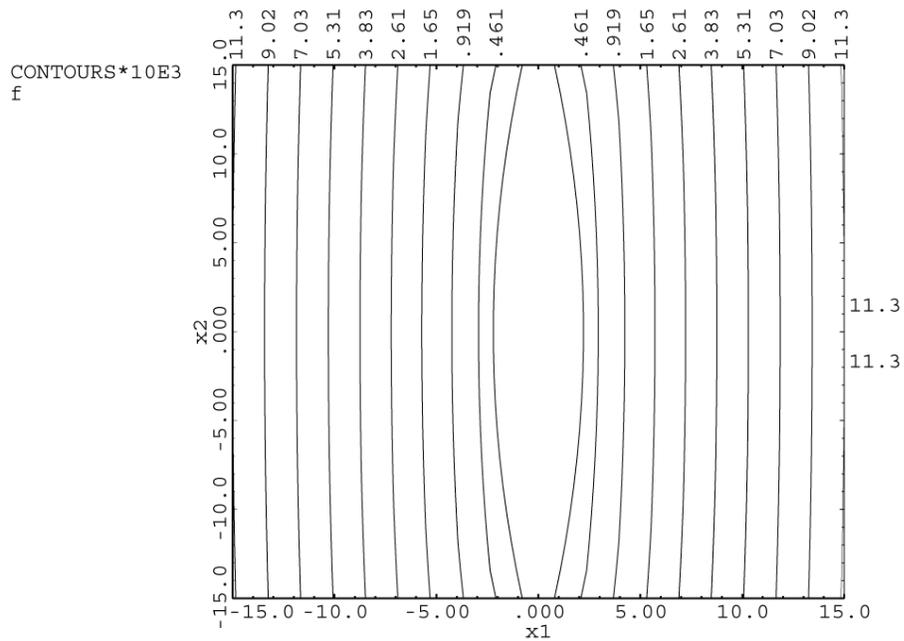


Fig. 3.9. Contours of the function, $f = 50x_1^2 + x_2^2$. Notice how the contours have been “stretched” out.

5. The Directional Derivative

It is sometimes useful to calculate $\frac{df}{d\alpha}$ along some search direction \mathbf{s} . From the chain rule for differentiation,

$$\frac{df}{d\alpha} = \sum \left(\frac{\partial f}{\partial x_i} \right) \left(\frac{dx_i}{d\alpha} \right)$$

Noting that $\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha \mathbf{s}$, or, for a single element of vector \mathbf{x} , $x_i^{k+1} = x_i^k + \alpha s_i^k$, we have

$$\frac{dx_i}{d\alpha} = s_i, \text{ so}$$

$$\frac{df}{d\alpha} = \sum \left(\frac{\partial f}{\partial x_i} \right) \left(\frac{dx_i}{d\alpha} \right) = \sum \left(\frac{\partial f}{\partial x_i} \right) s_i = \nabla f^T \mathbf{s} \quad (3.23)$$

As an example, we will find the directional derivative, $\frac{df}{d\alpha}$, for the problem given in Section

4.2 above, at $\alpha=0$. From (3.23): $\frac{df}{d\alpha} = \nabla f^T \mathbf{s} = [-8 \quad 14] \begin{bmatrix} 0.5 \\ -0.86 \end{bmatrix} = -16.04$

This gives us the change in the function for a small step in the search direction, i.e.,

$$\Delta f \approx \frac{df}{d\alpha} \Delta \alpha \quad (3.24)$$

If $\Delta \alpha = 0.01$, the predicted change is 0.1604. The actual change in the function is 0.1599.

Equation (3.23) is the same equation for checking if a direction goes downhill, given in Section 1.4. Before we just looked at the sign; if negative we knew we were going downhill. Now we see that the value has meaning as well: it represents the expected change in the

function for a small step. If, for example, the value of $\left. \frac{df}{d\alpha} \right|_{\alpha=0}$ is less than some epsilon, we

could terminate the line search, because the predicted change in the objective function is below a minimum threshold.

Another important value of $\frac{df}{d\alpha}$ occurs at α^* . If we locate the minimum exactly, then

$$\left. \frac{df}{d\alpha} \right|_{\alpha=\alpha^*} = (\nabla f^{k+1})^T \mathbf{s}^k = 0 \quad (3.25)$$

As we have seen in examples, when we take a minimizing step we stop where the search direction is tangent to the contours of the function. Thus the gradient at this new point is orthogonal to the previous search direction.

6. Newton's Method

6.1. Derivation

Another classical method we will briefly study is called Newton's method. It simply makes a quadratic approximation to a function at the current point and solves for where the necessary conditions (to the approximation) are satisfied. Starting with a Taylor series:

$$f^{k+1} = f^k + (\nabla f^k)^T \Delta \mathbf{x}^k + \frac{1}{2} (\Delta \mathbf{x}^k)^T \mathbf{H}^k \Delta \mathbf{x}^k \quad (3.26)$$

Since the gradient and Hessian are evaluated at k , they are just a vector and matrix of constants. Taking the gradient (Section 9.1),

$$\nabla f^{k+1} = \nabla f^k + \mathbf{H}^k \Delta \mathbf{x}^k$$

and setting $\nabla f^{k+1} = 0$, we have,

$$\mathbf{H}^k \Delta \mathbf{x}^k = -\nabla f^k$$

Solving for $\Delta \mathbf{x}$:

$$\Delta \mathbf{x}^k = -(\mathbf{H}^k)^{-1} \nabla f^k \quad (3.27)$$

Note that we have solved for a vector, i.e. $\Delta \mathbf{x}$, which has both a step length and direction.

6.2. Example: Newton's Method

We wish to optimize the function, $f(\mathbf{x}) = x_1^2 - 2x_1x_2 + 4x_2^2$ from the point $\mathbf{x}^0 = \begin{bmatrix} -3 \\ 1 \end{bmatrix}$.

At this point $\nabla f^0 = \begin{bmatrix} -8 \\ 14 \end{bmatrix}$ and the Hessian is, $\mathbf{H} = \begin{bmatrix} 2 & -2 \\ -2 & 8 \end{bmatrix}$. The Hessian inverse is given

by: $\begin{bmatrix} 0.6667 & 0.16667 \\ 0.16667 & 0.16667 \end{bmatrix}$. Thus $\Delta \mathbf{x} = -\begin{bmatrix} 0.6667 & 0.16667 \\ 0.16667 & 0.16667 \end{bmatrix} \begin{bmatrix} -8 \\ 14 \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$

So, $\mathbf{x}^1 = \mathbf{x}^0 + \Delta \mathbf{x} = \begin{bmatrix} -3 \\ 1 \end{bmatrix} + \begin{bmatrix} 3 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

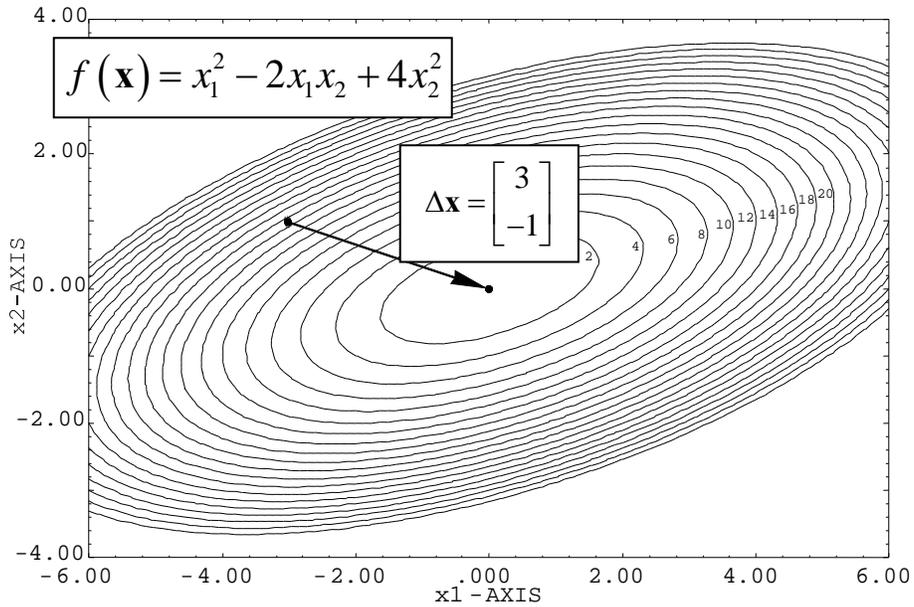


Fig. 3.10. The operation of Newton’s method.

6.3. Pros and Cons of Newton's Method

We can see in the above example Newton’s method solved the problem in one step. This is true in general: Newton’s method will drive to the stationary point of a quadratic in one step. On a non-quadratic, if we are near an optimum, Newton’s method will drive very close to the optimum in one step.

However we should note some drawbacks. First, it requires second derivatives. Normally we compute derivatives numerically, and computing second derivatives is computationally expensive, on the order of n^2 function evaluations, where n is the number of design variables.

The derivation of Newton’s method solved for where the gradient is equal to zero. The gradient is equal to zero at a min, a max or a saddle, and nothing in the method differentiates between these. Thus Newton’s method can *diverge*, or fail to go downhill (indeed, not only not go downhill, but go to a maximum!). This is obviously a serious drawback.

7. Quasi-Newton Methods

7.1. Introduction

Let’s summarize the pros and cons of Newton's method and Steepest Descent:

	Pros	Cons
Steepest Descent	Always goes downhill Always converges Simple to implement	Slow on eccentric functions
Newton’s Method	Solves quadratic in one step. Very fast when close to optimum on non quadratic.	Requires second derivatives, Can diverge

We want to develop a method that starts out like steepest descent and gradually becomes Newton's method, doesn't need second derivatives, doesn't have trouble with eccentric functions and doesn't diverge! Fortunately such methods exist. They combine the good aspects of steepest descent and Newton's method without the drawbacks. These methods are called *quasi-Newton* methods or sometimes *variable metric* methods.

In general we will define our search direction by the expression

$$\mathbf{s} = -\mathbf{N}\nabla f(\mathbf{x}) \quad (3.28)$$

where \mathbf{N} will be called the “direction matrix.”

If $\mathbf{N} = \mathbf{I}$, then $\mathbf{s} = -\nabla f(\mathbf{x}) \rightarrow$ Steepest Descent

If $\mathbf{N} = \mathbf{H}^{-1}$, then $\mathbf{s} = -\mathbf{H}^{-1}\nabla f(\mathbf{x}) \rightarrow$ Newton's Method

If \mathbf{N} is always positive definite, then \mathbf{s} always points downhill. To show this, our criterion for moving downhill is:

$$\mathbf{s}^T \nabla f < 0$$

Or,

$$\nabla f^T \mathbf{s} < 0 \quad (3.29)$$

Substituting (3.28) into (3.29):

$$-(\nabla f^T \mathbf{N} \nabla f) < 0 \quad (3.30)$$

Since \mathbf{N} is positive definite, we know that any vector which pre-multiplies \mathbf{N} and post-multiplies \mathbf{N} will result in a positive scalar. Thus the quantity within the parentheses is always positive; with the negative sign it becomes always negative, and therefore always goes downhill.

7.2. A Rank One Hessian Inverse Update

7.2.1. Development

In this section we will develop one of the simplest updates, called a “rank one” update because the correction to the direction matrix, \mathbf{N} , is a rank one matrix (i.e., it only has one independent row or column). We first start with some preliminaries.

Starting with a Taylor series:

$$f^{k+1} = f^k + (\nabla f^k)^T \Delta \mathbf{x}^k + \frac{1}{2} (\Delta \mathbf{x}^k)^T \mathbf{H} \Delta \mathbf{x}^k \quad (3.31)$$

where $\Delta \mathbf{x}^k = \mathbf{x}^{k+1} - \mathbf{x}^k$

the gradient is given by,

$$\nabla f^{k+1} = \nabla f^k + \mathbf{H} \Delta \mathbf{x}^k \quad (3.32)$$

and defining:

$$\boldsymbol{\gamma}^k = \nabla f^{k+1} - \nabla f^k \quad (3.33)$$

we have,

$$\boldsymbol{\gamma}^k = \mathbf{H} \Delta \mathbf{x}^k \quad \text{or} \quad \mathbf{H}^{-1} \boldsymbol{\gamma}^k = \Delta \mathbf{x}^k \quad (3.34)$$

Equation (3.34) is very important: it shows that for a quadratic function, the inverse of the Hessian matrix (\mathbf{H}^{-1}) maps differences in the gradients to differences in \mathbf{x} . The relationship expressed by (3.34) is called the *Newton condition*.

We will make the direction matrix satisfy this relationship. However, since we can only calculate $\boldsymbol{\gamma}^k$ and $\Delta \mathbf{x}^k$ after the line search, we will make

$$\mathbf{N}^{k+1} \boldsymbol{\gamma}^k = \Delta \mathbf{x}^k \quad (3.35)$$

This expression is sometimes called the *quasi-Newton condition*. It is “quasi” in that it involves $k+1$ for \mathbf{N} instead of k . Equation (3.35) involves more unknowns (the elements of \mathbf{N}^{k+1}) than equations, so how do we solve for \mathbf{N}^{k+1} ?

One of the simplest possibilities is:

$$\mathbf{N}^{k+1} = \mathbf{N}^k + a \mathbf{u} \mathbf{u}^T \quad (3.36)$$

Where we will “update” the direction matrix with a correction which is of the form $a \mathbf{u} \mathbf{u}^T$, which is a rank one symmetric matrix.

If we substitute (3.36) into (3.35), we have,

$$\mathbf{N}^k \boldsymbol{\gamma}^k + a \mathbf{u} \mathbf{u}^T \boldsymbol{\gamma}^k = \Delta \mathbf{x}^k \quad (3.37)$$

or

$$a \underbrace{\mathbf{u}^T \boldsymbol{\gamma}^k}_{\text{scalar}} = (\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k) \quad (3.38)$$

Noting that $\mathbf{u}^T \boldsymbol{\gamma}^k$ is a scalar, then \mathbf{u} must be proportional to $(\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k)$. Since any change in length can be absorbed by a , we will set

$$\mathbf{u} = (\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k) \quad (3.39)$$

Substituting (3.39) into (3.38):

$$a(\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k) \underbrace{(\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k)^T \boldsymbol{\gamma}^k}_{\text{scalar}} = (\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k) \quad (3.40)$$

For this to be true,

$$a(\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k)^T \boldsymbol{\gamma}^k = 1$$

so

$$a = \frac{1}{(\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k)^T \boldsymbol{\gamma}^k} \quad (3.41)$$

Substituting (3.41) and (3.39) into (3.36) gives the expression we need:

$$\mathbf{N}^{k+1} = \mathbf{N}^k + \frac{(\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k)(\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k)^T}{(\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k)^T \boldsymbol{\gamma}^k} \quad (3.42)$$

Equation (3.42) allows us to get a new direction matrix in terms of the previous matrix and the difference in \mathbf{x} and the gradient. We then use this to get a new search direction according to (3.28).

7.2.2. Example: Rank One Hessian Inverse Update

We wish to minimize the function $f(\mathbf{x}) = x_1^2 - 2x_1x_2 + 4x_2^2$

$$\text{starting from } \mathbf{x}^0 = \begin{bmatrix} -3 \\ 1 \end{bmatrix} \quad \nabla f^0 = \begin{bmatrix} -8 \\ 14 \end{bmatrix}$$

We let $\mathbf{N}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ so the search direction is

$$\mathbf{s}^0 = -\mathbf{N} \nabla f^0 = -\nabla f^0$$

We normalize the search direction to be: $\mathbf{s}^0 = \begin{bmatrix} 0.496 \\ -0.868 \end{bmatrix}$

We execute a line search in this direction (using, for example, a quadratic fit) and stop at

$$\mathbf{x}^1 = \begin{bmatrix} -2.030 \\ -0.698 \end{bmatrix} \quad \nabla f^1 = \begin{bmatrix} -2.664 \\ -1.522 \end{bmatrix}$$

$$\text{Then } \Delta \mathbf{x}^0 = \mathbf{x}^1 - \mathbf{x}^0 = \begin{bmatrix} -2.030 \\ -0.698 \end{bmatrix} - \begin{bmatrix} -3.000 \\ 1.000 \end{bmatrix} = \begin{bmatrix} 0.970 \\ -1.698 \end{bmatrix}$$

$$\boldsymbol{\gamma}^0 = \nabla f^1 - \nabla f^0 = \begin{bmatrix} -2.664 \\ -1.522 \end{bmatrix} - \begin{bmatrix} -8.000 \\ 14.000 \end{bmatrix} = \begin{bmatrix} 5.336 \\ -15.522 \end{bmatrix}$$

and

$$\begin{aligned} \Delta \mathbf{x}^0 - \mathbf{N}^0 \boldsymbol{\gamma}^0 &= \begin{bmatrix} 0.970 \\ -1.698 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 5.336 \\ -15.522 \end{bmatrix} = \begin{bmatrix} -4.366 \\ 13.824 \end{bmatrix} \\ a\mathbf{u}\mathbf{u}^T &= \frac{(\Delta \mathbf{x}^0 - \mathbf{N}^0 \boldsymbol{\gamma}^0)(\Delta \mathbf{x}^0 - \mathbf{N}^0 \boldsymbol{\gamma}^0)^T}{(\Delta \mathbf{x}^0 - \mathbf{N}^0 \boldsymbol{\gamma}^0)^T \boldsymbol{\gamma}^0} = \frac{\begin{bmatrix} -4.366 \\ 13.824 \end{bmatrix} \begin{bmatrix} -4.366 & 13.824 \end{bmatrix}}{\begin{bmatrix} -4.366 & 13.824 \end{bmatrix} \begin{bmatrix} 5.336 \\ -15.522 \end{bmatrix}} \\ &= \frac{\begin{bmatrix} 19.062 & -60.364 \\ -60.364 & 191.158 \end{bmatrix}}{-237.932} \\ &= \begin{bmatrix} -0.080 & 0.254 \\ 0.254 & -0.803 \end{bmatrix} \end{aligned}$$

$$\mathbf{N}^1 = \mathbf{N}^0 + a\mathbf{u}\mathbf{u}^T$$

$$\mathbf{N}^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} -0.080 & 0.254 \\ 0.254 & -0.803 \end{bmatrix}$$

$$\mathbf{N}^1 = \begin{bmatrix} 0.920 & 0.254 \\ 0.254 & 0.197 \end{bmatrix}$$

New search direction:

$$\begin{aligned} \mathbf{s}^1 &= - \begin{bmatrix} 0.920 & 0.254 \\ 0.254 & 0.197 \end{bmatrix} \begin{bmatrix} -2.664 \\ -1.522 \end{bmatrix} \\ &= \begin{bmatrix} 2.837 \\ 0.975 \end{bmatrix} \end{aligned}$$

When we step in this direction, using again a line search, we arrive at the optimum

$$\mathbf{x}^2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \nabla f(\mathbf{x}^2) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

At this point we are done. However, if we update the direction matrix one more time, we find it has become the inverse Hessian.

$$\Delta \mathbf{x}^1 = \mathbf{x}^2 - \mathbf{x}^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} -2.030 \\ -0.698 \end{bmatrix} = \begin{bmatrix} 2.030 \\ 0.698 \end{bmatrix}$$

$$\boldsymbol{\gamma}^1 = \nabla f^2 - \nabla f^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} -2.664 \\ -1.524 \end{bmatrix} = \begin{bmatrix} 2.664 \\ 1.524 \end{bmatrix}$$

$$(\Delta \mathbf{x}^1 - \mathbf{N}^1 \boldsymbol{\gamma}^1) = \begin{bmatrix} 2.030 \\ 0.698 \end{bmatrix} - \begin{bmatrix} 0.920 & 0.254 \\ 0.254 & 0.197 \end{bmatrix} \begin{bmatrix} 2.664 \\ 1.524 \end{bmatrix}$$

$$= \begin{bmatrix} 2.030 \\ 0.698 \end{bmatrix} - \begin{bmatrix} 2.838 \\ 0.977 \end{bmatrix} = \begin{bmatrix} -0.808 \\ -0.279 \end{bmatrix}$$

$$a\mathbf{u}\mathbf{u}^T = \frac{(\Delta \mathbf{x}^1 - \mathbf{N}^1 \boldsymbol{\gamma}^1)(\Delta \mathbf{x}^1 - \mathbf{N}^1 \boldsymbol{\gamma}^1)^T}{(\Delta \mathbf{x}^1 - \mathbf{N}^1 \boldsymbol{\gamma}^1)^T \boldsymbol{\gamma}^1} = \frac{\begin{bmatrix} -0.808 \\ -0.279 \end{bmatrix} \begin{bmatrix} -0.808 & -0.279 \end{bmatrix}}{\begin{bmatrix} -0.808 & -0.279 \end{bmatrix} \begin{bmatrix} 2.664 \\ 1.524 \end{bmatrix}} = \begin{bmatrix} -0.253 & -0.088 \\ -0.088 & -0.030 \end{bmatrix}$$

$$\mathbf{N}^2 = \mathbf{N}^1 + a\mathbf{u}\mathbf{u}^T = \begin{bmatrix} 0.920 & 0.254 \\ 0.254 & 0.197 \end{bmatrix} + \begin{bmatrix} -0.253 & -0.088 \\ -0.088 & -0.030 \end{bmatrix} = \begin{bmatrix} 0.667 & 0.166 \\ 0.166 & 0.167 \end{bmatrix} = \mathbf{H}^{(-1)}$$

7.2.3. The Hereditary Property

The hereditary property is an important property of all update methods. The hereditary property states that not only will \mathbf{N}^{k+1} satisfy (3.35), but

$$\begin{aligned} \mathbf{N}^{k+1} \boldsymbol{\gamma}^k &= \Delta \mathbf{x}^k \\ \mathbf{N}^{k+1} \boldsymbol{\gamma}^{k-1} &= \Delta \mathbf{x}^{k-1} \\ \mathbf{N}^{k+1} \boldsymbol{\gamma}^{k-2} &= \Delta \mathbf{x}^{k-2} \\ \mathbf{N}^{k+1} \boldsymbol{\gamma}^{k-n+1} &= \Delta \mathbf{x}^{k-n+1} \end{aligned} \tag{3.43}$$

where n is the number of variables. That is, (3.35) is not only satisfied for the current step, but for the *last $n-1$ steps*. Why is this significant? Let's write this relationship of (3.43) as follows:

$$\mathbf{N}^{k+1} \begin{bmatrix} \boldsymbol{\gamma}^k, \boldsymbol{\gamma}^{k-1}, \boldsymbol{\gamma}^{k-2}, \dots, \boldsymbol{\gamma}^{k-n+1} \end{bmatrix} = \begin{bmatrix} \Delta \mathbf{x}^k, \Delta \mathbf{x}^{k-1}, \Delta \mathbf{x}^{k-2}, \dots, \Delta \mathbf{x}^{k-n+1} \end{bmatrix}$$

Let the matrix defined by the columns of $\boldsymbol{\gamma}$ be denoted by \mathbf{G} , and the matrix defined by columns of $\Delta \mathbf{x}$ be denoted by $\Delta \mathbf{X}$. Then,

$$\mathbf{N}^{k+1}\mathbf{G} = \Delta\mathbf{X}$$

If $\boldsymbol{\gamma}^k \dots \boldsymbol{\gamma}^{k-n+1}$ are independent, and if we have n vectors, i.e. \mathbf{G} is a square matrix, then the inverse for \mathbf{G} exists and is unique and

$$\mathbf{N}^{k+1} = \Delta\mathbf{X}\mathbf{G}^{-1} \quad (3.44)$$

is uniquely defined.

Since the Hessian inverse satisfies (3.44) for a quadratic function, then we have the important result that, *after n updates the direction matrix becomes the Hessian inverse for a quadratic function*. This implies the quasi-Newton method will solve a quadratic in no more than $n+1$ steps. The proof that our rank one update has the hereditary property is given in the next section.

7.2.4. Proof of the Hereditary Property for the Rank One Update

THEOREM. Let \mathbf{H} be a constant symmetric matrix and suppose that $\Delta\mathbf{x}^0, \Delta\mathbf{x}^1, \dots, \Delta\mathbf{x}^k$ and $\boldsymbol{\gamma}^0, \boldsymbol{\gamma}^1, \dots, \boldsymbol{\gamma}^k$ are given vectors, where $\boldsymbol{\gamma}^i = \mathbf{H}\Delta\mathbf{x}^i$, $i = 0, 1, 2, \dots, k$, where $k < n$. Starting with any initial symmetric matrix \mathbf{N}^0 , let

$$\mathbf{N}^{k+1} = \mathbf{N}^k + \frac{(\Delta\mathbf{x}^k - \mathbf{N}^k\boldsymbol{\gamma}^k)(\Delta\mathbf{x}^k - \mathbf{N}^k\boldsymbol{\gamma}^k)^T}{(\Delta\mathbf{x}^k - \mathbf{N}^k\boldsymbol{\gamma}^k)^T \boldsymbol{\gamma}^k} \quad (3.45)$$

then

$$\mathbf{N}^{k+1}\boldsymbol{\gamma}^i = \Delta\mathbf{x}^i \quad \text{for } i \leq k \quad (3.46)$$

PROOF. The proof is by induction. We will show that if (3.46) holds for previous direction matrix, it holds for the current direction matrix. We know that at the current point, k , the following is true,

$$\mathbf{N}^{k+1}\boldsymbol{\gamma}^k = \Delta\mathbf{x}^k \quad (3.47)$$

because we enforced this condition when we developed the update. Now, *suppose* it is true that,

$$\mathbf{N}^k\boldsymbol{\gamma}^i = \Delta\mathbf{x}^i \quad \text{for } i \leq k-1 \quad (3.48)$$

i.e. that the hereditary property holds for the *previous* direction matrix. We can post multiply (3.45) by $\boldsymbol{\gamma}^i$, giving,

$$\mathbf{N}^{k+1}\boldsymbol{\gamma}^i = \mathbf{N}^k\boldsymbol{\gamma}^i + \frac{(\Delta\mathbf{x}^k - \mathbf{N}^k\boldsymbol{\gamma}^k)(\Delta\mathbf{x}^k - \mathbf{N}^k\boldsymbol{\gamma}^k)^T \boldsymbol{\gamma}^i}{(\Delta\mathbf{x}^k - \mathbf{N}^k\boldsymbol{\gamma}^k)^T \boldsymbol{\gamma}^k} \quad (3.49)$$

To simplify things, let $\mathbf{y}^k = \frac{(\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k)}{(\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k)^\top \boldsymbol{\gamma}^k}$ so that we can write (3.49) as,

$$\mathbf{N}^{k+1} \boldsymbol{\gamma}^i = \mathbf{N}^k \boldsymbol{\gamma}^i + \mathbf{y}^k (\Delta \mathbf{x}^k - \mathbf{N}^k \boldsymbol{\gamma}^k)^\top \boldsymbol{\gamma}^i \quad (3.50)$$

We can distribute the transpose on the last term, and distribute the post multiplication $\boldsymbol{\gamma}^i$ to give (Note: Recall that when you take the transpose inside a product, the order of the product is reversed; also because \mathbf{N} is symmetric, $\mathbf{N}^T = \mathbf{N}$ thus: $(\mathbf{N}^k \boldsymbol{\gamma}^k)^\top \boldsymbol{\gamma}^i = (\boldsymbol{\gamma}^k)^\top \mathbf{N}^k \boldsymbol{\gamma}^i$),

$$\mathbf{N}^{k+1} \boldsymbol{\gamma}^i = \mathbf{N}^k \boldsymbol{\gamma}^i + \mathbf{y}^k \left[(\Delta \mathbf{x}^k)^\top \boldsymbol{\gamma}^i - (\boldsymbol{\gamma}^k)^\top \mathbf{N}^k \boldsymbol{\gamma}^i \right] \quad (3.51)$$

Since we have assumed (3.48) is true, we can replace $\mathbf{N}^k \boldsymbol{\gamma}^i$ with $\Delta \mathbf{x}^i$:

$$\mathbf{N}^{k+1} \boldsymbol{\gamma}^i = \Delta \mathbf{x}^i + \mathbf{y}^k \left[(\Delta \mathbf{x}^k)^\top \boldsymbol{\gamma}^i - (\boldsymbol{\gamma}^k)^\top \Delta \mathbf{x}^i \right] \quad (3.52)$$

Now we examine the term in brackets. We note that,

$$(\boldsymbol{\gamma}^k)^\top \Delta \mathbf{x}^i = (\mathbf{H} \Delta \mathbf{x}^k)^\top \Delta \mathbf{x}^i = (\Delta \mathbf{x}^k)^\top \mathbf{H} \Delta \mathbf{x}^i = (\Delta \mathbf{x}^k)^\top \boldsymbol{\gamma}^i \quad (3.53)$$

So the term in brackets in (3.52) vanishes, giving,

$$\mathbf{N}^{k+1} \boldsymbol{\gamma}^i = \Delta \mathbf{x}^i \quad \text{for } i \leq k \quad (3.54)$$

Thus, if the hereditary property holds for the previous direction matrix, it holds for the current direction matrix. When $k = 0$, condition (3.47) is all that is needed to have the hereditary property for the first update, \mathbf{N}^1 . The second update, \mathbf{N}^2 , will then have the hereditary property since \mathbf{N}^1 does, and so on.

7.3. Conjugacy

7.3.1. Definition

Quasi-Newton methods are also methods of *conjugate directions*. A set of search directions, $\mathbf{s}^0, \mathbf{s}^1, \dots, \mathbf{s}^k$ are said to be *conjugate* with respect to a square, symmetric matrix, \mathbf{H} , if,

$$(\mathbf{s}^k)^\top \mathbf{H} \mathbf{s}^i = 0 \quad \text{for all } i \neq k \quad (3.55)$$

A set of conjugate directions possesses an important property: If minimizing line searches are used along each conjugate direction, a method of conjugate directions is guaranteed to minimize a quadratic function of n variables in at most n steps. Himmelblau indicates the excellent convergence properties of quasi-Newton methods on general functions may be due more to their conjugate direction properties than to their ability to approximate the Hessian inverse.² Because of the importance of conjugate directions, we will prove two results here.

PROPOSITION. If \mathbf{H} is positive definite and the set of non-zero vectors $\mathbf{s}^0, \mathbf{s}^1, \dots, \mathbf{s}^{n-1}$ are conjugate to \mathbf{H} , then these vectors are linearly independent.

PROOF. Suppose we have constants, $\alpha^i, i = 0, 2, \dots, n-1$ such that

$$\alpha^0 \mathbf{s}^0 + \alpha^1 \mathbf{s}^1 + \dots + \alpha^k \mathbf{s}^k + \dots + \alpha^{n-1} \mathbf{s}^{n-1} = \mathbf{0} \quad (3.56)$$

Now we multiply each term by $(\mathbf{s}^k)^T \mathbf{H}$:

$$\alpha^0 \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^0}_{=0} + \alpha^1 \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^1}_{=0} + \dots + \alpha^k \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^k}_{\text{positive}} + \dots + \alpha^{n-1} \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^{n-1}}_{=0} = \mathbf{0} \quad (3.57)$$

From conjugacy, all of the terms except $\alpha^k (\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^k$ are zero. Since \mathbf{H} is positive definite, then the only way for this remaining term to be zero is for α^k to be zero. In this way we can show that for (3.57) to be satisfied all the α coefficients must be zero. This is the definition of linear independence.

7.3.2. Conjugate Direction Theorem

We will now show that a method of conjugate directions will solve a quadratic function in n steps, if minimizing steps are taken.

THEOREM. Let $\mathbf{s}^0, \mathbf{s}^1, \dots, \mathbf{s}^{n-1}$ be a set of non-zero \mathbf{H} conjugate vectors, with \mathbf{H} a positive definite matrix. For the function,

$$f^{k+1} = f^k + (\nabla f^k)^T (\mathbf{x}^{k+1} - \mathbf{x}^k) + \frac{1}{2} (\mathbf{x}^{k+1} - \mathbf{x}^k)^T \mathbf{H} (\mathbf{x}^{k+1} - \mathbf{x}^k) \quad (3.58)$$

the sequence,

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha^k \mathbf{s}^k \quad (3.59)$$

with,

$$\alpha^k = - \frac{(\nabla f^k)^T \mathbf{s}^k}{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^k} \quad (3.60)$$

$$\nabla f^{k+1} = \nabla f^k + \mathbf{H} (\mathbf{x}^{k+1} - \mathbf{x}^k)$$

² Himmelblau, *Applied Nonlinear Programming*, p. 112.

converges to the unique solution, $\mathbf{H}(\mathbf{x}^* - \mathbf{x}^k) = -\nabla f^k$, after n steps, that is $\mathbf{x}^n = \mathbf{x}^*$.

PROOF. Based on (3.59) above we note that,

$$\mathbf{x}^1 = \mathbf{x}^0 + \alpha^0 \mathbf{s}^0$$

Likewise for \mathbf{x}^2 :

$$\mathbf{x}^2 = \mathbf{x}^1 + \alpha^1 \mathbf{s}^1 = \mathbf{x}^0 + \alpha^0 \mathbf{s}^0 + \alpha^1 \mathbf{s}^1$$

Or, in general

$$(\mathbf{x}^k - \mathbf{x}^0) = \alpha^0 \mathbf{s}^0 + \alpha^1 \mathbf{s}^1 + \dots + \alpha^{k-1} \mathbf{s}^{k-1} \quad (3.61)$$

After n steps, we can write the optimum (assuming the directions are independent, which we just showed) as,

$$(\mathbf{x}^* - \mathbf{x}^0) = \alpha^0 \mathbf{s}^0 + \alpha^1 \mathbf{s}^1 + \dots + \alpha^k \mathbf{s}^k + \dots + \alpha^{n-1} \mathbf{s}^{n-1} \quad (3.62)$$

Multiplying both sides of (3.62) by $(\mathbf{s}^k)^T \mathbf{H}$, we have,

$$(\mathbf{s}^k)^T \mathbf{H}(\mathbf{x}^* - \mathbf{x}^0) = \alpha^0 \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^0}_{=0} + \alpha^1 \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^1}_{=0} + \dots + \alpha^k \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^k}_{\text{positive}} + \dots + \alpha^{n-1} \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^{n-1}}_{=0}$$

Solving for α^k :

$$\alpha^k = \frac{(\mathbf{s}^k)^T \mathbf{H}(\mathbf{x}^* - \mathbf{x}^0)}{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^k} \quad (3.63)$$

Unfortunately (3.63) is in terms of \mathbf{x}^* , which we presumably don't know. However, if we multiply (3.61) by $(\mathbf{s}^k)^T \mathbf{H}$, we have,

$$(\mathbf{s}^k)^T \mathbf{H}(\mathbf{x}^k - \mathbf{x}^0) = \alpha^0 \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^0}_{=0} + \alpha^1 \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^1}_{=0} + \dots + \alpha^{k-1} \underbrace{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^{k-1}}_{=0} \quad (3.64)$$

which gives,

$$(\mathbf{s}^k)^T \mathbf{H}(\mathbf{x}^k - \mathbf{x}^0) = 0 \quad (3.65)$$

Substituting this result into (3.63), we have

$$\alpha^k = \frac{(\mathbf{s}^k)^T \mathbf{H}(\mathbf{x}^* - \mathbf{x}^k)}{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^k} \quad (3.66)$$

Noting that $\mathbf{H}(\mathbf{x}^* - \mathbf{x}^k) = -\nabla f^k$ is the solution to (3.58), we can solve for the α^k as,

$$\alpha^k = -\frac{(\nabla f^k)^T \mathbf{s}^k}{(\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^k}$$

which is identical with (3.60).

We notice that (3.60) is the same as the minimizing step we derived in Section 9.2. Thus the conjugate direction theorem relies on taking minimizing steps.

7.3.3. Examples

We stated earlier that quasi-Newton methods are also methods of conjugate directions. Thus for the example given in Section 7.3, we should have,

$$(\mathbf{s}^0)^T \mathbf{H} \mathbf{s}^1 = 0$$

Substituting the search directions and Hessian of that problem,

$$[0.496 \quad -0.868] \begin{bmatrix} 2. & -2. \\ -2. & 8. \end{bmatrix} \begin{bmatrix} 2.837 \\ 0.975 \end{bmatrix} = 0.0017$$

Within the round-off of the data, we see this is verified.

In the previous problem we only had two search directions. Let's look at a problem where we have three search directions so we have more conjugate relationships to examine. We will consider the problem, $\text{Min } f = 2x_1 + x_1^2 + 4x_2 + 4x_2^2 + 8x_3 + 2x_3^2$.

$$\text{Starting from } \mathbf{x}^0 = \begin{bmatrix} 2 \\ 3 \\ 3 \end{bmatrix} \quad \nabla f^0 = \begin{bmatrix} 6 \\ 28 \\ 20 \end{bmatrix} \quad \mathbf{s}^0 = \begin{bmatrix} -0.172 \\ -0.802 \\ -0.573 \end{bmatrix}$$

We execute a line search in the direction of steepest descent (normalized as \mathbf{s}^0 above), stop at α^* and determine the new point and gradient. We calculate the new search direction using our rank 1 update,

$$\mathbf{x}^1 = \begin{bmatrix} 1.079 \\ -1.300 \\ -0.072 \end{bmatrix} \quad \nabla f^1 = \begin{bmatrix} 4.157 \\ -6.401 \\ 7.714 \end{bmatrix} \quad \mathbf{s}^1 = \begin{bmatrix} -4.251 \\ 3.320 \\ -8.657 \end{bmatrix}$$

We go through this cycle again,

$$\mathbf{x}^2 = \begin{bmatrix} 0.019 \\ -0.473 \\ -2.229 \end{bmatrix} \quad \nabla f^2 = \begin{bmatrix} 2.038 \\ 0.218 \\ -0.917 \end{bmatrix} \quad \mathbf{s}^2 = \begin{bmatrix} -2.107 \\ -0.056 \\ 0.474 \end{bmatrix}$$

After stepping in the above direction we arrive at the optimum,

$$\mathbf{x}^3 = \begin{bmatrix} -1.000 \\ -0.500 \\ -2.000 \end{bmatrix} \quad \nabla f^3 = \begin{bmatrix} 0.000 \\ 0.000 \\ 0.000 \end{bmatrix}$$

Since we have used a method of conjugate directions, \mathbf{s}^2 should be conjugate to \mathbf{s}^1 and \mathbf{s}^0 . We will check this:

$$(\mathbf{s}^0)^T \mathbf{H} \mathbf{s}^2 = \begin{bmatrix} -0.172 & -0.802 & -0.573 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} -2.107 \\ -0.056 \\ 0.474 \end{bmatrix} = -0.0023$$

$$(\mathbf{s}^1)^T \mathbf{H} \mathbf{s}^2 = \begin{bmatrix} -4.251 & 3.320 & -8.657 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} -2.107 \\ -0.056 \\ 0.474 \end{bmatrix} = 0.0127$$

7.3.4. Some Insight into Conjugacy

As we did in section 4.3, we will define the “error” in the objective at the current value of \mathbf{x} as,

$$E(\mathbf{x}) = \frac{1}{2} (\mathbf{x} - \mathbf{x}^*)^T \mathbf{H} (\mathbf{x} - \mathbf{x}^*)$$

We can rewrite this expression as,

$$E(\mathbf{a}) = \frac{1}{2} (\mathbf{a} - \mathbf{a}^*)^T \mathbf{S}^T \mathbf{H} \mathbf{S} (\mathbf{a} - \mathbf{a}^*) \quad (3.67)$$

Where \mathbf{S} is a matrix with columns, $\mathbf{s}^0, \mathbf{s}^1, \dots, \mathbf{s}^{n-1}$. If the \mathbf{s} vectors are conjugate then (3.67) reduces to,

$$E(\boldsymbol{\alpha}) = \frac{1}{2} \sum_{i=0}^{n-1} (\alpha^i - \alpha^*)^2 d^i$$

where $d^i = (\mathbf{s}^i)^T \mathbf{H} \mathbf{s}^{(i)}$. $E(\boldsymbol{\alpha})$ can then be minimized by choosing $\alpha^i = \alpha^*$, i.e., by making exact line searches. Quoting Fletcher,³ “Thus conjugacy implies a diagonalizing transformation $\mathbf{S}^T \mathbf{H} \mathbf{S}$ of \mathbf{H} to a new coordinate system, $\boldsymbol{\alpha}$, in which the variables are decoupled. A conjugate direction method is then the alternating variables method applied in this new coordinate system.” The “alternating variables” method referred to is just a method where the optimum is found with respect to one variable, holding the rest constant, and then a second variable, etc. Usually such a scheme would not work well. Conjugate directions are such that the α^i ’s are decoupled so it does work here.

As we show in Section 9.3, another result of conjugacy is that at the $k+1$ step,

$$(\nabla f^{k+1})^T \mathbf{s}^i = 0 \quad \text{for all } i \leq k \quad (3.68)$$

Equation (3.68) indicates 1) that the current gradient is orthogonal to all the past search directions, and 2) at the current point we have zero slope with respect to all past search directions, i.e.,

$$\frac{\partial f}{\partial \alpha^i} = 0 \quad \text{for all } i \leq k$$

meaning we have minimized the function in the “subspace” of the previous directions. As an example, for the three variable function of Section 7.5, ∇f^2 should be orthogonal to \mathbf{s}^0 and \mathbf{s}^1 :

$$\begin{aligned} (\nabla f^2)^T \mathbf{s}^0 &= [2.038 \quad 0.218 \quad -0.917] \begin{bmatrix} -0.172 \\ -0.802 \\ -0.573 \end{bmatrix} = 0.0007 \\ (\nabla f^2)^T \mathbf{s}^1 &= [2.038 \quad 0.218 \quad -0.917] \begin{bmatrix} -4.251 \\ 3.320 \\ -8.657 \end{bmatrix} = -0.0013 \end{aligned}$$

7.4. Rank 2 Updates

7.4.1. The DFP Method

Although the rank one update does have the hereditary property (and is a method of conjugate directions), it does not guarantee that at each stage the direction matrix, \mathbf{N} , is positive definite. It is important that the update remain positive definite because this insures the search

³ R. Fletcher, *Practical Methods of Optimization, Second Edition*, 1987, pg. 26.

direction will always go downhill. It has been shown that (3.42) is the only rank one update which satisfies the quasi-Newton condition. For more flexibility, rank 2 updates have been proposed. These are of the form,

$$\mathbf{N}^{k+1} = \mathbf{N}^k + a\mathbf{u}\mathbf{u}^T + b\mathbf{v}\mathbf{v}^T \quad (3.69)$$

If we substitute this into the quasi-Newton condition,

$$\mathbf{N}^{k+1}\boldsymbol{\gamma}^k = \Delta\mathbf{x}^k \quad (3.70)$$

we have,

$$\mathbf{N}^k\boldsymbol{\gamma}^k + a\mathbf{u}\mathbf{u}^T\boldsymbol{\gamma}^k + b\mathbf{v}\mathbf{v}^T\boldsymbol{\gamma}^k = \Delta\mathbf{x}^k \quad (3.71)$$

There are a number of possible choices for \mathbf{u} and \mathbf{v} . One choice is to try,

$$\mathbf{u} = \Delta\mathbf{x}^k, \quad \mathbf{v} = \mathbf{N}^k\boldsymbol{\gamma}^k \quad (3.72)$$

Substituting (3.72) into (3.71),

$$\mathbf{N}^k\boldsymbol{\gamma}^k + a\Delta\mathbf{x}^k \underbrace{(\Delta\mathbf{x}^k)^T \boldsymbol{\gamma}^k}_{\text{scalar}} + b\mathbf{N}^k\boldsymbol{\gamma}^k \underbrace{(\mathbf{N}^k\boldsymbol{\gamma}^k)^T \boldsymbol{\gamma}^k}_{\text{scalar}} = \Delta\mathbf{x}^k \quad (3.73)$$

In (3.73) we note that the dot products result in scalars. If we choose a and b such that,

$$a(\Delta\mathbf{x}^k)^T \boldsymbol{\gamma}^k = 1 \text{ and } b(\mathbf{N}^k\boldsymbol{\gamma}^k)^T \boldsymbol{\gamma}^k = -1 \quad (3.74)$$

Equation (3.71) becomes,

$$\mathbf{N}^k\boldsymbol{\gamma}^k + \Delta\mathbf{x}^k - \mathbf{N}^k\boldsymbol{\gamma}^k = \Delta\mathbf{x}^k \quad (3.75)$$

and is satisfied.

Combining (3.74), (3.72) and (3.69), the update is,

$$\mathbf{N}^{k+1} = \mathbf{N}^k + \frac{\Delta\mathbf{x}^k (\Delta\mathbf{x}^k)^T}{(\Delta\mathbf{x}^k)^T \boldsymbol{\gamma}^k} - \frac{\mathbf{N}^k\boldsymbol{\gamma}^k (\mathbf{N}^k\boldsymbol{\gamma}^k)^T}{(\mathbf{N}^k\boldsymbol{\gamma}^k)^T \boldsymbol{\gamma}^k} \quad (3.76)$$

Or, with some rearranging, as it is more commonly given,

$$\mathbf{N}^{k+1} = \mathbf{N}^k + \frac{\Delta\mathbf{x}^k (\Delta\mathbf{x}^k)^T}{(\Delta\mathbf{x}^k)^T \boldsymbol{\gamma}^k} - \frac{\mathbf{N}^k\boldsymbol{\gamma}^k (\boldsymbol{\gamma}^k)^T \mathbf{N}^k}{(\boldsymbol{\gamma}^k)^T \mathbf{N}^k\boldsymbol{\gamma}^k} \quad (3.77)$$

Davidon⁴ was the first one to propose this update. Fletcher and Powell further developed his method;⁵ thus this method came to be known as the Davidon-Fletcher-Powell (DFP) update. This update has the following properties,

For quadratic functions:

1. it has the hereditary property; after n updates, $\mathbf{N}^n = \mathbf{H}^{-1}$.
2. it is a method of conjugate directions and therefore terminates after at most n steps.

For general functions (including quadratics):

3. the direction matrix \mathbf{N} remains positive definite if we do exact line searches. This guarantees the search direction points downhill at every step. This property is proved in the next section.

7.4.2. Proof the DFP Update Stays Positive Definite

THEOREM. If $(\Delta \mathbf{x}^k)^T \boldsymbol{\gamma} > 0$ for all steps of the algorithm, and if we start with any symmetric, positive definite matrix, \mathbf{N}^0 , then the DFP update preserves the positive definiteness of \mathbf{N}^k for all k .

PROOF. The proof is inductive. We will show that if \mathbf{N}^k is positive definite, \mathbf{N}^{k+1} is also. From the definition of positive definiteness,

$$\mathbf{z}^T \mathbf{N}^{k+1} \mathbf{z} > 0 \quad \text{for all } \mathbf{z} \neq 0$$

For simplicity we will drop the superscript k on the update terms. From (3.66),

$$\mathbf{z}^T \mathbf{N}^{k+1} \mathbf{z} = \underbrace{\mathbf{z}^T \mathbf{N}^k \mathbf{z}}_{\text{term 1}} + \underbrace{\mathbf{z}^T \begin{pmatrix} \Delta \mathbf{x} \Delta \mathbf{x}^T \\ \Delta \mathbf{x}^T \boldsymbol{\gamma} \end{pmatrix} \mathbf{z}}_{\text{term 2}} - \underbrace{\mathbf{z}^T \begin{pmatrix} \mathbf{N} \boldsymbol{\gamma} \boldsymbol{\gamma}^T \mathbf{N} \\ \boldsymbol{\gamma}^T \mathbf{N} \boldsymbol{\gamma} \end{pmatrix} \mathbf{z}}_{\text{term 3}} \quad (3.78)$$

We need to show that all the terms on the right hand side are positive. We will focus for a moment on the first and third terms on the right hand side. Noting that \mathbf{N} can be written as $\mathbf{N} = \mathbf{L}\mathbf{L}^T$ via Choleski decomposition, and if we substitute $\mathbf{a} = \mathbf{L}^T \mathbf{z}$, $\mathbf{a}^T = \mathbf{z}^T \mathbf{L}$, $\mathbf{b} = \mathbf{L}^T \boldsymbol{\gamma}$, $\mathbf{b}^T = \boldsymbol{\gamma}^T \mathbf{L}$ the first and third terms are,

$$\mathbf{z}^T \mathbf{N} \mathbf{z} - \mathbf{z}^T \begin{pmatrix} \mathbf{N} \boldsymbol{\gamma} \boldsymbol{\gamma}^T \mathbf{N} \\ \boldsymbol{\gamma}^T \mathbf{N} \boldsymbol{\gamma} \end{pmatrix} \mathbf{z} = \mathbf{a}^T \mathbf{a} - \frac{(\mathbf{a}^T \mathbf{b})^2}{\mathbf{b}^T \mathbf{b}} \quad (3.79)$$

The Cauchy-Schwarz inequality states that for any two vectors, \mathbf{x} and \mathbf{y} ,

⁴ W. C. Davidon, *USAEC Doc. ANL-5990* (rev.) Nov. 1959

⁵ R. Fletcher and M. J. D. Powell, *Computer J.* 6: 163, 1963

$$\mathbf{x}^T \mathbf{x} \geq \frac{(\mathbf{x}^T \mathbf{y})^2}{\mathbf{y}^T \mathbf{y}} \quad \text{thus} \quad \mathbf{a}^T \mathbf{a} - \frac{(\mathbf{a}^T \mathbf{b})^2}{\mathbf{b}^T \mathbf{b}} \geq 0 \quad (3.80)$$

So the first and third terms of (3.78) are positive. Now we need to show this for the second term,

$$\mathbf{z}^T \left(\frac{\Delta \mathbf{x} \Delta \mathbf{x}^T}{\Delta \mathbf{x}^T \boldsymbol{\gamma}} \right) \mathbf{z} = \frac{\mathbf{z}^T \Delta \mathbf{x} \Delta \mathbf{x}^T \mathbf{z}}{\Delta \mathbf{x}^T \boldsymbol{\gamma}} = \frac{(\mathbf{z}^T \Delta \mathbf{x})^2}{\Delta \mathbf{x}^T \boldsymbol{\gamma}} \quad (3.81)$$

The numerator of the right-most expression is obviously positive. The denominator can be written,

$$\Delta \mathbf{x}^T \boldsymbol{\gamma} = (\Delta \mathbf{x}^k)^T \nabla f^{k+1} - (\Delta \mathbf{x}^k)^T \nabla f^k = \underbrace{\alpha (\mathbf{s}^k)^T \nabla f^{k+1}}_{\text{term 1}} - \underbrace{\alpha (\mathbf{s}^k)^T \nabla f^k}_{\text{term 2}} \quad (3.82)$$

The second term in (3.82), $(\mathbf{s}^k)^T \nabla f^k$, is negative if the search direction goes downhill which it does if \mathbf{N}^k is positive definite, and with the minus sign is therefore positive. The first term in (3.82), $\alpha (\mathbf{s}^k)^T \nabla f^{k+1}$, can be positive or negative; however, it is zero if we are at α^* ; thus the entire expression in (3.82) is positive if we take a minimizing step, α^* .

We have now shown that all three terms of (3.78) are positive if we take a minimizing step. Thus, if \mathbf{N}^k is positive definite, \mathbf{N}^{k+1} is positive definite, etc.

7.4.3. DFP Update: Closing Remarks

The DFP update was popular for many years. As mentioned, we need to take a minimizing step to insure \mathbf{N} stays positive definite. Recall that we find α^* using a parabolic fit; on non-quadratics there is usually some error here. We can reduce the error by refitting the parabola several times as we obtain more points in the region of α^* . However, this requires more function evaluations. The DFP method is more sensitive to errors in α^* than the BFGS update, described in the next section, and can degrade if α^* is not accurate.

7.5. The Broyden Fletcher Goldfarb Shanno (BFGS) Update

The current "best" update is known as the Broyden, Fletcher, Goldfarb, Shanno or "BFGS" update, suggested by all four authors independently in 1970. It is also a rank 2 update. It has the same properties as the DFP update but is less sensitive to errors in α^* . This means we can be "sloppy" in our line search when we are far away from the optimum and the method still works well. This update is,

$$\mathbf{N}^{k+1} = \mathbf{N}^k + \left(1 + \frac{(\boldsymbol{\gamma}^k)^T \mathbf{N}^k \boldsymbol{\gamma}^k}{(\Delta \mathbf{x}^k)^T \boldsymbol{\gamma}^k} \right) \left(\frac{\Delta \mathbf{x}^k (\Delta \mathbf{x}^k)^T}{(\Delta \mathbf{x}^k)^T \boldsymbol{\gamma}^k} \right) - \frac{\Delta \mathbf{x}^k (\boldsymbol{\gamma}^k)^T \mathbf{N}^k + \mathbf{N}^k \boldsymbol{\gamma}^k (\Delta \mathbf{x}^k)^T}{(\Delta \mathbf{x}^k)^T \boldsymbol{\gamma}^k} \quad (3.83)$$

This update is currently considered to be the best update for use in optimization. It is the update inside OptdesX, Excel and many other optimization packages.

7.6. Comments About Quasi-Newton Methods

The quasi-Newton methods explained here combine the advantages of steepest descent and Newton's method without the disadvantages. They start out as steepest descent, which works well far from the optimum, and gradually become Newton's method, which works well near the optimum. They do this without requiring the evaluation of second derivatives. By insuring the update is positive definite, the search direction will always go downhill.

Note that these methods use information the previous methods "threw away." Quasi-Newton methods use differences in gradients and differences in \mathbf{x} to estimate second derivatives according to (3.34). This allows information from previous steps to correct (or update) the current step.

As mentioned, quasi-Newton methods are also methods of conjugate directions. This is shown in Section 9.4.

7.7. Hessian Updates Vs. Hessian Inverse Updates

All of the updates we have presented so far are updates for the Hessian Inverse. We can easily develop updates for the Hessian itself, as will be required for the SQP algorithm, starting from the condition

$$\boldsymbol{\gamma}^k = \mathbf{H}^k \Delta \mathbf{x}^k \quad (3.84)$$

instead of $(\mathbf{H}^{-1})^k \boldsymbol{\gamma}^k = \Delta \mathbf{x}^k$ which we used before. The BFGS Hessian approximation (Equation (3.83) is the Hessian inverse approximation) is given by,

$$\mathbf{H}^{k+1} = \mathbf{H}^k + \frac{\boldsymbol{\gamma}^k (\boldsymbol{\gamma}^k)^T}{(\boldsymbol{\gamma}^k)^T \Delta \mathbf{x}^k} - \frac{\mathbf{H}^k \Delta \mathbf{x}^k (\Delta \mathbf{x}^k)^T \mathbf{H}^k}{(\Delta \mathbf{x}^k)^T \mathbf{H}^k \Delta \mathbf{x}^k} \quad (3.85)$$

You will note that this looks a lot like the DFP Hessian inverse update but with \mathbf{H} interchanged with \mathbf{N} and $\boldsymbol{\gamma}$ interchanged with $\Delta \mathbf{x}$. In fact these two formulas are said to be *complementary* to each other.

8. The Conjugate Gradient Method

8.1. Definition

There is one more method we will learn, called the *conjugate gradient* method. We will present the results for this method primarily because it is an algorithm used in Microsoft Excel.

The conjugate gradient method is built upon steepest descent, except a correction factor is added to the search direction. The correction makes this method a method of conjugate directions. For the conjugate direction method, the search direction is given by,

$$\mathbf{s}^{k+1} = -\nabla f^{k+1} + \beta^k \mathbf{s}^k \quad (3.86)$$

Where β^k , a scalar, is given by

$$\beta^k = \frac{(\nabla f^{k+1})^T \nabla f^{k+1}}{(\nabla f^k)^T \nabla f^k} \quad (3.87)$$

8.2. Example: Conjugate Gradient Method

We will optimize our usual function, $f = x_1^2 - 2x_1x_2 + 4x_2^2$

$$\text{starting from } \mathbf{x}^0 = \begin{bmatrix} -3 \\ 1 \end{bmatrix} \quad \nabla f^0 = \begin{bmatrix} -8 \\ 14 \end{bmatrix}$$

We take a minimizing step in the negative gradient direction and stop at

$$\mathbf{x}^1 = \begin{bmatrix} -2.03 \\ -0.7 \end{bmatrix} \quad \nabla f^1 = \begin{bmatrix} -2.664 \\ -1.522 \end{bmatrix}$$

Now we calculate β^0 as

$$\beta^0 = \frac{(\nabla f^1)^T \nabla f^1}{(\nabla f^0)^T \nabla f^0} = \frac{\begin{bmatrix} -2.664 & -1.522 \end{bmatrix} \begin{bmatrix} -2.664 \\ -1.522 \end{bmatrix}}{\begin{bmatrix} -8 & 14 \end{bmatrix} \begin{bmatrix} -8 \\ 14 \end{bmatrix}} = \frac{9.413}{260} = 0.0362$$

We calculate the new search direction as,

$$\mathbf{s}^1 = -\nabla f^1 + \beta \mathbf{s}^0 = -\begin{bmatrix} -2.664 \\ -1.522 \end{bmatrix} + 0.0362 \begin{bmatrix} 8 \\ -14 \end{bmatrix} = \begin{bmatrix} 2.954 \\ 1.015 \end{bmatrix}$$

when we step in this direction, we arrive at the optimum, $\mathbf{x}^2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ $\nabla f^2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

The main advantage of the conjugate gradient method, as compared to quasi-Newton methods, is computation and storage. The conjugate gradient method only requires that we store the last search direction and last gradient, instead of a full matrix. Thus this method is a good one to use for large problems (say with 500 variables).

Although both conjugate gradient and quasi-Newton methods will optimize quadratic functions in n steps, on real problems quasi-Newton methods are better. Further, small errors can build up in the conjugate gradient method so some researchers recommend *restarting* the algorithm periodically (such as every n steps) to be steepest descent.

9. Appendix

9.1. The Gradient of a Quadratic Function in Vector Form

We define the coordinate vector to be,

$$\mathbf{e}_i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{A single 1 in the } i^{\text{th}} \text{ position} \quad (3.88)$$

We note that $\nabla x_i = \mathbf{e}_i$ so

$$\begin{aligned} \nabla \mathbf{x}^T &= [\nabla x_1, \nabla x_2, \dots, \nabla x_n] \\ &= [\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n] = \mathbf{I} \end{aligned} \quad (3.89)$$

Suppose we have a linear function:

$$f(\mathbf{x}) = a + \mathbf{b}^T \mathbf{x}$$

$$\text{then } \nabla f(\mathbf{x}) = \nabla(a + \mathbf{b}^T \mathbf{x}) = \underbrace{\nabla a}_{\text{term 1}} + \underbrace{\nabla(\mathbf{b}^T \mathbf{x})}_{\text{term 2}}$$

For the first term, since a is a constant, $\nabla a = 0$. Looking at the second term, from the rule for differentiation of a product,

$$\nabla(\mathbf{b}^T \mathbf{x}) = (\nabla \mathbf{b}^T) \mathbf{x} + (\nabla \mathbf{x}^T) \mathbf{b}$$

but $\nabla \mathbf{b}^T = \mathbf{0}^T$ and $\nabla \mathbf{x}^T = \mathbf{I}$

$$\begin{aligned}
 \text{Thus } \nabla f(\mathbf{x}) &= \nabla a + \nabla(\mathbf{b}^T \mathbf{x}) \\
 &= 0 + (\nabla \mathbf{b}^T) \mathbf{x} + (\nabla \mathbf{x}^T) \mathbf{b} \\
 &= 0 + 0 + \mathbf{I} \mathbf{b} \\
 &= \mathbf{b}
 \end{aligned} \tag{3.90}$$

Now suppose we have a quadratic function of the form:

$$q(\mathbf{x}) = a + \mathbf{b}^T \mathbf{x} + \frac{1}{2} \mathbf{x}^T \mathbf{H} \mathbf{x} \tag{3.91}$$

We wish to evaluate the gradient in vector form. We will do this term by term,

$$\nabla q(\mathbf{x}) = \nabla a + \nabla(\mathbf{b}^T \mathbf{x}) + \frac{1}{2} \nabla(\mathbf{x}^T \mathbf{H} \mathbf{x})$$

Applying the results from a linear function,

$$\begin{aligned}
 \nabla q(\mathbf{x}) &= \nabla a + \nabla(\mathbf{b}^T \mathbf{x}) + \frac{1}{2} \nabla(\mathbf{x}^T \mathbf{H} \mathbf{x}) \\
 &= 0 + \mathbf{b} + \frac{1}{2} \nabla(\mathbf{x}^T \mathbf{H} \mathbf{x})
 \end{aligned}$$

So we only need to evaluate the term, $\frac{1}{2} \nabla(\mathbf{x}^T \mathbf{H} \mathbf{x})$. If we split this into two vectors, i.e.

$\mathbf{u} = \mathbf{x}$, $\mathbf{v} = \mathbf{H} \mathbf{x}$, then

$$\nabla(\mathbf{x}^T \mathbf{H} \mathbf{x}) = (\nabla \mathbf{x}^T) \mathbf{v} + (\nabla \mathbf{v}^T) \mathbf{x}$$

We know $(\nabla \mathbf{x}^T) \mathbf{v} = \mathbf{I} \mathbf{H} \mathbf{x} = \mathbf{H} \mathbf{x}$, so we must only evaluate $(\nabla \mathbf{v}^T) \mathbf{x} = (\nabla(\mathbf{H} \mathbf{x})^T) \mathbf{x}$. We can write,

$$(\mathbf{H} \mathbf{x})^T = [\mathbf{h}_{r1}^T \mathbf{x}, \mathbf{h}_{r2}^T \mathbf{x}, \dots, \mathbf{h}_m^T \mathbf{x}]$$

where \mathbf{h}_{r1}^T represents the first row of \mathbf{H} , \mathbf{h}_{r2}^T represents the second row, and so forth.

Applying the gradient operator,

$$\nabla(\mathbf{H} \mathbf{x})^T = \left[\nabla(\mathbf{h}_{r1}^T \mathbf{x}), \nabla(\mathbf{h}_{r2}^T \mathbf{x}), \dots, \nabla(\mathbf{h}_m^T \mathbf{x}) \right]$$

From the previous result for $\nabla \mathbf{b}^T \mathbf{x}$, we know that $\nabla \mathbf{h}_{ri}^T \mathbf{x} = \mathbf{h}_{ri}$ since \mathbf{h}_{ri} is a vector constant. Therefore,

$$\begin{aligned}\nabla(\mathbf{H}\mathbf{x})^T &= [\mathbf{h}_{r1}, \mathbf{h}_{r2}, \dots, \mathbf{h}_m] \\ &= \mathbf{H}^T\end{aligned}$$

Returning now to the gradient of the expression, $q(\mathbf{x}) = a + \mathbf{b}^T \mathbf{x} + \frac{1}{2} \mathbf{x}^T \mathbf{H} \mathbf{x}$

$$\begin{aligned}\nabla q(\mathbf{x}) &= \nabla a + \nabla(\mathbf{b}^T \mathbf{x}) + \nabla\left(\frac{1}{2} \mathbf{x}^T \mathbf{H} \mathbf{x}\right) \\ &= 0 + \mathbf{b} + \frac{1}{2} \left\{ \nabla(\mathbf{x}^T) \mathbf{H} + \nabla(\mathbf{H}\mathbf{x})^T \right\} \mathbf{x} \\ &= \mathbf{b} + \frac{1}{2} (\mathbf{H} + \mathbf{H}^T) \mathbf{x} \\ &= \mathbf{b} + \mathbf{H}\mathbf{x}\end{aligned}\tag{3.92}$$

If the quadratic we are approximating is a Taylor expansion,

$$f^{k+1} = f^k + (\nabla f^k)^T \Delta \mathbf{x}^k + \frac{1}{2} (\Delta \mathbf{x}^k)^T \mathbf{H}^k \Delta \mathbf{x}^k$$

Then (3.92) is:

$$\nabla f^{k+1} = \nabla f^k + \mathbf{H}^k \Delta \mathbf{x}^k\tag{3.93}$$

9.2. Optimal Step Length for Quadratic Function

In this section we will derive (3.12). If we start with a Taylor expansion,

$$f^{k+1} = f^k + (\nabla f^k)^T \Delta \mathbf{x}^k + \frac{1}{2} (\Delta \mathbf{x}^k)^T \mathbf{H} \Delta \mathbf{x}^k\tag{3.94}$$

When we do a line search,

$$\Delta \mathbf{x}^k = \alpha \mathbf{s}\tag{3.95}$$

Substituting (3.95) into (3.94) gives

$$f^{k+1} = f^k + (\nabla f^k)^T \alpha \mathbf{s} + \frac{1}{2} (\alpha \mathbf{s})^T \mathbf{H} \alpha \mathbf{s}$$

If we take the derivative of this expression with respect to α (a scalar),

$$\frac{df^{k+1}}{d\alpha} = (\nabla f^k)^T \mathbf{s} + \alpha \mathbf{s}^T \mathbf{H} \mathbf{s}\tag{3.96}$$

Setting the derivative equal to zero and solving for α gives:

$$\alpha^* = -\frac{(\nabla f^k)^T \mathbf{s}}{\mathbf{s}^T \mathbf{H} \mathbf{s}} \quad (3.97)$$

9.3. Proof that a Method of Conjugate Directions Minimizes the Current Subspace

THEOREM. A conjugate direction method is such that each \mathbf{x}^{k+1} is the minimizer in the subspace generated by \mathbf{x}^0 and the directions, $\mathbf{s}^0, \mathbf{s}^1, \dots, \mathbf{s}^k$, i.e.

$$\mathbf{x}^{k+1} = \mathbf{x}^0 + \sum \alpha^i \mathbf{s}^i \quad i = 0, 1, \dots, k.$$

We wish to show that,

$$(\nabla f^{k+1})^T \mathbf{s}^i = 0 \quad \text{for all } i \leq k \quad (3.98)$$

which indicates that we have zero slope along any search direction in the subspace generated by \mathbf{x}^0 and the search directions $\mathbf{s}^0, \mathbf{s}^1, \dots, \mathbf{s}^k$, i.e.,

$$\frac{\partial f}{\partial \alpha^i} = 0 \quad \text{for all } i \leq k$$

PROOF. The proof by induction. Given the usual expression for the gradient of a Taylor expansion,

$$\nabla f^{k+1} = \nabla f^k + \mathbf{H} \Delta \mathbf{x}^k$$

Which we will write as,

$$\nabla f^{k+1} = \nabla f^k + \alpha \mathbf{H} \mathbf{s}^k \quad (3.99)$$

If we multiply both sides by \mathbf{s}^k

$$(\mathbf{s}^k)^T \nabla f^{k+1} = (\mathbf{s}^k)^T \nabla f^k + \alpha (\mathbf{s}^k)^T \mathbf{H} \mathbf{s}^k = 0$$

By definition of α^k this is true for $i=k$. For $i < k$,

$$(\mathbf{s}^i)^T \nabla f^{k+1} = \underbrace{(\mathbf{s}^i)^T \nabla f^k}_{\text{term 1}} + \alpha \underbrace{(\mathbf{s}^i)^T \mathbf{H} \mathbf{s}^k}_{\text{term 2}}$$

Term 1 vanishes by the induction hypothesis, while term 2 vanishes from the definition of conjugate directions.

9.4. Proof that an Update with the Hereditary Property is Also a Method of Conjugate Directions

THEOREM. An update with the hereditary property and exact line searches is a method of conjugate directions and therefore terminates after $m \leq n$ iterations on a quadratic function.

We assume that the hereditary property holds for $k = 1, 2, \dots, m$

$$\mathbf{N}^{k+1}\boldsymbol{\gamma}^i = \Delta\mathbf{x}^i \quad \text{for all } i \leq k \quad (3.100)$$

We need to show that conjugacy holds as well,

$$(\mathbf{s}^k)^T \mathbf{H}\mathbf{s}^i = 0 \quad \text{for all } i \leq k-1 \quad (3.101)$$

The proof is by induction. We will show that if \mathbf{s}^k is conjugate then \mathbf{s}^{k+1} is as well, i.e.,

$$(\mathbf{s}^{k+1})^T \mathbf{H}\mathbf{s}^i = 0 \quad \text{for all } i \leq k \quad (3.102)$$

We note that

$$\mathbf{s}^{k+1} = -\mathbf{N}^{k+1}\nabla f^{k+1} \quad (3.103)$$

by definition of the quasi-Newton method. Or taking the transpose,

$$(\mathbf{s}^{k+1})^T = -(\nabla f^{k+1})^T \mathbf{N}^{k+1} \quad (3.104)$$

Substituting (3.104) into (3.102);

$$(\mathbf{s}^{k+1})^T \mathbf{H}\mathbf{s}^i = -(\nabla f^{k+1})^T \mathbf{N}^{k+1} \mathbf{H}\mathbf{s}^i \quad \text{for all } i \leq k \quad (3.105)$$

Also,

$$\mathbf{H}\mathbf{s}^i = \frac{\mathbf{H}\Delta\mathbf{x}^i}{\alpha^i} = \frac{\boldsymbol{\gamma}^i}{\alpha^i}$$

so (3.105) becomes,

$$(\mathbf{s}^{k+1})^T \mathbf{H}\mathbf{s}^i = -\frac{(\nabla f^{k+1})^T \mathbf{N}^{k+1} \boldsymbol{\gamma}^i}{\alpha^i} \quad \text{for all } i \leq k \quad (3.106)$$

From the hereditary property we have $\mathbf{N}^{k+1}\boldsymbol{\gamma}^i = \Delta\mathbf{x}^i \quad i \leq k$, so (3.106) can be written,

$$(\mathbf{s}^{k+1})^T \mathbf{H} \mathbf{s}^i = - \left[\frac{(\nabla f^{k+1})^T \Delta \mathbf{x}^i}{\alpha^i} \right] = 0 \quad \text{for all } i \leq k$$

The term in brackets is zero for all values of $i = 1, 2, \dots, k - 1$ from the assumption the previous search direction was conjugate which implies (3.98). It is zero for $i = k$ from the definition of α^* . Thus if we have conjugate directions at k , and the hereditary property holds, we have conjugate directions at $k+1$.

10. References

For more information on unconstrained optimization and in particular Hessian updates, see:

R. Fletcher, *Practical Methods of Optimization, Second Edition*, Wiley, 1987.

D. Luenberger, and Y. Ye, *Linear and Nonlinear Programming, Third Edition*, 2008.

CHAPTER 4

INTRODUCTION TO DISCRETE VARIABLE OPTIMIZATION

1. Introduction

1.1. Examples of Discrete Variables

One often encounters problems in which design variables must be selected from among a set of discrete values. Examples of discrete variables include catalog or standard sizes (I beams, motors, springs, fasteners, pipes, etc.), materials, and variables which are naturally integers such as people, gear teeth, number of shells in a heat exchanger and number of distillation trays in a distillation column. Many engineering problems are discrete in nature.

1.2. Solving Discrete Optimization Problems

At first glance it might seem solving a discrete variable problem would be easier than a continuous problem. After all, for a variable within a given range, a set of discrete values within the range is finite whereas the number of continuous values is infinite. When searching for an optimum, it seems it would be easier to search from a finite set rather than from an infinite set.

This is not the case, however. Solving discrete problems is harder than continuous problems. This is because of the combinatorial explosion that occurs in all but the smallest problems. For example if we have two variables which can each take 10 values, we have $10 * 10 = 10^2 = 100$ possibilities. If we have 10 variables that can each take 10 values, we have 10^{10} possibilities. Even with the fastest computer, it would take a long time to evaluate all of these. Obviously we need better strategies than just exhaustive search.

1.2.1. Example: Standard I Beam Sections

There are 195 standard I beam sections. If we have a structure with 20 different beam sizes, how many combinations of beams are possible?

$$195^{20} = 6.3 * 10^{45}$$

Since the number of grains of sand on the earth is estimated to be around $1 * 10^{25}$, this is a big number!

1.3. Related Discrete Variables

We also need to distinguish between discrete variables and *related discrete variables*. Often two discrete variables are related, i.e. the discrete values are somehow tied to each other. For example, suppose we wish to select a pipe, described by the thickness and diameter, from among a set of standard sizes. This makes thickness and diameter discrete. It also makes them *related*, because certain values of thickness are matched to certain diameters and vice-versa. In general, as diameters increase, the available values for thickness increase as well. Material properties also represent related discrete variables. When we pick a certain material, the modulus, density, yield strength, etc. are also set. These material properties are related to

each other. We cannot match, for example, the density of aluminum with the modulus for steel. When we have related discrete variables, we have discrete variables that fix the values of several variables at once.

2. Discrete Optimization with Branch and Bound

2.1. Description of Branch and Bound Algorithm

A classical method for handling discrete problem is called *Branch and Bound*. The word “branch” refers to a tree structure that is built. The word “bound” refers to an estimate of the objective function which is used to prune the tree. Branch and Bound requires that we have an efficient continuous optimization algorithm, which is called many times during the course of the discrete optimization.

The branch and bound strategy works by developing a tree structure. Initially, at the root of the tree, only one discrete variable is allowed to take on discrete values: other discrete variables are modeled as continuous. At each level in the tree one more discrete variable is made discrete. The various combinations of values for discrete variables constitute nodes in the tree.

We start by progressing down the tree according to the discrete variable combinations that appear to be the best. At each node, an optimization problem is performed for any continuous variables and those discrete variables modeled as continuous at that node. Assuming we are minimizing, the objective value of this optimization problem becomes a *lower bound* for any branches below that node, i.e. the objective value will *underestimate* (or, at best, be equal to) the true value of the objective since, until we reach the bottom of the tree, some of the discrete variables are modeled as continuous. Once a solution has been found for which all discrete variables have discrete values (we reach the bottom of the tree), then any node which has an objective function higher than the solution in hand can be *pruned*, which means that these nodes don't have to be considered further.

As an example, suppose we have 3 discrete variables: variables 1 and 2 have 3 possible discrete values and variable 3 has 4 possible discrete values. A branch and bound tree might look like Fig. 4.1 given below.

In this tree, "Level 1" represents the nodes where variable 1 is allowed to be discrete and variables 2 and 3 are continuous. For "Level 2," variables 1 and 2 are discrete; only variable 3 is continuous. In "Level 3," all variables are discrete. Each circle is a node. The numbers in the circles show the order in which the nodes were evaluated. The number shown at the upper right of each circle is the optimum objective value for the optimization problem performed at the node. An asterisk means no feasible solution could be found to the optimization problem; a double underscore indicates the node was pruned.

At the first level, three optimizations are performed with variable 1 at its 1st, 2nd and 3rd discrete values respectively (variables 2 and 3 continuous). The best objective value was obtained at node 3. This node is expanded further. Nodes 4, 5, 6 correspond to variable 1 at its 3rd discrete value and variable 2 at its 1st, 2nd and 3rd discrete values respectively, with variable 3 continuous. The best objective value for nodes 1, 2, 4, 5, and 6 is at node 1, so it is

expanded into nodes 7, 8, and 9. Now the best objective value among unexpanded nodes is at node 5, so it is expanded. Nodes 10, 11, 12, 13 correspond to variable 1 at its 3rd discrete value, variable 2 at its 2nd discrete value, and variable 3 at its 1st, 2nd, 3rd, and 4th values. The best objective value, 59, is obtained at node 11. This becomes the temporary optimal solution. Any nodes with objectives higher than 59 can automatically be pruned since the objective only increases as we go down the tree from a node and make more and more variables discrete. Thus nodes 2, 7, 8 are pruned. Node 9, however, looks promising, so we expand this node. As is shown, we eventually find at node 16, with variable 1 at its 1st discrete value, variable 2 at its 3rd discrete value, and variable 3 at its 3rd discrete value, a better optimum than we had before. At this point all further nodes are pruned and we can stop.

It should be clear that Branch and Bound gains efficiency by pruning nodes of the tree which have higher objective values (bounds) than known discrete solutions. In realistic problems, the majority of nodes are pruned—in the example which follows less than 0.001% of the nodes were expanded.

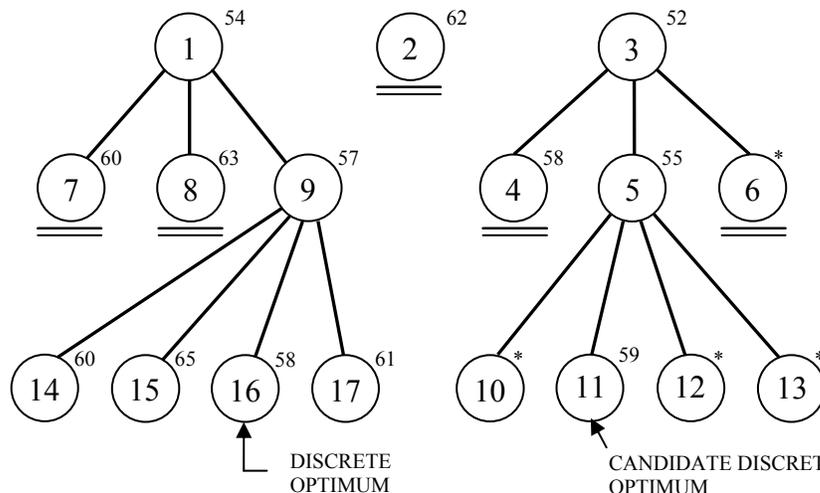
However, there is a cost—at each node we have to perform an optimization, which could involve many calls to the analysis program. One way to reduce this cost is by making a linear approximation to the actual problem and optimizing the linear approximation. This greatly reduces the number of analysis calls, but at some expense in accuracy.

Another way to reduce the size of the tree is to select discrete value neighborhoods around the continuous optimum. It is likely the discrete solution for a particular variable is close to the continuous one. Selecting neighborhoods of the closest discrete values makes it possible to further restrict the size of the problem and reduce computation.

SCHEMATIC OF BRANCH AND BOUND METHOD

* INFEASIBLE == PRUNED

LEVEL 1:
(1 Var Discrete)



LEVEL 2:
(2 Vars Discrete)

LEVEL 3:
(3 Vars Discrete)

Fig. 4.1 An example Branch and Bound Tree

3. Exhaustive Search

The exhaustive search strategy examines all possible combinations of discrete variables to locate the discrete optimum. In terms of a branch and bound tree, exhaustive search examines those nodes found at the bottom of an un-pruned branch and bound tree structure. If the branch and bound strategy is performed without much pruning, more nodes could be evaluated and thus more time required than exhaustive search. In the above example, 17 nodes were evaluated with branch and bound; exhaustive search would have required the evaluation of $3 \times 3 \times 4 = 36$ nodes. The efficiency of branch and bound relative to exhaustive search therefore depends on how much pruning is done.

4. Discrete Variables in OptdesX

4.1. Setting Up an Optimization Problem

Discrete variables are handled in OptdesX by mapping design variables to be discrete variables. This is accomplished by pressing the “C” (Continuous) button so it changes to “D” (discrete). The bottom right part of the window opens to show the discrete variables.

Related Discrete Variables are handled in OptdesX as a many-to-one mapping of design variables to a discrete variable. The discrete variables appear in the lower right hand side of the Variables window. For example in Fig. 4.2 below, width and diameter have been made discrete. Width is associated with a file called “widths.” Diameter is associated with a file called “pipes.” When OptdesX reads the file pipes, it finds two columns in the file which represent diameter and thickness. It concludes this discrete variable is a related discrete variable. It then opens another row (NULL VARIABLE below) and expects the second variable to be specified by clicking on its “C” button. The user would click on thickness, since it is diameter and thickness values which are in the file “pipes,” and thickness would take the place of NULL VARIABLE.

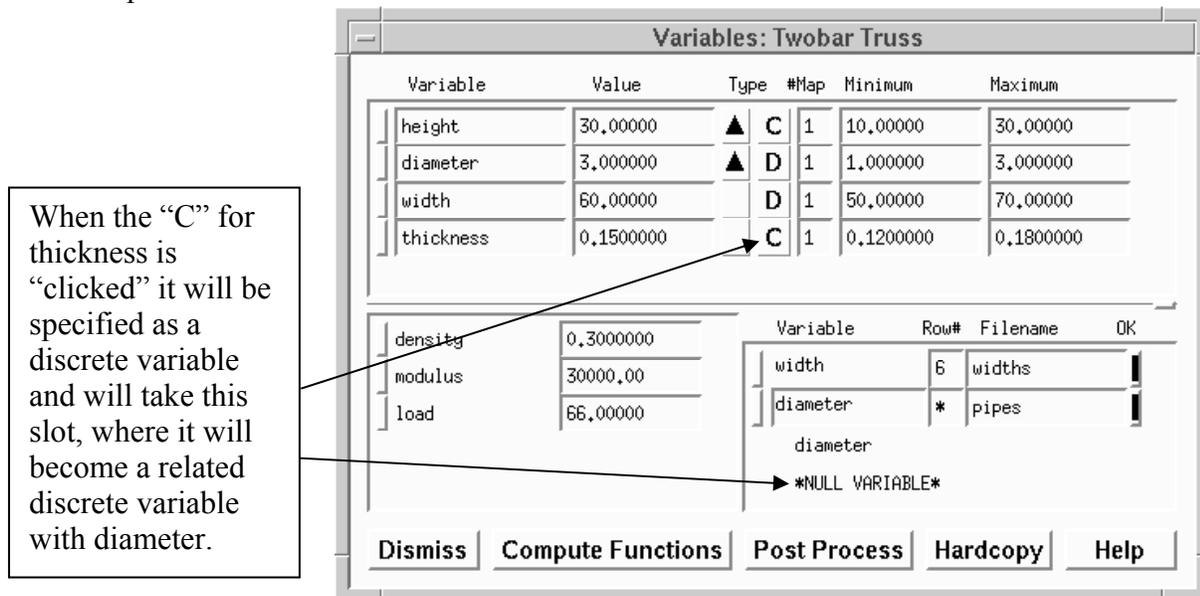


Fig. 4.2 The Variables window with discrete variables.

4.2. Providing Discrete Values in Files

In OptdesX we will specify discrete values for discrete variables in separate file(s). The first line of a discrete value file contains the number of rows and the number of columns. If the discrete variable is not a related discrete variable, the number of columns is 1. The discrete values then follow, one value per line.

Related discrete variables are specified by having multiple entries per line. For example, consider the following example data file:

```
19 2
3.0 .30
3.0 .28
3.0 .26
2.5 .28
2.5 .26
2.5 .24
2.5 .22
2.0 .24
2.0 .22
2.0 .20
2.0 .18
1.5 .20
1.5 .18
1.5 .16
1.5 .14
1.0 .16
1.0 .14
1.0 .12
1.0 .10
```

Note that the first line has two values. The first value specifies the number of rows (discrete values) and the second number specifies the number of columns (number of discrete variables which are related). Each subsequent line refers to a discrete value. In this instance the file contains the discrete data for the tubes of the Two-bar truss, where each tube is described by a diameter and thickness. Thus the third line of the file indicates we can have a tube with a diameter of 3.0 and a thickness of 0.28. A regular, unrelated discrete variable would only have one column of data.

4.3. Summary of Steps for Branch and Bound Algorithm with OptdesX

Step 1: Define the discrete variable files; set up the discrete problem and specify a starting point.

Step 2: Optimize to find the continuous optimum.
Standard algorithms (GRG, SQP) are executed to find the continuous optimum.

Step 3: Select neighborhoods.

For some problems the number of discrete combinations is too large to practically consider all possibilities. The user has the option to consider a neighborhood of discrete values within a certain distance of the continuous optimum. As the user selects the neighborhood radius in scaled space, OptdesX will display how many discrete values fall within that neighborhood.

Step 4: Select linearization option if desired.

The number of variable combinations may be so large as to prohibit a nonlinear optimization at each node. Therefore, the user can linearize the objective and constraints about the continuous optimum. Then at each node a LP optimization is done--this doesn't require any additional calls to the analysis subroutine. A nonlinear optimization is performed for the current best solution to verify actual feasibility. Linearized strategies are thus faster but involve approximations that may lead to solutions that are not the true discrete optimum.

Step 5: Perform Branch and Bound optimization.

The branch and bound algorithm is executed and the tree is created as the optimization proceeds.

5. Example: Design of 8 Story 3 Bay Planar Frame

Prof. Richard Balling applied Branch and Bound to the design of a 2D structure. The objective was to minimize weight, subject to AISC combined stress constraints.

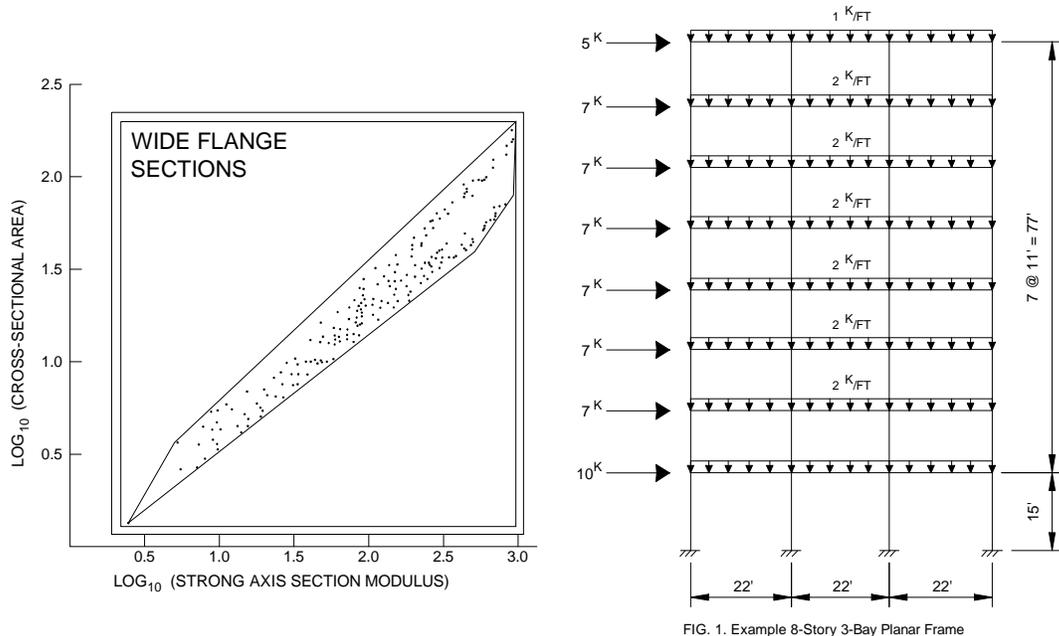


Fig. 4.3 Schematic of eight story 2D frame, and graph showing band of possible discrete values.

Each beam can take one of 195 possible sizes. It was assumed the same size girder continues across each floor. The same size column continues for two stories. Exterior columns were assumed the same size; interior columns were also assumed the same size. These assumptions resulted in 16 separate members; for each member we needed to specify A, I, S, giving a total of 48 design variables (but only 16 related discrete variables). The constraints represented combined stress constraints for each member.

For the starting design all members were W12x65. The weight was 58,243 lbs. There were 13 violated stress constraints.

Step 1: The discrete files were created and the problem set up.

Step 2: A continuous optimization was performed. Using GRG, an optimum was achieved after 9 iterations and 486 analyses. The weight was 36,253 lbs. There were 20 binding stress constraints.

Step 3: Neighborhoods were chosen to reduce problem size. Neighborhoods were sized so that each member had 3 to 5 discrete combinations.

Step 4: Linearization was selected to further reduce computation.

Step 5: Branch and bound was executed. The optimum required 3066 linear optimizations which represented only 0.001% of the possible nodes. The discrete optimum weighed 40,337 lbs. and had 5 binding stress constraints.

6. Simulated Annealing

6.1. Introduction

Branch and Bound and Exhaustive Search both suffer from a serious problem—as the number of variables increases, the number of combinations to be examined explodes. Clearly this is the case for Exhaustive Search, which does nothing to reduce the size of the problem. The same is true for Branch and Bound, although to a lesser degree, because of the pruning and approximations which are employed. Even if Branch and Bound reduces the search space by 99.99%, however, many problems are still too large. Consider, for example, a problem with 20 discrete variables which can each take on 25 values. The number of combinations equals,

$$25^{20} = 9.1 * 10^{27}$$

If we can reduce the search space by 99.99%, we still must search $9 * 10^{23}$ combinations! In general then, algorithms which try to search the entire combinatorial space can easily be overwhelmed by the sheer size of the problem. In contrast, the *evolutionary algorithms* we study in this and the following sections do not suffer from this problem. These algorithms are so-named because they mimic natural processes that govern how nature evolves. These algorithms do not attempt to examine the entire space. Even so, they have been shown to provide good solutions.

Simulated annealing copies a phenomenon in nature--the annealing of solids--to optimize a complex system. Annealing refers to heating a solid to a liquid state and then cooling it slowly so that thermal equilibrium is maintained. Atoms then assume a nearly globally minimum energy state. In 1953 Metropolis created an algorithm to simulate the annealing process. The algorithm simulates a small random displacement of an atom that results in a change in energy. If the change in energy is negative, the energy state of the new configuration is lower and the new configuration is accepted. If the change in energy is positive, the new configuration has a higher energy state; however, it may still be accepted according to the Boltzmann probability factor:

$$P = \exp\left(\frac{-\Delta E}{k_b T}\right) = e^{\left(-\Delta E/k_b T\right)} \quad (4.1)$$

where k_b is the Boltzmann constant and T is the current temperature. By examining this equation we should note two things: the probability is proportional to temperature--as the solid cools, the probability gets smaller; and inversely proportional to ΔE --as the change in energy is larger the probability of accepting the change gets smaller.

When applied to engineering design, an analogy is made between energy and the objective function. The design is started at a high “temperature,” where it has a high objective (we assume we are minimizing). Random perturbations are then made to the design. If the objective is lower, the new design is made the current design; if it is higher, it may still be accepted according the probability given by the Boltzmann factor. The Boltzmann probability is compared to a random number drawn from a uniform distribution between 0 and 1; if the random number is smaller than the Boltzmann probability, the configuration is accepted. This allows the algorithm to escape local minima.

As the temperature is gradually lowered, the probability that a worse design is accepted becomes smaller. Typically at high temperatures the gross structure of the design emerges which is then refined at lower temperatures.

Although it can be used for continuous problems, simulated annealing is especially effective when applied to combinatorial or discrete problems. Although the algorithm is not guaranteed to find the best optimum, it will often find near optimum designs with many fewer design evaluations than other algorithms. (It can still be computationally expensive, however.) It is also an easy algorithm to implement.

Fig. 4.4 below shows how the weight of a structure changed as simulated annealing was used to select from among discrete structural members. Each cycle represents a temperature. It can be seen that in earlier cycles worse designs were accepted more often than in later cycles.

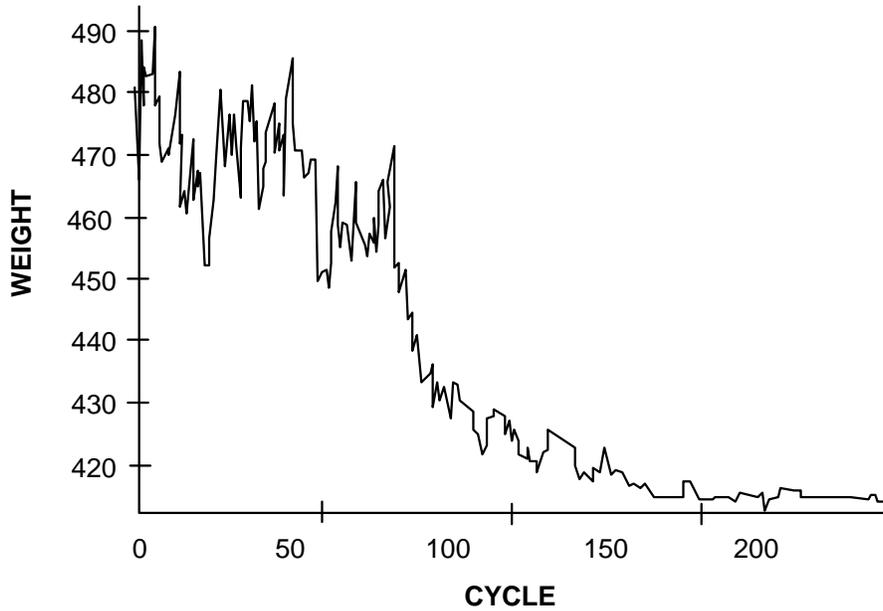


Fig. 4.4 Change in weight of structure during simulated annealing.

The analogy between annealing and simulated annealing is summarized in the table below (Liu, 1990).

Annealing objective – minimum energy configuration	Simulated annealing objective - minimum cost
Annealing Boltzmann equation $P = \exp\left(\frac{-\Delta E}{k_B T}\right) = e^{\left(-\Delta E/k_B T\right)}$	Simulated annealing Boltzmann equation $P = \exp\left(\frac{-\Delta E}{\Delta E_{avg} T}\right) = e^{\left(-\Delta E/\Delta E_{avg} T\right)}$
P is the probability that an atom will move from a lower to a higher energy state	P is the probability that a higher cost design will be accepted
ΔE is the change in energy to go from a lower energy state to a higher one	ΔE is the cost difference between the current design and the previous one
T is the absolute current annealing temperature; it correlates to the amount of mobility of the molecules or atoms	T is a unitless value; it correlates to the mobility of the optimization process to accept a higher cost design
k_B is the Boltzmann constant	ΔE_{avg} is the running average value of the ΔE ; it normalizes the change in the objective (ΔE)

6.2. Algorithm Description

6.2.1. Selecting Algorithm Parameters

In the above table, notice that instead of (4.1), we use the following to estimate Boltzmann probability:

$$P = \exp\left(\frac{-\Delta E}{\Delta E_{avg} T}\right) = e^{\left(\frac{-\Delta E}{\Delta E_{avg} T}\right)} \quad (4.2)$$

We see that this equation includes ΔE_{avg} instead of k . The constant ΔE_{avg} is a running average of all of the “accepted” ΔE (objective changes) to this point in the optimization. It normalizes the change in the objective, ΔE , in (4.2).

Equation (4.2) is also a function of a “temperature,” T . How is this chosen? We recommend you set the probability level, P_s , that you would like to have at the beginning of the optimization that a worse design could be accepted. Do the same for the probability level at the end of the optimization, P_f . Then, if we assume $\Delta E = \Delta E_{avg}$, (which is clearly true at the start of the optimization),

$$T_s = \frac{-1}{\ln(P_s)} \quad T_f = \frac{-1}{\ln(P_f)} \quad (4.3)$$

Select the total number of cycles, N , you would like to run. Each cycle corresponds to a temperature. Decrease temperature according to the expression,

$$T_{n+1} = F \cdot T_n \quad F = \left(\frac{T_f}{T_s}\right)^{1/(N-1)} \quad (4.4)$$

where T_{n+1} is the temperature for the next cycle and T_n is the current temperature. Note that the design should be perturbed at each temperature until “steady state” is reached. Since it is not clear when steady state is reached for a design, we recommend perturbing the design at least n (n = no. of design variables) or more if computationally feasible.

6.2.2. Example: Choosing Parameters for Simulated Annealing

We pick the following:

$$P_s = 0.5 \quad P_f = 10^{-8} \quad N = 100$$

and using (4.3) and (4.4) we calculate,

$$T_s = 1.4426 \quad T_f = 0.054278 \quad F = 0.9674$$

6.2.3. Algorithm Steps

1. Choose a starting design.
2. Select P_s , P_f , N , and calculate T_s , T_f and F .
3. Randomly perturb the design to different discrete values close to the current design.
4. If the new design is better, accept it as the current design.
5. If the new design is worse, generate a random number between 0 and 1 using a uniform distribution. Compare this number to the Boltzmann probability. If the random number is lower than the Boltzmann probability, accept the worse design as the current design.
6. Continue perturbing the design randomly at the current temperature until “steady state” is reached.
7. Decrease temperature according to $T_{n+1} = F \cdot T_n$
8. Go to step 3.
9. Continue the process until T_f is reached.

In the early stages, when the temperature is high, the algorithm has the freedom to “wander” around design space. Accepting worse designs allows it to escape local minima. As temperature is decreased the design becomes more and more “frozen” and confined to a particular region of design space.

A diagram of the algorithm is given in Fig. 4.5:

Simulated Annealing

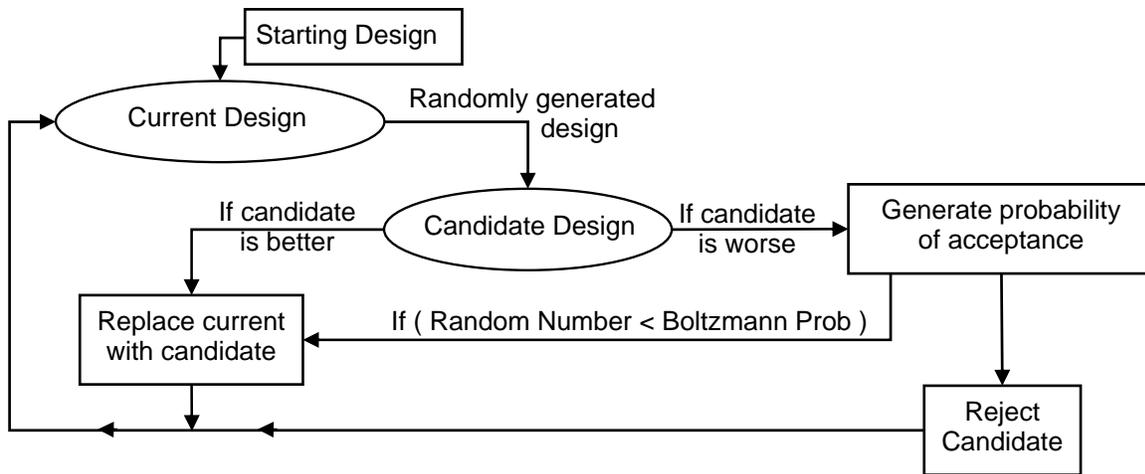


Fig. 4.5. The simulated annealing algorithm.

6.2.4. Limitations of Simulated Annealing

Simulated annealing is really developed for unconstrained problems. Questions arise when applied to constrained problems--if the perturbed design is infeasible, should it still be accepted? Some implementations automatically reject a design if it is infeasible; others use a

penalty function method so the algorithm “naturally” wants to stay away from infeasible designs.

Simulated annealing does not use any gradient information. Thus it is well suited for discrete problems. However, for continuous problems, if gradient information is available, a gradient-based algorithm will be much (>100 times) faster.

6.3. Examples of Simulated Annealing

Balling describes the optimization of a 3D, unsymmetric 6 story frame, shown below.

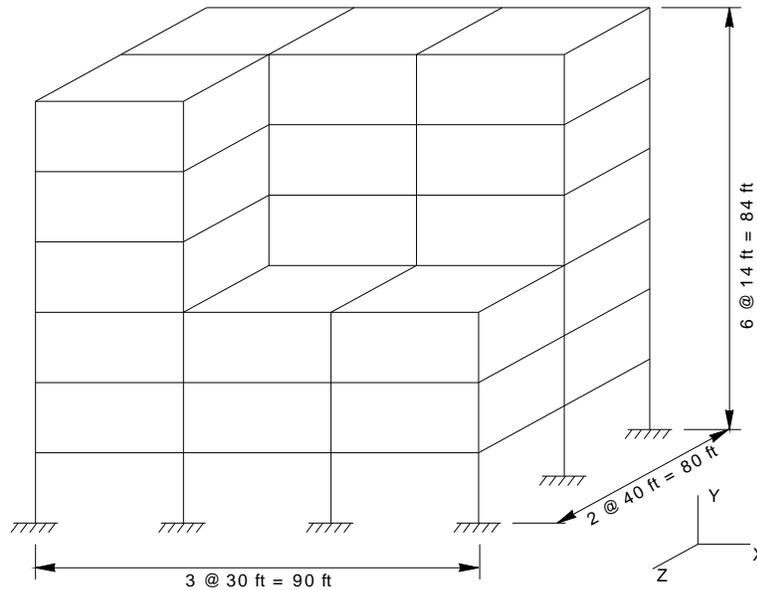


Fig. 4.6. Six story frame.

The 156 members were grouped into 11 member groups--7 column groups and 4 beam groups. Beams and columns must be selected from a set of 47 economy sections for beams and columns respectively. The starting design had a weight of 434,600 lbs. Eleven perturbations were examined at each temperature, and with $N = 100$, an optimization required 1100 analyses. Two iterations of simulated annealing were performed, with the starting design of the second iteration being the optimum from the first. The results were as follows:

Iteration	Optimal Weight	Execution Time
1	416,630 lbs.	1:24:09
2	414,450 lbs.	1:26:24
Total		2:50:33

The change in weight observed as temperature was decreased for the first iteration was very similar to the diagram given Fig. 4.4.

Simulated annealing was compared to the branch and bound strategy. First a continuous optimization was performed. Each design variable was then limited to the nearest 4 discrete variables. To reduce computation, a linear approximation of the structure was made using information at the continuous optimum. Because the neighborhoods were so small, the algorithm was run 4 iterations, where the starting point for the next iteration was the optimum from the current iteration.

Iteration	Optimal Weight	Execution Time
1	420,410 lbs.	0:13:44
2	418,180 lbs.	1:09:44
3	414,450 lbs.	0:12:24
Total		1:35:52

Liu used simulated annealing for the discrete optimization of pipe network systems. Pipes, like wide flange sections, are only manufactured in certain sizes. For this work, each pipe could be selected from 30 possible values.

An example network is shown in Fig. 4.7. This network has 22 pipe sizes to be chosen.

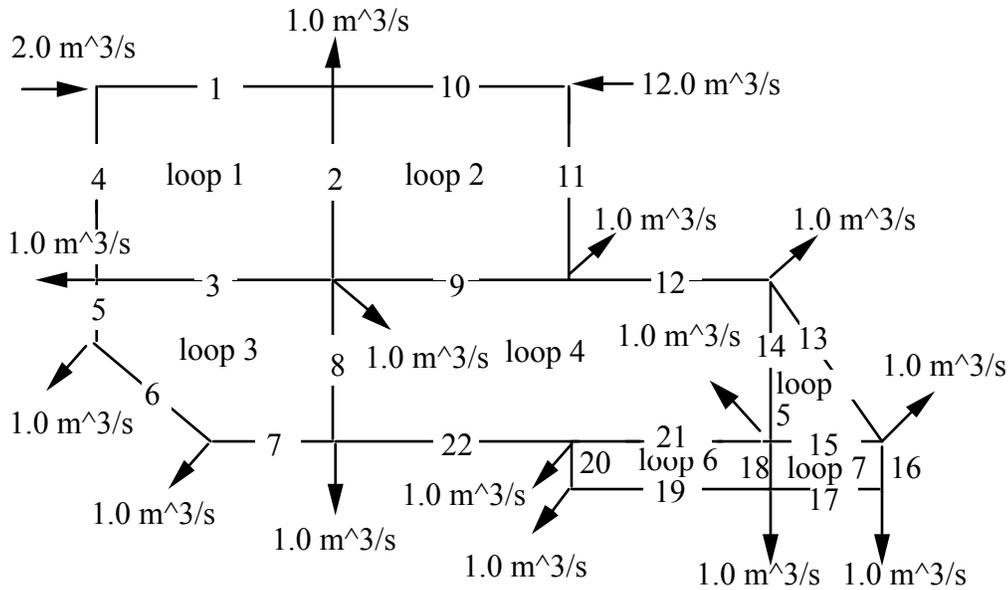


Fig. 4.7 Pipe network optimized with simulated annealing

Arrows give required flows that must enter or leave the network. Simulated annealing was used to find the network with the least cost that could meet these demands. For this problem, $P_s = 0.9$, $F=0.9$, $N=65$, and 5 perturbed designs were evaluated at each temperature. For this optimization 7221 analysis calls to the network flow solver were required.

The change in cost during optimization is given below.

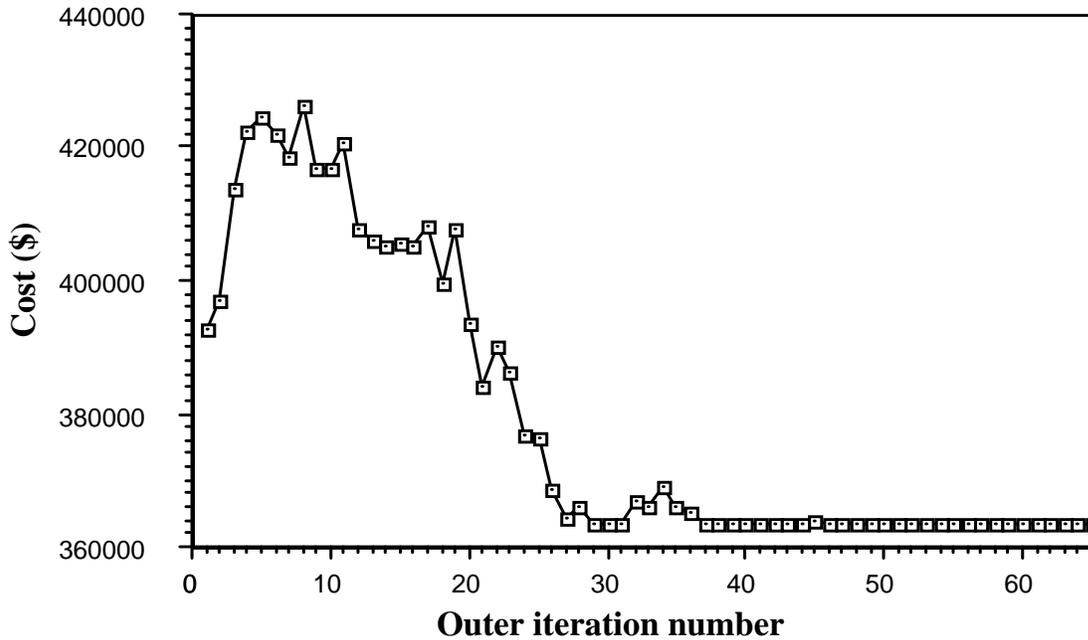


Fig. 4.8 Simulated annealing history for pipe network

Liu also compared the results of simulated annealing to branch and bound:

Comparison of continuous and branch and bound results with simulated annealing.
Ten different starting points.

Run	OptdesX continuous cost (\$)	OptdesX branch and bound cost (\$)	Simulated annealing cost (\$)
1	42729	45279	48687
2	44101	47013	47013
3	43241	46173	45288
4	44101	46770	45893
5	43241	46173	46080
6	44101	47013	45257
7	43242	46175	46990
8	43241	45275	47013
9	44097	47013	45279
10	44102	47013	45279

For branch and bound, local neighborhoods of 3 to 5 pipes around the continuous optimum were used.

7. Classical Genetic Algorithms

7.1. Introduction

Genetic Algorithms (GA) are based on the Darwinian theory of natural selection. The search algorithm mimics reproduction, crossover, and mutations in nature. The roots of genetic algorithms are traced to work done by (Holland, 1975). Taking a quote from (Davis, 1987):

“In nature species are searching for beneficial adaptations to a complicated and changing environment. The “knowledge” that each species gains with a new generation is embodied in the makeup of chromosomes. The operations that alter the chromosomal makeup are applied when parent reproduce; among them are random mutation, inversion of chromosomal material and crossover--exchange of chromosomal material between two parents’ chromosomes. Random mutation provides background variation and occasionally introduces beneficial material into a species’ chromosomes. Inversion alters the location of genes on a chromosome, allowing genes that are co-adapted to cluster on a chromosome, increasing their probability of moving together during crossover. Crossover exchanges corresponding genetic material from two parent chromosomes, allowing beneficial genes on different parents to be combined in their offspring.”

Goldberg has suggested four ways that genetic algorithms are different from traditional derivative-based algorithms:

- GA’s work with a coding of the variables, not the variables themselves.
- GA’s search from a population of points, not a single point.
- GA’s use objective function information, not derivatives or other auxiliary knowledge.
- GA’s use probabilistic transition rules, not deterministic rules.

As given in Gen (2000), there are five basic components to a genetic algorithm:

1. A genetic representation of solutions to the problem.
2. A way to create an initial population of solutions.
3. An evaluation function rating solutions in terms of the fitness.
4. Genetic operators that alter the genetic composition of children during reproduction.
5. Values for parameters of genetic algorithms.

In the next section all of the components will be specified as we step through the algorithm.

7.2. Steps of the Classical Algorithm

1. Determine a coding for the design. The classical algorithm uses binary coding. A design is coded as a “chromosome.”
2. Develop the initial population. This can be done by randomly creating a set of designs which are evenly spread through the design space. A population of 20 to 100 designs often works well.

3. Pick a crossover and mutation rate. Typical values are 0.8 and 0.01, respectively. These are problem dependent, however, and are often determined experimentally.
4. Select a way to measure the “fitness” or goodness of a design. Often we will just use the objective value. (In Chapter 5, we will learn other ways of measuring fitness.)
5. Select the mating pool. These will be the designs which will be chosen to become parents and produce the next generation. This selection can be done several ways. Two of the most popular are roulette wheel selection and tournament selection. In roulette wheel selection, we select a parent based on spinning a “roulette wheel.” The size of the slot on the wheel is proportional to the fitness. In tournament selection, a subset of the population is randomly selected and then the best design from the subset (the tournament) is taken to be the parent. We will illustrate both of these.
6. Perform “crossover.” This requires that we select a crossover site and then “swap” strings at the crossover point, creating two new children.
7. Perform “mutation.” The check for mutation is done on a bit by bit basis. The mutation rate is usually kept low (0.005, for example). If a bit mutates, change it from 1 to 0 or vice-versa.
8. The new population has now been created. Decode the population strings to get the normal variable values and evaluate the fitness of each design. We now have a new generation of designs, and so we go back to step 2.
9. Continue for a specific number of generations or until the average change in the fitness value falls below a specified tolerance.

The above steps can be modified by several changes to enhance performance. We will discuss these more in Chapter 5.

There are several parts to the above steps which must be explained further.

7.3. Binary Coded Chromosomes

7.3.1. Precision with Binary Strings

The original GA algorithm worked with binary coded strings. Since our design variables are usually real and continuous, we will need to convert them to binary. This requires that we establish an acceptable level of precision. This is determined from,

$$\text{Precision} = \frac{(U_i - L_i)}{2^p - 1} \quad (4.8)$$

where U_i = upper bound for i th variable
 L_i = lower bound for i th variable
 p = length of binary string

The precision determines the smallest change we can make in a variable and have it reflected in the binary string.

7.3.2. Example: Determining the Precision of Binary Coding

We decide to have a binary string length of 8, and a variable has an upper bound of 10 and a lower bound of zero. The precision is,

$$\frac{(10-0)}{2^8-1} = \frac{10}{255} = 0.0392$$

This is the smallest change in a variable we will be able to distinguish using a binary coding with a string length of 8.

7.3.3. Converting from Real to Binary

To convert a real number value to a binary string, first convert it to a base 10 integer value, using the formula,

$$x_{\text{int}10} = \frac{(x_{\text{real}} - L) * J}{(U - L)} \quad (4.9)$$

$$\begin{aligned} \text{where } x_{\text{real}} &= \text{real number value} \\ x_{\text{int}10} &= \text{base 10 integer} \\ J &= 2^p - 1 \end{aligned}$$

Then convert the integer to a binary string using $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, etc.

7.3.4. Example: Converting from Real to Binary

We have a variable value of 3.567, with a string of length 8, and an upper bound of 10 and a lower bound of zero. The base 10 integer value is,

$$x_{\text{int}10} = \frac{(3.567 - 0) 255}{10 - 0} = 90.95 = 91$$

In binary, this value is $01011011 = ((2^0 = 1) + (2^1 = 2) + (2^3 = 8) + (2^4 = 16) + (2^6 = 64)) = 91$

To go from binary back to real, just solve (4.9) for the real value:

$$x_{\text{real}} = x_{\text{int}10} * \frac{(U - L)}{J} + L \quad (4.10)$$

7.3.5. Creating Chromosomes

A chromosome is created by combining the binary strings of all variables together. If we had two variables, which in binary form were 01011011 and 10001101, the chromosome would be:

0101101110001101

7.4. Genetic Operators: Crossover and Mutation

7.4.1. Crossover

Crossover is the exchange of genetic material between two parent chromosomes to form two children. We randomly pick the crossover point and then swap the binary strings which follow after the crossover point. For example if before crossover we have the following parents with the indicated crossover point,

```
Parent 1:  00111 | 010
Parent 2:  11100 | 111
```

Then after crossover we have:

```
Child 1:  00111111
Child 2:  11100010
```

7.4.2. Mutation

It is sometimes beneficial to introduce random “mutations” into our design. This is usually done bit by bit. If we have a mutation rate of 0.001 (0.1%), we draw random numbers from a uniform distribution. If the number is less than 0.001, we switch the bit from 0 to 1 or vice versa. Mutation helps introduce unexpected changes into the population. However, the mutation rate cannot be too high, or a high fraction of designs are essentially changed randomly. However, some practitioners have used mutation rates as high as 10%.

7.5. Example: Converting a Chromosome Back to Variables

Suppose the chromosome for two variables (x_1, x_2), each of string length 10, is given by:

```
00101101111011011100
```

We partition the chromosome into:

```
x1: 0010110111          x2: 1011011100
```

Minimum and maximum values:

```
5 ≤ x1 ≤ 10          1 ≤ x2 ≤ 25
```

Base10 integer values:

```
x1,int10 = 183          x2,int10 = 732
```

Continuous real values:

```
x1,real = 5.894          x2,real = 18.17
```

7.6. Example: Classical Genetic Algorithm for One Generation

In this problem we have the objective function $f = x_1^1 + x_2^2$ which we wish to maximize. The variables range from -2 to $+5$. We will have a string length of 7. We assume we have available a random number generator that generates uniformly distributed numbers between

0 and 1. We will have a population size of 6 (which is small, but is used here for illustration purposes), a crossover probability of 0.8 and a mutation probability of 0.001. We will use roulette wheel selection to choose parents.

We randomly generate the following six designs as our starting population:

Design	x_1	x_2	Fitness
1	1.521	-0.824	2.9924
2	3.922	-1.006	16.394
3	2.179	-0.033	4.7491
4	-0.292	4.405	19.489
5	-0.523	-1.636	2.95
6	2.956	-1.951	12.544

We then convert the designs to binary strings and form chromosomes:

Design	x_1	Base 10 Int	Binary	x_2	Base 10 Int	Binary	Chromosome
1	1.521	64	1000000	-0.824	21	0010101	10000000010101
2	3.922	107	1101011	-1.006	18	0010010	11010110010010
3	2.179	76	1001100	-0.033	36	0100100	10011000100100
4	-0.292	31	0011111	4.405	116	1110100	00111111110100
5	-0.523	27	0011011	-1.636	7	0000111	00110110000011
6	2.956	90	1011010	-1.951	1	0000001	101101000000001

There is some additional information we need to compute for the roulette wheel selection:

Design	Fitness	Fitness/Sum	Cumulative Probability
1	2.992	0.0506	0.0506
2	16.39	0.2773	0.328
3	4.749	0.0803	0.408
4	19.49	0.3297	0.738
5	2.950	0.0499	0.788
6	12.54	0.2122	1.00
Sum	59.12	1.00	
Average	9.85		

The cumulative probability will be used to set the size of the slots on the roulette wheel. For example, Design 2 has a relatively high fitness of 16.39; this represents 27.7% of the total fitness for all designs. Thus it has a slot which represents 27.7% of the roulette wheel. This slot is the distance from 0.0506 to 0.328 under the cumulative probability column. If a random number falls within this interval, Design 2 is chosen as a parent. In like manner, Design 5, which has a low fitness, only gets 5% of the roulette wheel, represented by the interval from 0.738 to 0.788.

We draw out six random numbers:
0.219, 0.480, 0.902, 0.764, 0.540, 0.297

The first random number is in the interval of Design 2—so Design 2 is chosen as a parent. The second number is within the interval of Design 4—so Design 4 is chosen as a parent. Proceeding in this fashion we find the parents chosen to mate are 2,4,6,5,4,2.

We will mate the parents in the order they were selected. Thus we will mate 2 with 4, 6 with 5, and 4 with 2.

Should we perform crossover for the first set of parents? We draw a random number, 0.422, which is less than 0.8 so we do. We determine the crossover point by selecting a random number, multiplying by 13 (the length of the chromosome minus 1) and taking the interval the number lies within for the crossover point (i.e., 0-1 gives crossover at point 1, 10-11 gives crossover at point 11, etc.), since there are 1-13 crossover points in a 14 bit string. Crossover occurs at: $0.659 * 13 = 8.56 = 9\text{th place}$.

Parent 1: 001111111 | 10100
Parent 2: 110101100 | 10010

Child 1: 00111111110010
Child 2: 11010110010100

Do we perform mutation on any of the children? We check random numbers bit by bit--none are less than 0.001.

Do we do crossover for the second set of parents? We draw a random number of 0.749, less than 0.8, so we do. Crossover for second mating pair: $0.067 * 13 = 0.871 = 1\text{st place}$

Parent 3: 1 | 0110100000001
Parent 4: 0 | 0110110000111

Child 3: 10110110000111
Child 4: 00110100000001

Again, no mutation is performed.

Do we do crossover for third set of parents? Random number = $0.352 \leq 0.8$, so we do. Crossover for third mating pair: $0.260 * 13 = 3.38 = 4\text{th place}$

Parent 5: 1101 | 0110010010
Parent 6: 0111 | 1111110100

Child 5: 11011111110100
Child 6: 00110110010010

As we check mutation, we draw a random number less than 0.001 for the last bit of Child 5. We switch this bit. Thus this child becomes,

Child 5: 11011111110101

We now have a new generation.

We decode the binary strings to Base 10 integers which are converted to real values, using (4.10). Information on this new generation is given below.

Design	Chromosome	Binary x_1	Base 10 Int	x_1	Binary x_2	Base 10 Int	x_2	Fitness
1	00111111110010	0011111	31	-0.291	1110010	114	4.283	18.43
2	11010110010100	1101011	107	3.898	0010100	20	-0.898	16.00
3	10110110000111	1011011	91	3.016	0000111	7	-1.614	11.7
4	00110100000001	0011010	26	-0.567	0000001	1	-1.945	4.10
5	11011111110101	1101111	111	4.118	1110101	117	4.394	36.26
6	00110110010010	0011011	27	-0.5118	0010010	18	-1.008	1.278
							Sum	87.78
							Average	14.63

We see that the average fitness has increased from 9.85 to 14.63.

This completes the process for one generation. We continue the process for as many generations as we desire.

7.7. Example: Genetic Algorithm with Tournament Selection

The previous example used roulette wheel selection. The roulette wheel selection process is dependent upon the scaling we choose for the objective. It must also be modified if we wish to minimize instead of maximize.

Another way to select parents which does not have these drawbacks is *tournament selection*. This involves randomly selecting a subset of the population and then taking the best design of the subset. For example, with a tournament size of two, two designs are randomly chosen and the best of the two is selected as a parent.

Tournament selection can be partly understood by considering the extremes. With a tournament size of one you would have random selection of parents, and with a tournament size equal to the population size, you would always be picking the best design as the parent.

We will have a tournament size of two. We generate two random numbers and then multiply them by the population size:

$$0.219 * 6 = 1.31. \text{ Values between 1 and 2 give Design 2}$$

$0.812 * 6 = 4.87$. Values between 4 and 5 give Design 5

The best design of these two is Design 2, so design 2 is chosen to be Parent 1. (We are still working with the starting generation, not the second generation given above.) We then conduct another tournament to find Parent 2. We continue in a similar fashion until six parents are chosen.

8. Comparison of Algorithms

Some time ago I came across this comparison of gradient-based algorithms, simulated annealing and genetic algorithms. I regret I cannot give credit to the author. The author assumes we are trying to find the top of a hill using kangaroo(s).

“Notice that in all [hill climbing, i.e., gradient-based] methods discussed so far, the kangaroo can hope at best to find the top of a mountain close to where he starts. There’s no guarantee that this mountain will be Everest, or even a very high mountain. Various methods are used to try to find the actual global optimum.

In simulated annealing, the kangaroo is drunk and hops around randomly for a long time. However, he gradually sobers up and tends to hop up hill.

In genetic algorithms, there are lots of kangaroos that are parachuted into the Himalayas (if the pilot didn’t get lost) at random places. These kangaroos do not know that they are supposed to be looking for the top of Mt. Everest. However, every few years, you shoot the kangaroos at low altitudes and hope the ones that are left will be fruitful and multiply.”

9. References

Aarts E. and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, Wiley, 1989.

Balling, R. J., S. May, “Large Scale Discrete Structural Optimization: Simulated Annealing, Branch and Bound, and Other Techniques,” Dept. of Civil Engineering, BYU, Provo, UT 84602.

Bohachevsky, I., M. Johnson, and M. Stein, “Generalized Simulated Annealing for Function Optimization,” *Technometrics*, vol. 28, no. 3, August 1986, p.209;

Borup L., *Optimization Algorithms for Engineering Models Combining Heuristic and Analytic Knowledge*, M.S. Thesis, BYU, 1991

Davis L., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991

Goldberg D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison Wesley, 1989.

Gen, M. and R. Cheng, *Genetic Algorithms and Engineering Optimization*, Wiley Series in Engineering Design and Automation, 2000.

Holland, J., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975, MIT Press, 1992.

Kirkpatrick, S., C.D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, May 1983, p. 671.

Liu, Lit, *Discrete Optimization of Pipe Networks Systems by Simulated Annealing*, Masters Thesis, BYU, December 1990.

Michalewicz Z., *Genetic Algorithms plus Data Structures = Evolution Programs*, Third Edition, Springer, 1999.

Press, Flannery, Teukolsky, Tetterling, *Numerical Recipes*, Cambridge Press, Chapter 10, 1990

CHAPTER 5

GENETIC AND EVOLUTIONARY OPTIMIZATION

1. Introduction

Gradient-based algorithms have some weaknesses relative to engineering optimization. Specifically, it is difficult to use gradient-based algorithms for optimization problems with:

- 1) discrete-valued design variables
- 2) large number of design variables
- 3) multiple local minima, maxima, and saddle points
- 4) nondifferentiable objectives and constraints
- 5) analysis programs which crash for some designs

In recent years, a new family of optimization algorithms has emerged for dealing with the above characteristics. These algorithms are known as evolutionary algorithms. Evolutionary algorithms mimic the optimization process in nature as it optimizes biological species in order to maximize survival of the fittest. One type of evolutionary algorithm is the genetic algorithm. We will examine genetic algorithms in detail.

I express my appreciation to Professor Richard J. Balling of the Civil and Environmental Engineering Department at BYU for allowing me to use this chapter.

2. Genetic Algorithms: Representation

2.1. Chromosomes and Genes

In order to apply a genetic algorithm to a particular optimization problem, one must first devise a representation. A representation involves representing candidate designs as *chromosomes*. The simplest representation is a *value representation* where the chromosome consists of the values of the design variables placed side by side. For example, suppose we have 6 discrete design variables whose values are integer values ranging from 1 to 5 corresponding to 5 different cross-sectional shapes for each of 6 members. Suppose we also have 4 continuous design variables whose values are real numbers ranging from 3.000 to 9.000 representing vertical coordinates of each of 4 joints. A possible chromosome is shown in Fig. 5.1:

4	3	1	3	2	5	3.572	6.594	5.893	8.157
---	---	---	---	---	---	-------	-------	-------	-------

Fig. 5.1: Chromosome for a Candidate Design

The chromosome in Fig. 5.1 consists of ten *genes*, one for each design variable. The value of each gene is the value of the corresponding design variable. Thus, a chromosome represents a particular design since values are specified for each of the design variables.

Another possible representation is the *binary representation*. In this representation, multiple genes may be used to represent each design variable. The value of each gene is either zero or

one. Consider the case of a discrete design variable whose value is an integer ranging from 1 to 5. We would need three binary genes to represent this design variable, and we would have to set up a correspondence between the gene values and the discrete values of the design variable such as the following:

gene values	design variable value
0 0 0	1
0 0 1	2
0 1 0	3
0 1 1	4
1 0 0	5
1 0 1	1
1 1 0	2
1 1 1	3

In this case, note that there is bias in the representation since the discrete values 1, 2, and 3 occur twice as often as the discrete values 4 and 5.

Consider the case of a continuous design variable whose value is a real number ranging from 3.000 to 9.000. The number of genes used to represent this design variable in a binary representation will dictate the precision of the representation. For example, if three genes are used, we may get the following correspondence between the gene values and equally-spaced continuous values of the design variable:

gene values	design variable value
0 0 0	3.000
0 0 1	3.857
0 1 0	4.714
0 1 1	5.571
1 0 0	6.429
1 0 1	7.286
1 1 0	8.143
1 1 1	9.000

Note that the precision of this representation is $\frac{9.000 - 3.000}{2^3 - 1} = 0.857$.

Historically, the binary representation was used in the first genetic algorithms rather than the value representation. However, the value representation avoids the problems of bias for discrete design variables and limited precision for continuous design variables. It is also easy to implement since it is not necessary to make conversions between gene values and design variable values.

2.2. Generations

Genetic algorithms work with generations of designs. The designer specifies the generation size N , which is the number of designs in each generation. The genetic algorithm begins with a starting generation of randomly generated designs. This is accomplished by randomly generating the values of the genes of the N chromosomes in the starting generation. From the starting generation, the genetic algorithm creates the second generation, and then the third generation, and so forth until a specified M = number of generations has been created.

3. Fitness

The genetic algorithm requires that a fitness function be evaluated for every chromosome in the current generation. The fitness is a single number indicating the quality of the design represented by the chromosome. To evaluate the fitness, each design must be analyzed to evaluate the objective f (minimized) and constraints $g_i \leq 0$ ($i = 1$ to m). If there are no constraints, the fitness is simply the value of the objective f . When constraints exist, the objective and constraint values must be combined into a single fitness value. We begin by defining the feasibility of a design:

$$g = \max(0, g_1, g_2, \dots, g_m) \quad (5.1)$$

Note that the design is infeasible if $g > 0$ and feasible if $g = 0$. We assume that in (5.1) the constraints are properly scaled.

One possible definition of fitness involves a user-specified positive penalty parameter P :

$$\text{fitness} = f + P * g \quad (5.2)$$

The fitness given by (5.2) is minimized rather than maximized as in biological evolution. If the penalty parameter P in (5.2) is relatively small, then some infeasible designs will be more fit than some feasible designs. This will not be the case if P is a large value.

An alternative to the penalty approach to fitness is the segregation approach. This approach does not require a user-specified parameter:

$$\text{fitness} = \begin{cases} f & \text{if } g = 0 \\ f_{\max}^{\text{feas}} + g & \text{if } g > 0 \end{cases} \quad (5.3)$$

In (5.3), f_{\max}^{feas} is the maximum value of f for all feasible designs in the generation (designs with $g = 0$). The fitness given by (5.3) is minimized. The segregation approach guarantees that the fitness of feasible designs in the generation is always better (lower) than the fitness of infeasible designs.

3.1. Example 1

The three-bar truss in Fig. 5.2 has two design variables: x_1 = cross-sectional area of members AB and AD, and x_2 = cross-sectional area of member AC. Suppose each design variable ranges from 0.0 to 0.5 and is represented by a single continuous value gene. The starting generation consists of the following six chromosomes.

- | | | |
|-------------------|-------------------|-------------------|
| 1) 0.2833, 0.1408 | 2) 0.0248, 0.0316 | 3) 0.1384, 0.4092 |
| 4) 0.3229, 0.1386 | 5) 0.0481, 0.1625 | 6) 0.4921, 0.2845 |

The problem is the same as in a previous example where the objective and constraints are:

$$f = (100\text{in})x_1 + (40\text{in})x_2$$

$$g_1 = -x_1 \leq 0$$

$$g_2 = -x_2 \leq 0$$

$$g_3 = 9600\text{kip} - (38400\text{ksi})x_1 - (37500\text{ksi})x_2 \leq 0$$

$$g_4 = 15000\text{kip} - (76800\text{ksi})x_1 - (75000\text{ksi})x_2 \leq 0$$

Scale the objective and constraints by their respective values at $x_1 = x_2 = 0.5\text{in}^2$. Then evaluate the segregation fitness of the starting generation. Calculate the average and best fitness for the generation.

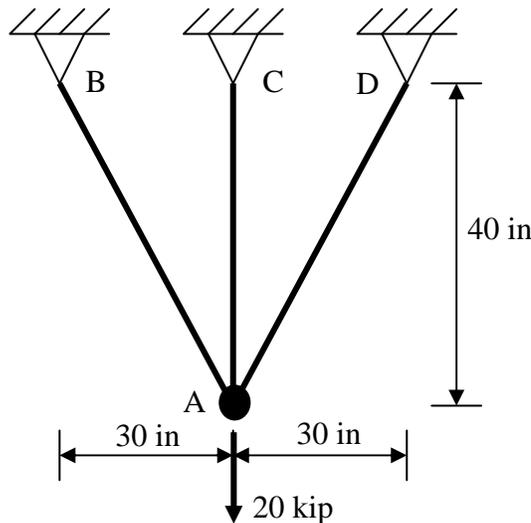


Fig. 5.2 The three-bar truss

Solution

Evaluating the objective and constraints at $x_1 = x_2 = 0.5\text{in}^2$ gives:

$$f = 70\text{in}^3 \quad g_1 = 0.5\text{in}^2 \quad g_2 = 0.5\text{in}^2 \quad g_3 = 28350\text{kip} \quad g_4 = 60900\text{kip}$$

The scaled objective and constraints are:

$$f = \frac{(100\text{in})x_1 + (40\text{in})x_2}{70\text{in}^3} = (1.429\text{in}^{-2})x_1 + (0.571\text{in}^{-2})x_2$$

$$g_1 = \frac{-x_1}{0.5\text{in}^2} = -(2\text{in}^{-2})x_1 \leq 0$$

$$g_2 = \frac{-x_2}{0.5\text{in}^2} = -(2\text{in}^{-2})x_2 \leq 0$$

$$g_3 = \frac{9600\text{kip} - (38400\text{ksi})x_1 - (37500\text{ksi})x_2}{28350\text{kip}}$$

$$= 0.3386 - (1.354\text{in}^{-2})x_1 - (1.323\text{in}^{-2})x_2 \leq 0$$

$$g_4 = \frac{15000\text{kip} - (76800\text{ksi})x_1 - (75000\text{ksi})x_2}{60900\text{kip}}$$

$$= 0.2463 - (1.261\text{in}^{-2})x_1 - (1.232\text{in}^{-2})x_2 \leq 0$$

design 1: $x_1 = 0.2833\text{in}^2$ $x_2 = 0.1408\text{in}^2$
 $f = 0.4852$ $g_1 = -0.5666$ $g_2 = -0.2816$ $g_3 = -0.2313$ $g_4 = -0.2844$
 $g = 0$ $\text{fitness} = 0.4852$

design 2: $x_1 = 0.0248\text{in}^2$ $x_2 = 0.0316\text{in}^2$
 $f = 0.0535$ $g_1 = -0.0496$ $g_2 = -0.0632$ $g_3 = 0.2632$ $g_4 = 0.1761$
 $g = 0.2632$

design 3: $x_1 = 0.1384\text{in}^2$ $x_2 = 0.4092\text{in}^2$
 $f = 0.4314$ $g_1 = -0.2768$ $g_2 = -0.8184$ $g_3 = -0.3902$ $g_4 = -0.4324$
 $g = 0$ $\text{fitness} = 0.4314$

design 4: $x_1 = 0.3229\text{in}^2$ $x_2 = 0.1386\text{in}^2$
 $f = 0.5406$ $g_1 = -0.6458$ $g_2 = -0.2772$ $g_3 = -0.2820$ $g_4 = -0.3316$
 $g = 0$ $\text{fitness} = 0.5406$

design 5: $x_1 = 0.0481\text{in}^2$ $x_2 = 0.1625\text{in}^2$
 $f = 0.1615$ $g_1 = -0.0962$ $g_2 = -0.3250$ $g_3 = 0.0585$ $g_4 = -0.0146$
 $g = 0.0585$

design 6: $x_1 = 0.4921\text{in}^2$ $x_2 = 0.2845\text{in}^2$
 $f = 0.8657$ $g_1 = -0.9842$ $g_2 = -0.5690$ $g_3 = -0.7041$ $g_4 = -0.7247$
 $g = 0$ $\text{fitness} = 0.8657$

$$f_{\max}^{\text{feas}} = 0.8657$$

design 2: fitness = $0.8657 + 0.2632 = 1.1289$

design 5: fitness = $0.8657 + 0.0585 = 0.9242$

average fitness for generation 1 = 0.7293 best fitness for generation 1 = 0.4314

4. Genetic Algorithms: New Generations

The genetic algorithm goes through a four-step process to create a new generation from the current generation:

- 1) selection
- 2) crossover
- 3) mutation
- 4) elitism

4.1. Selection

In this step, we select two designs from the current generation to be the mother design and the father design. Selection is based on fitness. The probability of being selected as mother or father should be greatest for those designs with the best fitness. We will mention two popular selection processes. The first selection process is known as *tournament selection*. With tournament selection, the user specifies a tournament size. Suppose our tournament size is three. We randomly select three designs from the current generation, and the most fit of the three becomes the mother design. Then we randomly select three more designs from the current generation, and the most fit of the three becomes the father design. One may vary the *fitness pressure* by changing the tournament size. The greater the tournament size, the greater the fitness pressure. In the extreme case where the tournament size is equal to the generation size, the most fit design in the current generation would always be selected as both the mother and father. At the other extreme where the tournament size is one, fitness is completely ignored in the random selection of the mother and father.

The second selection process is known as *roulette-wheel selection*. In roulette-wheel selection, the continuous interval from zero to one is divided into subintervals, one for each design in the current generation. If we assume fitness is positive and minimized, then the lengths of the subintervals are proportional to $(1/\text{fitness})^\gamma$. Thus, the longest subintervals correspond to the most fit designs in the current generation. The greater the roulette exponent, γ , the greater the fitness pressure in roulette-wheel selection. A random number between zero and one is generated, and the design corresponding to the subinterval containing the random number becomes the mother design. Another random number between zero and one is generated, and the design corresponding to the subinterval containing the random number becomes the father design.

4.2. Crossover

After selecting the mother and father designs from the current generation, two children designs are created for the next generation by the crossover process. First, we must determine whether or not crossover should occur. A crossover probability is specified by the user. A random number between zero and one is generated, and if it is less than the

crossover probability, crossover is performed. Otherwise, the mother and father designs, without modification, become the two children designs. There are several different ways to perform crossover. One of the earliest crossover methods developed for genetic algorithms is *single-point crossover*. Fig. 5.3 shows the chromosomes for a mother design and a father design. Each chromosome has $n = 10$ binary genes:

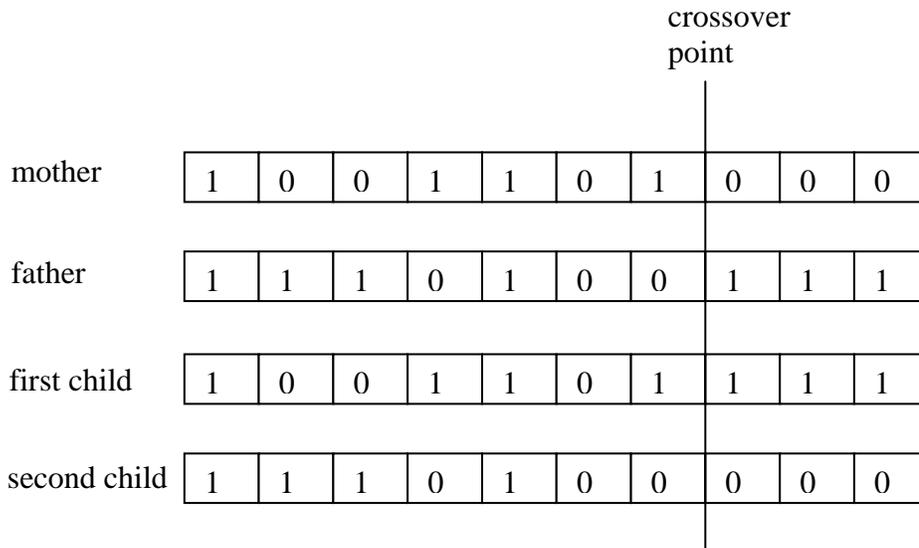


Fig. 5.3: Single-Point Crossover

With single-point crossover, we randomly generate an integer i from 1 to n known as the crossover point, where n is the number of genes in the chromosome. We then cut the mother and father chromosomes after gene i , and swap the tail portions of the chromosomes. In Fig. 5.3, $i = 7$. The first child is identical to the mother before the crossover point, and identical to the father after the crossover point. The second child is identical to the father before the crossover point, and identical to the mother after the crossover point.

Another crossover method is *uniform crossover*. With uniform crossover, a random number r between zero and one is generated for each of the n genes. For a particular gene, if x_1 is the value from the mother design and x_2 is the value from the father design, then the values y_1 and y_2 for the children designs are:

$$\begin{aligned}
 \text{if } r \leq 0.5 & \quad y_1 = x_2 & \quad y_2 = x_1 \\
 \text{if } r > 0.5 & \quad y_1 = x_1 & \quad y_2 = x_2
 \end{aligned}
 \tag{5.4}$$

The goal of crossover is to generate two new children designs that *inherit* many of the characteristics of the fit parent designs. However, this goal may not be achieved when the binary representation is used. Suppose the last four genes in the chromosomes in Fig. 5.3 represent a single discrete design variable whose value is equal to the base ten value of the last four genes. For the mother design, binary values of 1000 give a design variable value of 8, and for the father design, binary values of 0111 give a design variable value of 7. For the first child design, binary values of 1111 give a design variable value of 15, and for the

second child, binary values of 0000 give a design variable value of 0. Thus, the parents have close values of 8 and 7, while the children have values that are very different from the parents of 15 and 0. This is known as the *Hamming cliff* problem of the binary representation. With a value representation, a single gene would have been used for this design variable, and the parent values of 7 and 8 would have been inherited by the children with either single point or uniform crossover.

Blend crossover is similar to uniform crossover since it is also performed gene by gene. Blend crossover makes it possible for children designs to receive random values anywhere in the interval between the mother value and the father value. Thus, we generate a random number between zero and one for a particular gene. If x_1 is the mother value and x_2 is the father value, then the children values y_1 and y_2 are:

$$\begin{aligned} y_1 &= (r)x_1 + (1-r)x_2 \\ y_2 &= (1-r)x_1 + (r)x_2 \end{aligned} \tag{5.5}$$

It is possible to transition between uniform and blend crossover with a user-specified crossover parameter η :

$$\begin{aligned} y_1 &= (a)x_1 + (1-a)x_2 \\ y_2 &= (1-a)x_1 + (a)x_2 \end{aligned} \tag{5.6}$$

where:

$$\begin{aligned} \text{if } r \leq 0.5 & \quad a = \frac{(2r)^{1/\eta}}{2} \\ \text{if } r > 0.5 & \quad a = 1 - \frac{(2-2r)^{1/\eta}}{2} \end{aligned} \tag{5.7}$$

Note that if $\eta = 1$, then $a = r$ and (5.6) becomes (5.5), giving blend crossover. If $\eta = 0$, then in the limit $a = 0$ for $r \leq 0.5$ and $a = 1$ for $r > 0.5$, and (5.6) becomes (5.4), giving uniform crossover. In the limit as η goes to ∞ , a goes to 0.5 and (5.7) becomes $y_1 = y_2 = (x_1+x_2)/2$, which we may call *average crossover*.

4.3. Mutation

The next step for creating the new generation is mutation. A mutation probability is specified by the user. The mutation probability is generally much lower than the crossover probability. The mutation process is performed for each gene of the first child design and for each gene of the second child design. The mutation process is very simple. One generates a random number between zero and one. If the random number is less than the mutation probability, the gene is randomly changed to another value. Otherwise, the gene is left alone. Since the mutation probability is low, the majority of genes are left alone. Mutation makes it possible to occasionally introduce *diversity* into the population of designs.

If all possible values are equally probable for the mutated gene, the mutation is said to be *uniform mutation*. It may be desirable to start out with uniform mutation in the starting generation, but as one approaches the later generations one may wish to favor values near the current value of the gene. We will refer to such mutation as *dynamic mutation*. Let x be the current value of the gene. Let r be a random number between x_{\min} and x_{\max} , which are the minimum and maximum possible values of x , respectively. Let the current generation number be j , and let M be the total number of generations. The parameter β is a user-supplied mutation parameter. The new value of the gene is:

$$\begin{aligned} \text{if } r \leq x & \quad y = x_{\min} + (r - x_{\min})^\alpha (x - x_{\min})^{1-\alpha} \\ \text{if } r > x & \quad y = x_{\max} - (x_{\max} - r)^\alpha (x_{\max} - x)^{1-\alpha} \end{aligned} \quad (5.8)$$

where

$$\alpha = \left(1 - \frac{j-1}{M}\right)^\beta \quad (5.9)$$

In Fig. 5.4, we plot the value of y as a function of r for various values of the uniformity exponent α . Note that if $\alpha = 1$, then $y = r$, which is uniform mutation. For values of α less than one, the mutated gene value favors values near x . The bias increases as α decreases. In fact if $\alpha = 0$, then $y = x$, which means that the gene is not mutated at all.

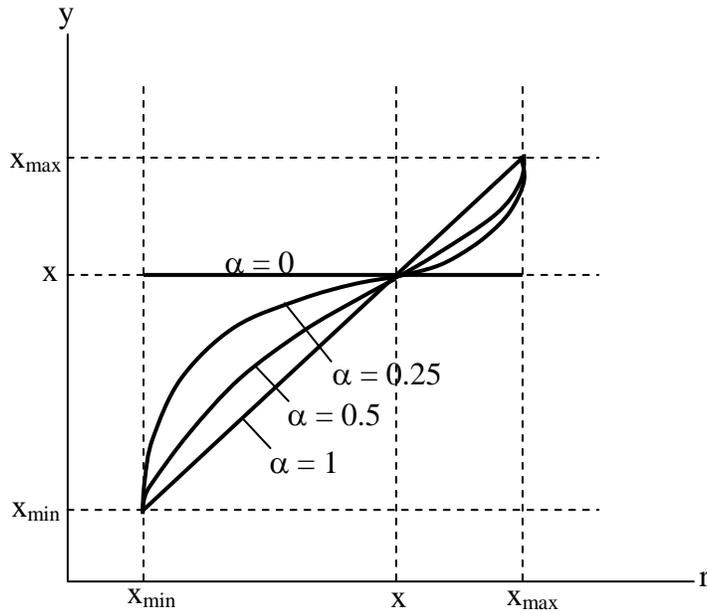


Fig. 5.4: Dynamic Mutation

In Fig. 5.5, we plot the uniformity exponent α versus the mutation parameter β and the generation number j . Note that if $\beta = 0$, then $\alpha = 1$, and the mutation is uniform for all generations. If $\beta > 0$, then $\alpha = 1$ for the starting generation ($j = 1$) and decreases to near zero in the final generation, giving dynamic mutation.

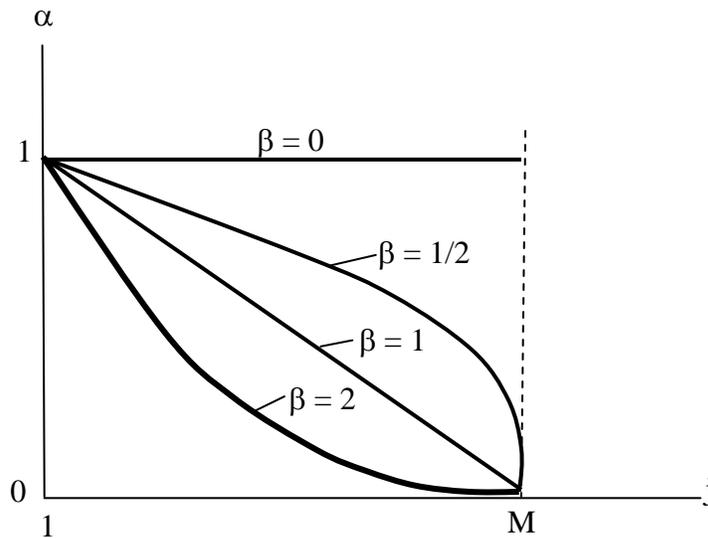


Fig. 5.5: Uniformity Exponent α

4.4. Elitism

The selection, crossover, and mutation processes produce two new children designs for the new generation. These processes are repeated again and again to create more and more

children until the number of designs in the new generation reaches the specified generation size. The final step that must be performed on this new generation is elitism. This step is necessary to guarantee that the best designs survive from generation to generation. One may think of elitism as the rite of passage for children designs to qualify as future parents. The new generation is combined with the previous generation to produce a combined generation of $2N$ designs, where N is the generation size. The combined generation is sorted by fitness, and the N most fit designs survive as the next parent generation. Thus, children must compete with their parents to survive to the next generation.

4.5. Summary

Note that there are many algorithm parameters in a genetic algorithm including: generation size, number of generations, penalty parameter, tournament size, roulette exponent, crossover probability, crossover parameter, mutation probability, and mutation parameter. Furthermore, there are choices between value and binary representation, penalty and segregation fitness, tournament and roulette-wheel selection, single-point, uniform, and blend crossover, and uniform and dynamic mutation. Thus, there is no single genetic algorithm that is best for all applications. One must tailor the genetic algorithm to a specific application by numerical experimentation.

Genetic algorithms are far superior to random trial-and-error search. This is because they are based on the fundamental ideas of fitness pressure, inheritance, and diversity. Children designs inherit characteristics from the best designs in the preceding generation selected according to fitness pressure. Nevertheless, diversity is maintained via the randomness in the starting generation, and the randomness in the selection, crossover, and mutation processes. Research has shown that genetic algorithms can achieve remarkable results rather quickly for problems with huge combinatorial search spaces.

Unlike gradient-based algorithms, it is not possible to develop conditions of optimality for genetic algorithms. Nevertheless, for many optimization problems, genetic algorithms are the only game in town. They are tailor-made to handle discrete-valued design variables. They also work well on ground-structure problems where constraints are deleted when their associated members are deleted, since constraints need not be differentiable or continuous. Analysis program crashes can be handled in genetic algorithms by assigning poor fitness to the associated designs and continuing onward. Genetic algorithms can find all optima including the global optimum if multiple optima exist. Genetic algorithms are conceptually much simpler than gradient-based algorithms. Their only drawback is that they require many executions of the analysis program. This problem will diminish as computer speeds increase, especially since the analysis program may be executed in parallel for all designs in a particular generation.

4.6. Example 2

Perform selection and crossover on the starting generation from Example 1. Use tournament selection with a tournament size of two, and blend crossover according to (5.5) with a crossover probability of 0.6. Use the following random number sequence:

Chapter 5: Genetic and Evolutionary Optimization

0.5292	0.0436	0.2949	0.0411	0.9116	0.7869
0.3775	0.8691	0.1562	0.5616	0.8135	0.4158
0.7223	0.3062	0.1357	0.5625	0.2974	0.6033

Solution

1 + truncate(0.5292(6)) = design 4 fitness = 0.5406

1 + truncate(0.0436(6)) = design 1 fitness = 0.4852

mother = design 1

1 + truncate(0.2949(6)) = design 2 fitness = 1.1289

1 + truncate(0.0411(6)) = design 1 fitness = 0.4852

father = design 1

since mother and father are the same, no crossover needed

child 1 = child 2 = 0.2833, 0.1408

1 + truncate(0.9116(6)) = design 6 fitness = 0.8657

1 + truncate(0.7869(6)) = design 5 fitness = 0.9242

mother = design 6

1 + truncate(0.3775(6)) = design 3 fitness = 0.4314

1 + truncate(0.8691(6)) = design 6 fitness = 0.8657

father = design 3

0.1562 < 0.6 ==> perform crossover

$$(0.5616)0.4921 + (1 - 0.5616)0.1384 = 0.3370$$

$$(1 - 0.5616)0.4921 + (0.5616)0.1384 = 0.2935$$

$$(0.8135)0.2845 + (1 - 0.8135)0.4092 = 0.3078$$

$$(1 - 0.8135)0.2845 + (0.8135)0.4092 = 0.3859$$

child 3 = 0.3370, 0.3078

child 4 = 0.2935, 0.3859

1 + truncate(0.4158(6)) = design 3 fitness = 0.4314

1 + truncate(0.7223(6)) = design 5 fitness = 0.9242

mother = design 3

1 + truncate(0.3062(6)) = design 2 fitness = 1.1289

1 + truncate(0.1357(6)) = design 1 fitness = 0.4852

father = design 1

0.5625 < 0.6 ==> perform crossover

(0.2974) 0.1384+ (1-0.2974) 0.2833= 0.2402

(1-0.2974) 0.1384+ (0.2974) 0.2833= 0.1815

(0.6033) 0.4092+ (1-0.6033) 0.1408 = 0.3027

(1-0.6033) 0.4092+ (0.6033) 0.1408 = 0.2473

child 5 = 0.2402, 0.3027

child 6 = 0.1815, 0.2473

4.7. Example 3

Perform mutation on the problem in Examples 1 and 2. Use a mutation probability of 10%, and perform dynamic mutation according to (5.8) and (5.9) with a mutation parameter of $\beta = 5$. Assume that this is the second of 10 generations. Use the following random number sequence:

0.2252	0.7413	0.5135	0.8383	0.4788	0.1916
0.4445	0.8220	0.2062	0.0403	0.5252	0.3216
0.8673					

Solution

child 1, gene 1: 0.2252 > 0.1 ==> no mutation

child 1, gene 2: 0.7413 > 0.1 ==> no mutation

child 1 = 0.2833, 0.1408

child 2, gene 1: 0.5135 > 0.1 ==> no mutation

child 2, gene 2: 0.8383 > 0.1 ==> no mutation

child 2 = 0.2833, 0.1408

child 3, gene 1: 0.4788 > 0.1 ==> no mutation

child 3, gene 2: 0.1916 > 0.1 ==> no mutation

child 3 = 0.3370, 0.3078

child 4, gene 1: 0.4445 > 0.1 ==> no mutation

child 4, gene 2: 0.8220 > 0.1 ==> no mutation

child 4 = 0.2935, 0.3859

child 5, gene 1: 0.2062 > 0.1 ==> no mutation

child 5, gene 2: 0.0403 > 0.1 ==> mutate!!!

$$\alpha = \left(1 - \frac{2-1}{10}\right)^5 = 0.5905$$

$x_{\min} = 0.0$ $x_{\max} = 0.5$ $x = 0.3027$ (child 5, gene 2 from Example 2)

Generate random number y between x_{\min} and x_{\max} .

$$y = x_{\min} + 0.5252(x_{\max} - x_{\min}) = 0.2626 < x$$

$$z = x_{\min} + (y - x_{\min})^\alpha (x - x_{\min})^{1-\alpha} = 0.2783$$

child 5 = 0.2402, 0.2783

child 6, gene 1: 0.3216 > 0.1 ==> no mutation

child 6, gene 2: 0.8673 > 0.1 ==> no mutation

child 6 = 0.1815, 0.2473

4.8. Example 4

Determine the segregation fitness for each of the 6 child chromosomes in Example 3. Then perform the elitism step to create the second generation. Calculate the average and best fitness of the second generation and compare to the average and best fitness of the starting generation.

Solution

Recall the formulas for the scaled objective and constraints from Example 3(?):

$$f = (1.429 \text{in}^{-2})x_1 + (0.571 \text{in}^{-2})x_2$$

$$g_1 = -(2 \text{in}^{-2})x_1 \leq 0$$

$$g_2 = -(2 \text{in}^{-2})x_2 \leq 0$$

$$g_3 = 0.3386 - (1.354 \text{in}^{-2})x_1 - (1.323 \text{in}^{-2})x_2 \leq 0$$

$$g_4 = 0.2463 - (1.261 \text{in}^{-2})x_1 - (1.232 \text{in}^{-2})x_2 \leq 0$$

child 1: $x_1 = 0.2833 \text{in}^2$ $x_2 = 0.1408 \text{in}^2$

$$f = 0.4852 \quad g_1 = -0.5666 \quad g_2 = -0.2816 \quad g_3 = -0.2313 \quad g_4 = -0.2844$$

$$g = 0 \quad \text{fitness} = 0.4852$$

child 2: $x_1 = 0.2833 \text{in}^2$ $x_2 = 0.1408 \text{in}^2$

$$f = 0.4852 \quad g_1 = -0.5666 \quad g_2 = -0.2816 \quad g_3 = -0.2313 \quad g_4 = -0.2844$$

$$g = 0 \quad \text{fitness} = 0.4852$$

child 3: $x_1 = 0.3370 \text{in}^2$ $x_2 = 0.3078 \text{in}^2$

$$f = 0.6573 \quad g_1 = -0.6740 \quad g_2 = -0.6156 \quad g_3 = -0.5249 \quad g_4 = -0.5579$$

$$g = 0 \quad \text{fitness} = 0.6573$$

child 4: $x_1 = 0.2935 \text{in}^2$ $x_2 = 0.3859 \text{in}^2$

$$f = 0.6398 \quad g_1 = -0.5870 \quad g_2 = -0.7718 \quad g_3 = -0.5693 \quad g_4 = -0.5992$$

$$g = 0 \quad \text{fitness} = 0.6398$$

child 5: $x_1 = 0.2402 \text{in}^2$ $x_2 = 0.2783 \text{in}^2$

$$f = 0.5022 \quad g_1 = -0.4804 \quad g_2 = -0.5566 \quad g_3 = -0.3548 \quad g_4 = -0.3995$$

$$g = 0 \quad \text{fitness} = 0.5022$$

child 6: $x_1 = 0.1815\text{in}^2 \quad x_2 = 0.2473\text{in}^2$

$$f = 0.4006 \quad g_1 = -0.3630 \quad g_2 = -0.4946 \quad g_3 = -0.2343 \quad g_4 = -0.2872$$

$$g = 0 \quad \text{fitness} = 0.4006$$

Parent generation:

design 1: 0.2833, 0.1408	fitness = 0.4852
design 2: 0.0248, 0.0316	fitness = 1.1289
design 3: 0.1384, 0.4092	fitness = 0.4314
design 4: 0.3229, 0.1386	fitness = 0.5406
design 5: 0.0481, 0.1625	fitness = 0.9242
design 6: 0.4921, 0.2845	fitness = 0.8657

Child generation:

child 1: 0.2833, 0.1408	fitness = 0.4852
child 2: 0.2833, 0.1408	fitness = 0.4852
child 3: 0.3370, 0.3078	fitness = 0.6573
child 4: 0.2935, 0.3859	fitness = 0.6398
child 5: 0.2402, 0.2783	fitness = 0.5022
child 6: 0.1815, 0.2473	fitness = 0.4006

Generation 2:

design 1: 0.1815, 0.2473	fitness = 0.4006
design 2: 0.1384, 0.4092	fitness = 0.4314
design 3: 0.2833, 0.1408	fitness = 0.4852
design 4: 0.2833, 0.1408	fitness = 0.4852
design 5: 0.2833, 0.1408	fitness = 0.4852
design 6: 0.2402, 0.2783	fitness = 0.5022

average fitness for generation 2 = 0.4650 best fitness for generation 2 = 0.4006
Significantly better than starting generation.

5. Multi-Objective Optimization

Many optimization problems possess multiple objective functions. In structural design we may wish to minimize cost, maximize safety, maximize aesthetic beauty, minimize maintenance, maximize usable space, etc. Suppose, for example, we desire to minimize cost and minimize deflection at a particular location. These two objectives are *competing*. This means that the minimum cost design is not likely to be the minimum deflection design. Fig. 5.6 shows an *objective space plot* for a particular structural optimization problem. The shaded region represents the possible combinations of cost and deflection for all feasible

designs. Design A is the minimum cost design and design B is the minimum deflection design. Designs lying on the *Pareto front* are good compromise designs between the two objectives. It is often difficult to numerically quantify the relative preference of cost versus deflection. Many people do not know what their preferences are until they have a chance to inspect a variety of good designs. Without a numerical quantification of preference, it is impossible to combine the two objectives into a single objective and then execute an optimization algorithm. Since genetic algorithms work with generations of designs, they have the ability to produce a variety of designs on the Pareto front in a single run without requiring any numerical quantification of preference. Designers can then inspect these designs, form their opinions, and make a selection.

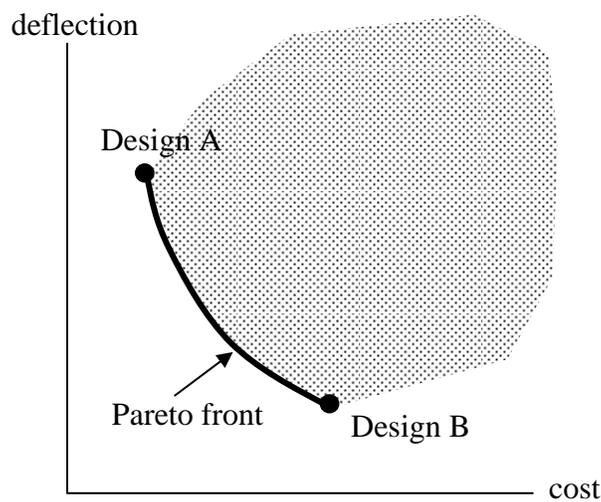


Fig. 5.6 Objective Space Plot

Now let's see how to modify a genetic algorithm to produce a variety of Pareto designs in a single run. First, it is necessary to formally define Pareto design. Pareto designs are the nondominated designs from a given set of designs. Design j dominates design i if it is equal or better in every objective, and better in at least one objective. Consider the generation of ten designs plotted in objective space in Fig. 5.7:

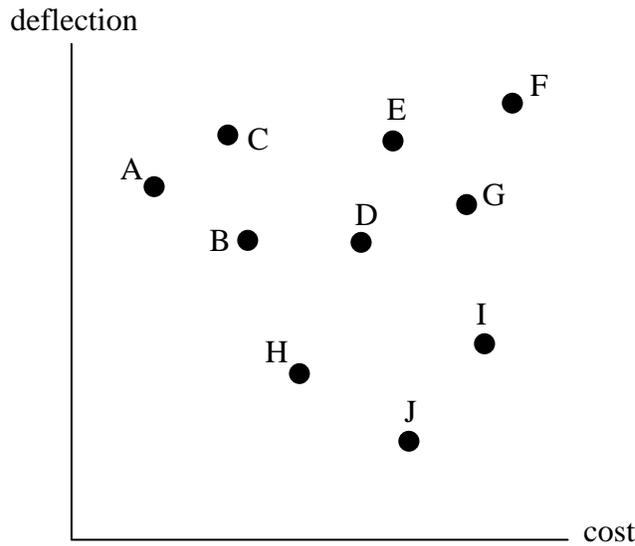


Fig. 5.7: A Generation of Ten Designs

Design H dominates design D because it has lower values in both objectives. Design B dominates design D because it has a lower value of cost and an equal value of deflection. There are no designs that dominate designs A, B, H, or J. These four designs are the Pareto designs for this particular set of ten designs.

It is our goal that the genetic algorithm converges to the Pareto designs for the set of all possible designs. We do this by modifying the fitness function. We will examine three different fitness functions for multi-objective optimization. The first fitness function is called the *scoring fitness function*. For a particular design i in a particular generation, the scoring fitness is equal to one plus the number of designs that dominate design i . We minimize this fitness function. For example, in Fig. 5.7, the scoring fitness of design D is 3 since it is dominated by designs B and H. The scoring fitness of design F is 10 since it is dominated by all other designs. Note that the scoring fitness of the Pareto designs is one since they are nondominated.

Another fitness function for multi-objective optimization is the *ranking fitness function*. This fitness function is also minimized. To begin, the Pareto designs in the generation are identified and assigned a rank of one. Thus, designs A, B, H, and J in Fig. 5.7 are assigned a rank of one. These designs are temporarily deleted, and the Pareto designs of the remaining set are identified and assigned a rank of two. Thus, designs C, D, and I in Fig. 5.7 are assigned a rank of two. These designs are temporarily deleted, and the Pareto designs of the remaining set are identified and assigned a rank of three. Thus, designs E and G in Fig. 5.7 are assigned a rank of three. This procedure continues until all designs in the generation have been assigned a rank. Thus, design F in Fig. 5.7 is assigned a rank of four. The ranking fitness differs from the scoring fitness. In Fig. 5.7, designs C and D have the same rank but they have different scores (design C has a score of 2 and design D has a score of 3).

Note that since all Pareto designs in a particular generation have ranks and scores of one, they are all regarded as equally fit. Thus, there is nothing to prevent clustering on the Pareto front. Indeed, numerical experiments have shown that genetic algorithms with scoring or ranking fitness will often converge to a single design on the Pareto front. We also observe that the scoring and ranking fitness functions are discontinuous functions of the objective values. An infinitesimal change in the value of an objective may cause the ranking or scoring fitness to jump to another integer value.

The third multi-objective fitness function we will consider is the *maximin fitness function*. We derive this fitness function directly from the definition of dominance. Let us assume that the designs in a particular generation are distinct in objective space, and that the n objectives are minimized. Let $f_k^j = \text{value of the } k\text{'th objective for design } i$. Design j dominates design i if:

$$f_k^i \geq f_k^j \quad \text{for } k = 1 \text{ to } n \quad (5-10)$$

Equation (5-10) is equivalent to:

$$\min_k (f_k^i - f_k^j) \geq 0 \quad (5-11)$$

Thus, design i is a dominated design if:

$$\max_{j \neq i} \left(\min_k (f_k^i - f_k^j) \right) \geq 0 \quad (5-12)$$

The maximin fitness of design i is:

$$\max_{j \neq i} \left(\min_k (f_k^i - f_k^j) \right) \geq 0 \quad (5-13)$$

The maximin fitness is minimized. The maximin fitness of Pareto designs will be less than zero, while the maximin fitness of dominated designs will be greater than or equal to zero. The maximin fitness of all Pareto designs is not the same. The more isolated a design is on the Pareto front, the more negative its maximin fitness will be. On the other hand, two designs that are infinitesimally close to each other on the Pareto front will have maximin fitnesses that are negative and near zero. Thus, the maximin fitness function avoids clustering. Furthermore, the maximin fitness is a continuous function of the objective values.

5.1. Example 5

Consider an optimization problem with two design variables, x_1 and x_2 , no constraints, and two objectives:

$$f_1 = 10x_1 - x_2 \quad f_2 = \frac{1 + x_2}{x_1}$$

A particular generation in a genetic algorithm consists of the following six designs:

design 1	$x_1=1$	$x_2=1$	design 4	$x_1=1$	$x_2=0$
design 2	$x_1=1$	$x_2=8$	design 5	$x_1=3$	$x_2=17$
design 3	$x_1=7$	$x_2=55$	design 6	$x_1=2$	$x_2=11$

Calculate the objective values for these designs and make an objective space plot of this generation.

Solution

$$f_1^1 = 10(1) - 1 = 9 \qquad f_2^1 = \frac{1+1}{1} = 2$$

$$f_1^2 = 10(1) - 8 = 2 \qquad f_2^2 = \frac{1+8}{1} = 9$$

$$f_1^3 = 10(7) - 55 = 15 \qquad f_2^3 = \frac{1+55}{7} = 8$$

$$f_1^4 = 10(1) - 0 = 10 \qquad f_2^4 = \frac{1+0}{1} = 1$$

$$f_1^5 = 10(3) - 17 = 13 \qquad f_2^5 = \frac{1+17}{3} = 6$$

$$f_1^6 = 10(2) - 11 = 9 \qquad f_2^6 = \frac{1+11}{2} = 6$$

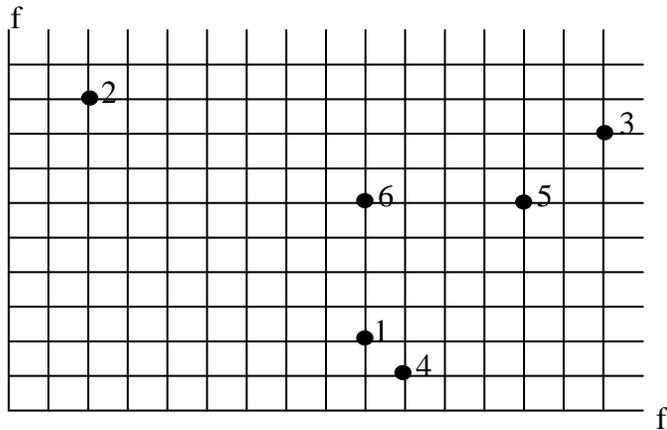


Fig. 5.8

5.2. Example 6

Determine the scoring fitness and ranking fitness for the designs in Example 5.

Solution

Scoring Fitness:

Designs 2, 1, and 4 are not dominated by any other designs. Their score is 1.

Design 6 is dominated by design 1. Its score is 2.

Design 5 is dominated by designs 1, 4, and 6. Its score is 4.

Design 3 is dominated by designs 1, 4, 6, and 5. Its score is 5.

Ranking Fitness:

Designs 2, 1, and 4 have a rank of 1.

Design 6 has a rank of 2.

Design 5 has a rank of 3.

Design 3 has a rank of 4.

5.3. Example 7

Determine the maximin fitness for the designs in Example 5.

Solution

$$\begin{aligned}
 \text{design 1:} \quad & \max \left(\begin{array}{l} \min(9-2, 2-9), \min(9-15, 2-8), \min(9-10, 2-1), \\ \min(9-13, 2-6), \min(9-9, 2-6) \end{array} \right) \\
 & = \max(\min(7, -7), \min(-6, -6), \min(-1, 1), \min(-4, -4), \min(0, -4)) \\
 & = \max(-7, -6, -1, -4, -4) \\
 & = -1
 \end{aligned}$$

$$\begin{aligned}
 \text{design 2:} \quad & \max \left(\begin{array}{l} \min(2-9, 9-2), \min(2-15, 9-8), \min(2-10, 9-1), \\ \min(2-13, 9-6), \min(2-9, 9-6) \end{array} \right) \\
 & = \max(\min(-7, 7), \min(-13, 1), \min(-8, 8), \min(-11, 3), \min(-7, 3))
 \end{aligned}$$

$$= \max(-7, -13, -8, -11, -7)$$

$$= -7$$

$$\text{design 3: } \max\left(\begin{array}{l} \min(15-9, 8-2), \min(15-2, 8-9), \min(15-10, 8-1), \\ \min(15-13, 8-6), \min(15-9, 8-6) \end{array}\right)$$

$$= \max(\min(6,6), \min(13,-1), \min(5,7), \min(2,2), \min(6,2))$$

$$= \max(6, -1, 5, 2, 2)$$

$$= 6$$

$$\text{design 4: } \max\left(\begin{array}{l} \min(10-9, 1-2), \min(10-2, 1-9), \min(10-15, 1-8), \\ \min(10-13, 1-6), \min(10-9, 1-6) \end{array}\right)$$

$$= \max(\min(1,-1), \min(8,-8), \min(-5,-7), \min(-3,-5), \min(1,-5))$$

$$= \max(-1, -8, -7, -5, -5)$$

$$= -1$$

$$\text{design 5: } \max\left(\begin{array}{l} \min(13-9, 6-2), \min(13-2, 6-9), \min(13-15, 6-8), \\ \min(13-10, 6-1), \min(13-9, 6-6) \end{array}\right)$$

$$= \max(\min(4,4), \min(11,-3), \min(-2,-2), \min(3,5), \min(4,0))$$

$$= \max(4, -3, -2, 3, 0)$$

$$= 4$$

$$\text{design 6: } \max\left(\begin{array}{l} \min(9-9, 6-2), \min(9-2, 6-9), \min(9-15, 6-8), \\ \min(9-10, 6-1), \min(9-13, 6-6) \end{array}\right)$$

$$= \max(\min(0,4), \min(7,-3), \min(-6,-2), \min(-1,5), \min(-4,0))$$

$$= \max(0, -3, -6, -1, -4)$$

$$= 0$$

Note that design 2 is more fit than designs 1 and 4 even though all three are Pareto designs (negative fitness). This is because designs 1 and 4 are clustered.

CHAPTER 6

CONSTRAINED OPTIMIZATION 1: K-T CONDITIONS

1 Introduction

We now begin our discussion of gradient-based constrained optimization. Recall that in Chapter 3 we looked at gradient-based unconstrained optimization and learned about the necessary and sufficient conditions for an unconstrained optimum, various search directions, conducting a line search, and quasi-Newton methods. We will build on that foundation as we extend the theory to problems with constraints.

2 Necessary Conditions for Constrained Optimum

At an *unconstrained* local optimum, there is no direction in which we can move to improve the objective function. We can state the necessary conditions mathematically as $\nabla f = 0$. At a *constrained* local optimum, there is no *feasible* direction in which we can move to improve the objective. That is, there may be directions from the current point that will improve the objective, but these directions point into infeasible space.

The necessary conditions for a constrained local optimum are called the *Kuhn-Tucker Conditions*, and these conditions play a very important role in constrained optimization theory and algorithm development.

2.1 Problem Form

It will be convenient to cast our optimization problem into one of two particular forms. This is no restriction since any problem can be cast into either of these forms.

$$\begin{aligned} \text{Max} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \\ & g_i(\mathbf{x}) - b_i \leq 0 \quad i = 1, \dots, k \\ & g_i(\mathbf{x}) - b_i = 0 \quad i = k + 1, \dots, m \end{aligned}$$

or

$$\begin{aligned} \text{Min} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \\ & g_i(\mathbf{x}) - b_i \geq 0 \quad i = 1, \dots, k \\ & g_i(\mathbf{x}) - b_i = 0 \quad i = k + 1, \dots, m \end{aligned}$$

2.2 Graphical Examples

For the graphical examples below, we will assume we are maximizing with \leq constraints.

We have previously considered how we can tell mathematically if some arbitrary vector, \mathbf{s} , points downhill. That condition is, $\mathbf{s}^T \nabla f < 0$. We developed this condition by noting that

any vector \mathbf{s} could be resolved into vector components which lie in the tangent plane and along the gradient (or negative gradient) direction.

Now suppose we have the situation shown in Fig. 6.1 below. We are maximizing. We have contours increasing in the direction of the arrow. The gradient vector is shown. What is the set of directions which improves the objective? It is the set for which $\mathbf{s}^T \nabla f > 0$. We show that set as a semi-circle in Fig. 6.1

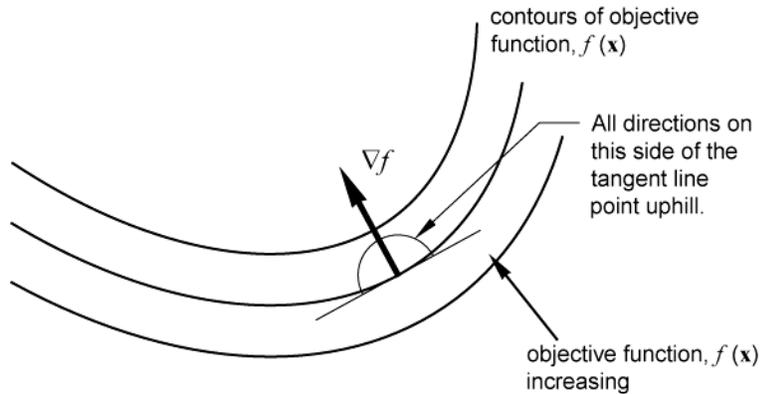


Fig. 6.1 Gradient and set of directions which improves objective function.

Now suppose we add in a less-than inequality constraint, $g(\mathbf{x}) \leq 0$. Contours for this constraint are given in Fig. 6.2. The triangular markers indicate the contour for the allowable value and point towards the direction of the feasible space. What is the set of directions which is feasible? It is the set for which $\mathbf{s}^T \nabla g < 0$. That set is shown as a semi-circle in the figure.

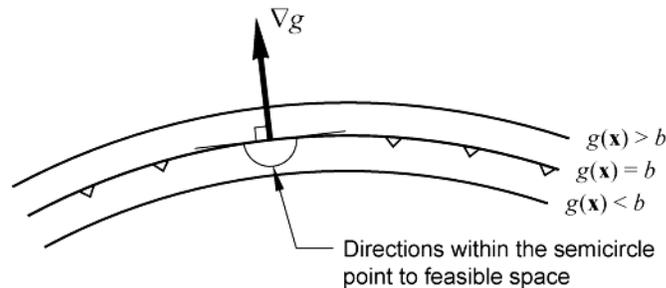


Fig. 6.2 Gradient and set of feasible directions for a constraint.

Now suppose we overlay these two sets of contours on top of each other, as in Fig. 6.3. Where does the optimum lie? By definition, a constrained optimum is a point for which there is no feasible direction which improves the objective. We can see that that condition occurs when the gradient for the objective and gradient for the constraint lie on top of each other. When this happens, the set of directions which improves the objective (dashed semi-circle) does not overlap with the set of feasible directions (solid semi-circle.)

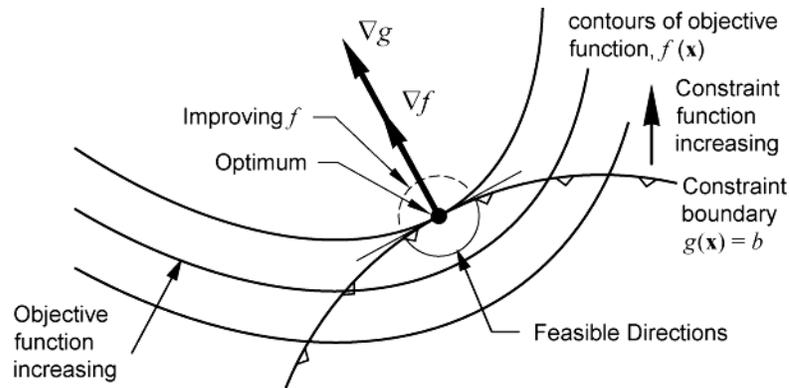


Fig. 6.3 An optimum for one binding constraint occurs when the gradient vectors overlap. When this condition occurs, no feasible point exists which improves the objective.

Mathematically we can write the above condition as

$$\nabla f(\mathbf{x}^*) = \lambda \nabla g_1(\mathbf{x}^*) \quad (6.1)$$

where λ is a positive constant.

Now consider a case where there are two binding constraints at the solution, as shown in Fig. 6.4

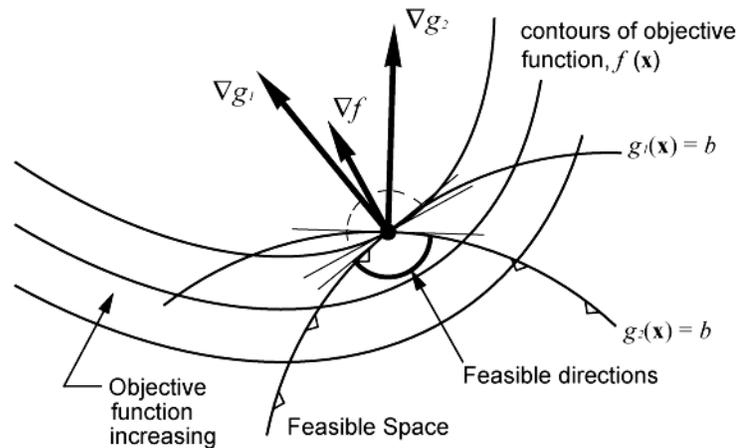


Fig. 6.4 Two binding constraints at an optimum. As long as the objective gradient is within the cone of the constraint gradients, no feasible point exists which improves the objective.

We see that the objective gradient vector is “contained inside” the constraint gradient vectors. If the objective gradient vector is within the constraint gradient vectors, then no

direction exists which simultaneously improves the objective and points in the feasible region. We can state this condition mathematically as:

$$\nabla f(\mathbf{x}^*) = \lambda_1 \nabla g_1(\mathbf{x}^*) + \lambda_2 \nabla g_2(\mathbf{x}^*) \quad (6.2)$$

where, as with the single constraint case, λ_1 and λ_2 are positive constants. Having graphically motivated the development of the main mathematical conditions for a constrained optimum, we are now ready to state these conditions.

2.3 The Kuhn-Tucker Conditions

The Kuhn-Tucker conditions are the necessary conditions for a point to be a constrained local optimum, for either of the general problems given below. (The K-T equations also work for an unconstrained optimum, as we will explain later.)

If \mathbf{x}^* is a local max for:

$$\text{Max } f(\mathbf{x}) \quad (6.3)$$

s.t.:

$$g_i(\mathbf{x}) - b_i \leq 0 \quad i = 1, \dots, k \quad (6.4)$$

$$g_i(\mathbf{x}) - b_i = 0 \quad i = k + 1, \dots, m \quad (6.5)$$

Or if \mathbf{x}^* is a local min for:

$$\text{Min } f(\mathbf{x}) \quad (6.6)$$

s.t.:

$$g_i(\mathbf{x}) - b_i \geq 0 \quad i = 1, \dots, k \quad (6.7)$$

$$g_i(\mathbf{x}) - b_i = 0 \quad i = k + 1, \dots, m \quad (6.8)$$

and if the constraint gradients at the optimum, $\nabla g_i(\mathbf{x}^*)$, are independent, then there exist $(\lambda^*)^T = [\lambda_1 \dots \lambda_m]$, called *Lagrange multipliers*, such that \mathbf{x}^* and λ^* satisfy the following system of equations,

$$g_i(\mathbf{x}^*) - b_i \text{ is feasible } \quad i = 1, \dots, m \quad (6.9)$$

$$\nabla f(\mathbf{x}^*) - \sum_{i=1}^m \lambda_i^* \nabla g_i(\mathbf{x}^*) = \mathbf{0} \quad (6.10)$$

$$\lambda_i^* [g_i(\mathbf{x}^*) - b_i] = 0 \quad i = 1, \dots, k \quad (6.11)$$

$$\lambda_i^* \geq 0 \quad i = 1, \dots, k \quad (6.12)$$

$$\lambda_i^* \text{ unrestricted for } i = k + 1, \dots, m \quad (6.13)$$

or $-\infty < \lambda_i < \infty$

Note in the above equations, $i = 1, \dots, k$ indicates inequality constraints, $i = k + 1, \dots, m$ indicates equality constraints, and $i = 1, \dots, m$ indicates all constraints.

Just as with the necessary conditions for an unconstrained optimum, the K-T conditions are necessary but not sufficient conditions for a constrained optimum.

We will now explain each of these conditions.

Equation (6.9) requires that a constrained optimum be feasible with respect to all constraints.

Equation (6.10) requires the objective function gradient to be a linear combination of the constraint gradients. This insures there is no direction that will simultaneously improve the objective and satisfy the constraints.

Equation (6.11) enforces a condition known as *complementary slackness*. Notice that this condition is for the inequality constraints only. This condition states that either an inequality constraint is binding, or the associated Lagrange multiplier is zero. Essentially this means that nonbinding inequality constraints drop out of the problem.

Equation (6.12) states that the Lagrange multipliers for the inequality constraints must be positive.

Equation (6.13) states that the Lagrange multipliers for the equality constraints can be either positive or negative.

Note that (6.10) above, which is given in vector form, represents a system of n equations. We can rewrite (6.10) as:

$$\begin{array}{ccccccc} \frac{\partial f}{\partial x_1} & -\lambda_1 \frac{\partial g_1}{\partial x_1} & -\lambda_2 \frac{\partial g_2}{\partial x_1} & - \dots - & \lambda_m \frac{\partial g_m}{\partial x_1} & = & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ \frac{\partial f}{\partial x_n} & -\lambda_1 \frac{\partial g_1}{\partial x_n} & -\lambda_2 \frac{\partial g_2}{\partial x_n} & - \dots - & \lambda_m \frac{\partial g_m}{\partial x_n} & = & 0 \end{array} \quad (6.14)$$

We note there is a Lagrange multiplier, λ , for every constraint. Recall, however, that if the constraint is not binding then its Lagrange multiplier is zero, from (6.11).

Taken together, the K-T conditions represent $m+n$ equations in $m+n$ unknowns. The equations are the n equations given by (6.14) (or (6.10)) and the m constraints ((6.9)). The unknowns are the n elements of the vector \mathbf{x} and the m elements of the vector $\boldsymbol{\lambda}$

2.4 Examples of the K-T Conditions

2.4.1 Example 1: An Equality Constrained Problem

Using the K-T equations, find the optimum to the problem,

$$\begin{aligned} \text{Min} \quad & f(\mathbf{x}) = 2x_1^2 + 4x_2^2 \\ \text{s.t.} \quad & g_1: 3x_1 + 2x_2 = 12 \end{aligned}$$

A picture of this problem is given below:

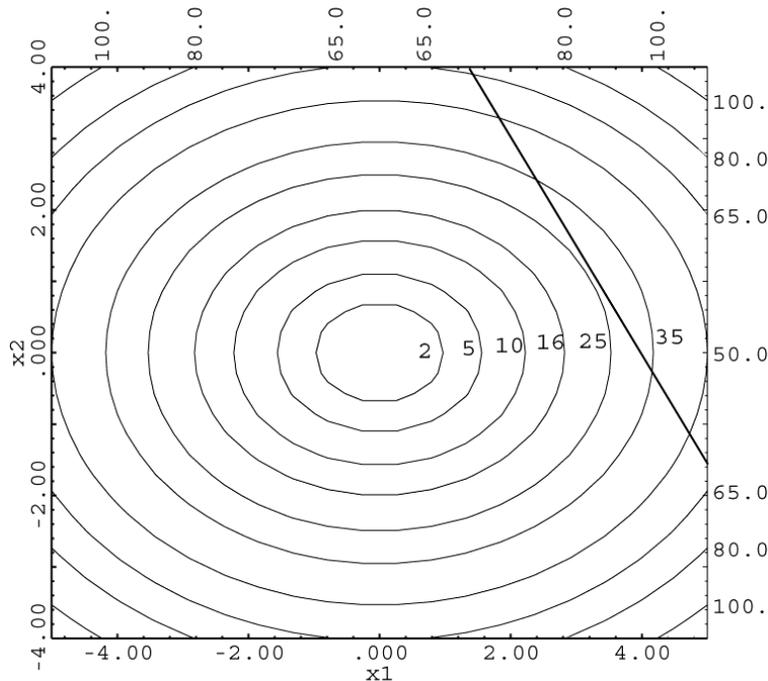


Fig. 6.5 Contours of functions for Example 1.

Since the constraint is an equality constraint, we know it is binding, so the Lagrange multiplier will be non-zero. With two variables and one constraint, the K-T equations represent three equations in three unknowns.

The K-T conditions can be written:

$$\begin{aligned} \frac{\partial f}{\partial x_1} - \lambda \frac{\partial g_1}{\partial x_1} &= 0 \\ \frac{\partial f}{\partial x_2} - \lambda \frac{\partial g_1}{\partial x_2} &= 0 \\ g_1(\mathbf{x}) - b_1 &= 0 \end{aligned}$$

evaluating these expressions:

$$\begin{aligned}4x_1 - \lambda(3) &= 0 \\8x_2 - \lambda(2) &= 0 \\3x_1 + 2x_2 - 12 &= 0\end{aligned}$$

which we can write in matrix form as:

$$\begin{bmatrix} 4 & 0 & -3 \\ 0 & 8 & -2 \\ 3 & 2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 12 \end{bmatrix}$$

The solution to this system of equations is $x_1 = 3.2727$, $x_2 = 1.0909$, $\lambda = 4.3636$. The value of the objective at this point is $f = 26.18$. This optimum occurs where the gradient vectors of the constraint and objective overlap, just as indicated by the graphical discussion. We should verify to make sure this is a constrained min and not a constrained max, since the K-T equations apply at both.

Because the objective function for this problem is quadratic and the constraint is linear, the K-T equations are linear and thus easy to solve. We call a problem with a quadratic objective and linear constraints a *quadratic programming problem* for this reason. Usually the K-T equations are not linear. However, the SQP algorithm attempts to solve these equations by solving a sequence of quadratic program approximations to the real program—thus the name of “Sequential Quadratic Programming.”

2.4.2 Example 2: An Inequality Constrained Problem

In general it is more difficult to use the K-T conditions to solve for the optimum of an inequality constrained problem (than for a problem with equality constraints only) because we don’t know beforehand which constraints are binding at the optimum. Thus we often use the K-T conditions to *verify* that a point we have reached is a candidate optimal solution. Given a point, it is easy to check which constraints are binding.

Verify that the point $\mathbf{x}^T = [0.7059 \quad 2.8235]$ is an optimum to the problem:

$$\begin{aligned}\text{Min} \quad & f(\mathbf{x}) = x_1^2 + x_2^2 \\ \text{s.t.} \quad & g : x_1 + 4x_2 \geq 12\end{aligned}$$

Step 1: Put problem in proper form:

$$\begin{aligned}\text{Min} \quad & f(\mathbf{x}) = x_1^2 + x_2^2 \\ \text{s.t.} \quad & g = x_1 + 4x_2 - 12 \geq 0\end{aligned}$$

Step 2: See which constraints are binding:

$$0.7059 + 4(2.8235) - 12 = -0.0001 \Rightarrow \lambda \neq 0$$

Since this constraint is binding, the associated Lagrange multiplier is solved for. (If it were not binding, the Lagrange multiplier would be zero, from complementary slackness.)

Step 3: Write out the Lagrange multiplier equations represented by (6.10):

$$\begin{aligned} \frac{\partial f}{\partial x_1} - \lambda \frac{\partial g_1}{\partial x_1} &= 2x_1 - \lambda(1) = 0 \\ \frac{\partial f}{\partial x_2} - \lambda \frac{\partial g_1}{\partial x_2} &= 2x_2 - \lambda(4) = 0 \end{aligned}$$

Step 4: Substitute in the given point:

$$2(0.7059) = \lambda \tag{6.15}$$

$$2(2.8235) = 4\lambda \tag{6.16}$$

$$\text{From (6.15), } \lambda = 1.4118$$

$$\text{From (6.16), } \lambda = 1.4118$$

Since these λ 's are consistent and positive, the above point satisfies the K-T equations and is a candidate optimal solution.

2.4.3 Example 3: Another Inequality Constrained Problem

Given the problem:

$$\begin{aligned} \text{Min} \quad & f(\mathbf{x}) = x_1^2 + x_2 \\ \text{s.t.} \quad & g_1(\mathbf{x}) = x_1^2 + x_2^2 - 9 \leq 0 \\ & g_2(x) = x_1 + x_2 - 1 \leq 0 \end{aligned}$$

See if $\mathbf{x}^* = [0 \quad -3]^T$ satisfies the K-T conditions.

Graphically the problem looks like,

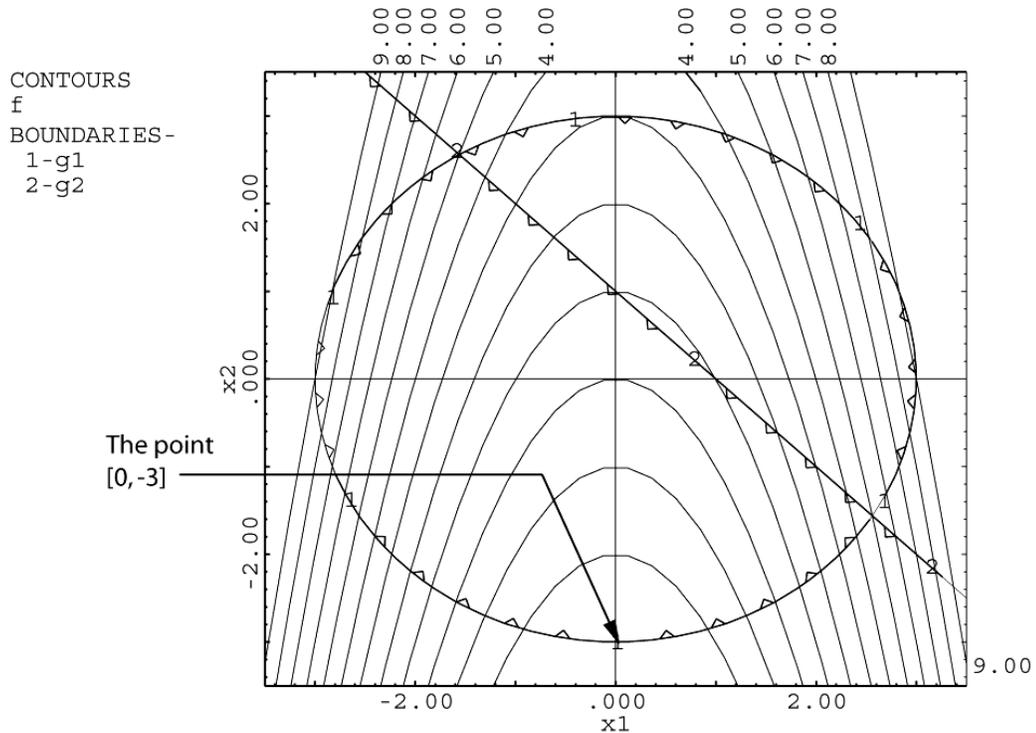


Fig. 6.6. Contour plot and proposed point for Example 3.

At the proposed point constraint g_1 is binding; g_2 is not.

Step 1: Change problem to be in the form of (6.3-6.5):

$$\begin{aligned} \text{Max} \quad & f(\mathbf{x}) = -x_1^2 - x_2 \\ \text{s.t.} \quad & g_1(\mathbf{x}): x_1^2 + x_2^2 - 9 \leq 0 \\ & g_2(\mathbf{x}): x_1 + x_2 - 1 \leq 0 \end{aligned}$$

Step 2: See which constraints are binding:

In this case we can check constraints graphically. Because $g_2(\mathbf{x})$ is not binding, $\lambda_2 = 0$ from (6.11). However, λ_1 is solved for since $g_1(\mathbf{x})$ is binding.

Step 3: Write out the Lagrange multiplier equations represented by (6.10):

$$\frac{\partial f}{\partial x_1} - \lambda_1 \frac{\partial g_1}{\partial x_1} - \lambda_2 \frac{\partial g_2}{\partial x_1} = -2x_1 - \lambda_1(2x_1) = 0 \quad (6.17)$$

$$\frac{\partial f}{\partial x_2} - \lambda_1 \frac{\partial g_1}{\partial x_2} - \lambda_2 \frac{\partial g_2}{\partial x_2} = -1 - \lambda_1(2x_2) = 0 \quad (6.18)$$

Step 4: Substitute in the given point:

At $\mathbf{x}^T = [0 \ -3]$, (6.17) vanishes; from (6.18):

$$-\lambda_1(2)(-3) = 1 \Rightarrow \lambda_1 = \frac{1}{6}$$

so a valid set of λ 's, i.e., $(\lambda^*)^T = \left[\frac{1}{6} \ 0 \right]$ has been found and the K-T conditions are satisfied. This is therefore a candidate optimal solution.

2.4.4 Example 4: Another Point for the Same Problem

Check to see if $(\mathbf{x}^*)^T = [1 \ 0]$ satisfies the K-T conditions for the problem given in Example 3 above. This point is shown in Fig. 6.7

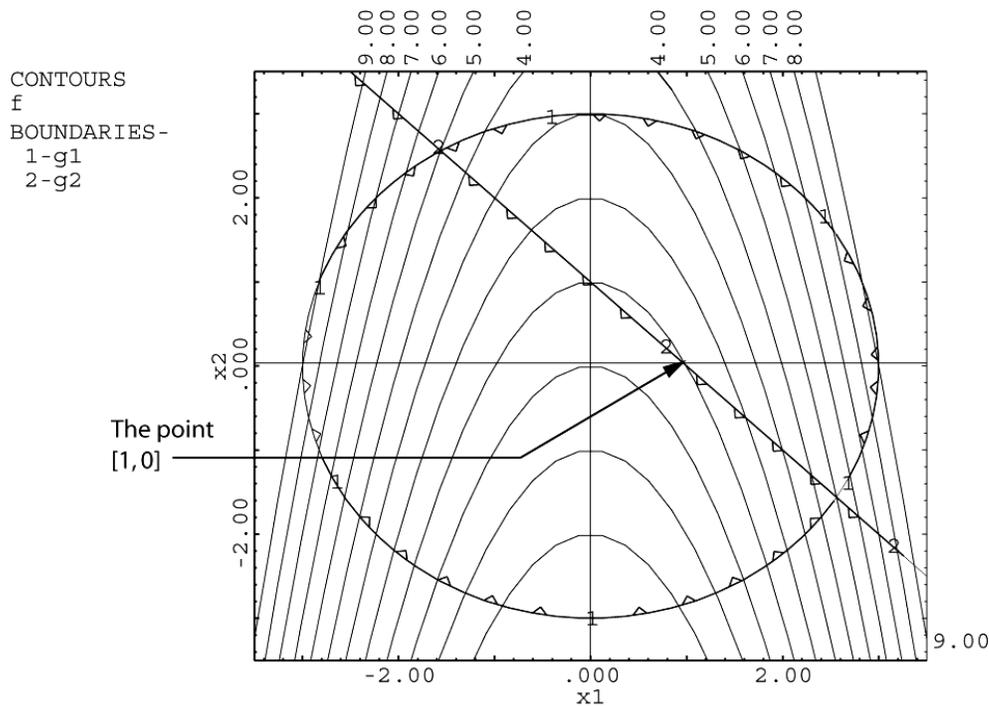


Fig. 6.7 Contour plot and proposed point for Example 4.

Step 1: Change problem to be in the form of (6.3-6.5):

$$\begin{aligned} \text{Max} \quad & f(\mathbf{x}) = -x_1^2 - x_2 \\ \text{s.t.} \quad & g_1(\mathbf{x}): x_1^2 + x_2^2 - 9 \leq 0 \\ & g_2(\mathbf{x}): x_1 + x_2 - 1 \leq 0 \end{aligned}$$

Step 2: See which constraints are binding:

From Fig. 6.7 we can see that $g_1(\mathbf{x})$ is not binding and therefore $\lambda_1 = 0$; $g_2(\mathbf{x})$ is binding so $\lambda_2 \neq 0$

Step 3: Write out the Lagrange multiplier equations represented by (6.10):

$$\frac{\partial f}{\partial x_1} - \lambda_1 \frac{\partial g_1}{\partial x_1} - \lambda_2 \frac{\partial g_2}{\partial x_1} = -2x_1 - \lambda_2(1) = 0 \quad (6.19)$$

$$\frac{\partial f}{\partial x_2} - \lambda_1 \frac{\partial g_1}{\partial x_2} - \lambda_2 \frac{\partial g_2}{\partial x_2} = -1 - \lambda_2(1) = 0 \quad (6.20)$$

Step 4: Substitute in the given point:

Substituting $\mathbf{x}^T = [1 \ 0]$

$$\lambda_2 = -2 \quad \text{from (6.19)}$$

$$\lambda_2 = -1 \quad \text{from (6.20)}$$

Since we cannot find a consistent set of λ 's, and the λ 's are negative as well (either condition being enough to disqualify the point), this point does not satisfy the Kuhn-Tucker conditions and *cannot* be a constrained optimum.

Question: In Examples 3 and 4 we have looked at two points—a constrained min, and point which is not an optimum. Are there any other points which would satisfy the K-T conditions for this problem? Where would a constrained max be found? Would the K-T conditions apply there?

2.5 Unconstrained Problems

We mentioned the K-T conditions also apply to unconstrained problems. This is fortunate since a constrained optimization problem does not have to have a constrained solution. The optimum might be an unconstrained optimum in the interior of the constraints.

If we have an unconstrained optimum to a constrained problem, what happens to the K-T conditions? In this case none of the constraints are binding so *all* of the Lagrange multipliers are zero, from (6.11), so (6.10) becomes,

$$\nabla f(\mathbf{x}^*) - \left[\sum_{i=1}^m \lambda_i^* \nabla g_i(\mathbf{x}^*) \right] = \nabla f(\mathbf{x}^*) = 0$$

Thus we see that the K-T equations simplify to the necessary conditions for an unconstrained optimum when no constraints are binding.

3 The Lagrangian Function

3.1 Definition

It will be convenient for us to define the *Lagrangian Function*:

$$L(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{i=1}^m \lambda_i [g_i(\mathbf{x}) - b_i] \quad (6.21)$$

Note that the Lagrangian function is a function of both \mathbf{x} and $\boldsymbol{\lambda}$. Thus the gradient of the Lagrangian function is made up of partials with respect to \mathbf{x} and $\boldsymbol{\lambda}$:

$$[\nabla L(\mathbf{x}, \boldsymbol{\lambda})]^T = \left[\frac{\partial L}{\partial x_1} \quad \dots \quad \frac{\partial L}{\partial x_n} \quad \frac{\partial L}{\partial \lambda_1} \quad \dots \quad \frac{\partial L}{\partial \lambda_m} \right] \quad (6.22)$$

We will evaluate some of these partials to become familiar with them,

The partial $\frac{\partial L}{\partial x_1}$ is:
$$\frac{\partial L}{\partial x_1} = \frac{\partial f}{\partial x_1} - \sum_{i=1}^m \lambda_i \frac{\partial g_i}{\partial x_1}$$

Similarly, $\frac{\partial L}{\partial x_2}$ is given by,
$$\frac{\partial L}{\partial x_2} = \frac{\partial f}{\partial x_2} - \sum_{i=1}^m \lambda_i \frac{\partial g_i}{\partial x_2}$$

The partial $\frac{\partial L}{\partial \lambda_1}$ is:
$$\frac{\partial L}{\partial \lambda_1} = -[g_1 - b_1]$$

It is convenient, given these results, to split the gradient vector of the Lagrangian function into two parts: the vector containing the partial derivatives with respect to x , written $\nabla_x L$, and the vector containing the partials with respect to λ , written $\nabla_\lambda L$.

The gradient of the Lagrangian function with respect to x can be written in vector form as:

$$\nabla_x L = \nabla f(\mathbf{x}) - \sum_{i=1}^m \lambda_i \nabla g_i(\mathbf{x}) \quad (6.23)$$

so that we could replace (6.10) by $\nabla_x L = \mathbf{0}$ if we wished.

The gradient of the Lagrangian function with respect to λ is:

$$\nabla_{\lambda} L = - \begin{bmatrix} g_1(\mathbf{x}) - b_1 \\ g_2(\mathbf{x}) - b_2 \\ \vdots \\ g_m(\mathbf{x}) - b_m \end{bmatrix} \quad (6.24)$$

3.2 The Lagrangian Function and Optimality

For a problem with equality constraints only, we can compactly state the K-T conditions as,

$$\nabla L = \begin{bmatrix} \nabla_x L \\ \nabla_{\lambda} L \end{bmatrix} = \mathbf{0} \quad (6.25)$$

For a problem with inequality constraints as well, the main condition of the K-T equations, (6.10), can be stated as,

$$\nabla_x L = \mathbf{0} \quad (6.26)$$

Thus we can consider that at an optimum, there exist λ^* and \mathbf{x}^* such that \mathbf{x}^* is a stationary point of the Lagrangian function.

The Lagrangian function provides information about how the objective and binding constraints together affect an optimum. Suppose we are at a constrained optimum. If we were to change the objective function, this would clearly have an effect on the solution. Likewise, if we were to change the binding constraints (perhaps by changing the right hand sides), these changes would also affect the value of the solution. The Lagrangian function tells how these changes trade-off against each other,

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) - \sum_{i=1}^m \lambda_i [g_i(\mathbf{x}) - b_i]$$

The Lagrange multipliers serve as “weighting factors” between the individual constraints and the objective. Appropriately, the multipliers have units of (objective function/constraint function). Thus if our objective had units of pounds, and constraint i had units of inches, Lagrange multiplier i would have units of pounds per inch.

3.3 Interpreting Values of Lagrange Multipliers

Thus far we have solved for Lagrange multipliers, but we have not attached any significance to their values. We will use the Lagrangian function to help us interpret what their values mean.

We will start with the Lagrangian function at an optimum:

$$L(\mathbf{x}^*, \boldsymbol{\lambda}^*) = f(\mathbf{x}^*) - \sum_{i=1}^m \lambda_i^* [g_i(\mathbf{x}^*) - b_i] \quad (6.27)$$

Suppose now we consider the right-hand side of constraint i , b_i , to be a variable. How does the optimal solution change as we change b_i ? To answer this question, we need to find $\frac{df^*}{db_i}$.

That is, we need to find how the *optimal* value of the objective changes as we change the right hand side, b_i . The Lagrangian function, which relates how constraints and the objective interact at the optimum, can give us this information.

We will be concerned only with small perturbations at the optimum. This allows us to ignore *nonbinding* inequality constraints, which will be treated as if they were not there. Thus instead of m constraints, we will have m^* constraints, which is the set of equality and binding inequality constraints.

At an optimum, the value of L becomes the same as the value of f . This is because all of the terms in braces go to zero,

$$L(\mathbf{x}^*, \boldsymbol{\lambda}^*) = f(\mathbf{x}^*) - \left[\sum_{i=1}^{m^*} \lambda_i^* [g_i(\mathbf{x}^*) - b_i] \right] = f(\mathbf{x}^*)$$

since all constraints in our set m^* are binding. At the optimum therefore, $\frac{df^*}{db_i} = \frac{dL^*}{db_i}$.

As we change b_i , we would expect \mathbf{x}^* and $\boldsymbol{\lambda}^*$ to change, so we need to consider \mathbf{x} and $\boldsymbol{\lambda}$ themselves to be functions of b_i , i.e., $\mathbf{x}(b_i)$, $\boldsymbol{\lambda}(b_i)$. Then by the chain rule:

$$\frac{df^*}{db_i} = \frac{dL^*}{db_i} = \frac{\partial \mathbf{x}^T}{\partial b_i} \nabla_{\mathbf{x}} L + \frac{\partial \boldsymbol{\lambda}^T}{\partial b_i} \nabla_{\boldsymbol{\lambda}} L + \frac{\partial L}{\partial b_i} \quad (6.28)$$

At the optimum $\nabla_{\mathbf{x}} L = 0$ and $\nabla_{\boldsymbol{\lambda}} L = 0$, leaving only $\frac{\partial L}{\partial b_i}$.

From the Lagrangian function, $\frac{\partial L}{\partial b_i} = \lambda_i$,

Thus we have the result,

$$\frac{df^*}{db_i} = \lambda_i^* \quad (6.29)$$

The Lagrange multipliers provide us with *sensitivity information* about the optimal solution. They tell us how the optimal objective value would change if we changed the right-hand side of the constraints. This can be very useful information, telling us, for example, how we

could expect the optimal weight of a truss to change if we relaxed the right-hand side of a binding stress constraint (i.e. increased the allowable value of the stress constraint).

Caution: The sensitivity information provided by the Lagrange multipliers is valid only for small changes about the optimum (since for nonlinear functions, derivatives change as we move from point to point) and assumes *that the same constraints* are binding at the perturbed optimum.

3.3.1 Example 5: Interpreting the Value of the Lagrange Multipliers

In Example 1, Section 2.4.1, we solved the following problem

$$\begin{array}{ll} \text{Min} & f(\mathbf{x}) = 2x_1^2 + 4x_2^2 \\ \text{s.t.} & g_1: 3x_1 + 2x_2 = 12 \end{array}$$

We found the optimal solution to be:

$$x_1 = 3.2727, x_2 = 1.0909, \lambda = 4.3636, \text{ at which point } f^* = 26.18.$$

What would be the expected change in the objective be if we increased the right-hand side of the constraint from 12 to 13? From (6.29),

$$\Delta f^* \approx \lambda_i^* \Delta b_i$$

For $\Delta b = 1$, the change in the objective should be approximately 4.36.

If we change the right hand side and re-optimize the problem, the new optimum is, $x_1 = 3.5454, x_2 = 1.1818, \lambda = 4.7272$, at which point $f^* = 30.72$. The actual change in the objective is 4.54. (Indeed, it is the average of the two λ 's.)

Thus, without optimizing, the Lagrange multiplier provides an estimate of how much the objective would change per unit change in the constraint right hand side. This helps us evaluate how sensitive the optimum is to changes.

Sometimes the objective represents profit, and the right hand side of a constraint is viewed as a *resource* which can be purchased. The value of the Lagrange multiplier is a breakpoint between realizing a net increase in profit or a net loss. If, for a binding constraint, we can purchase more right hand side for less than the Lagrange multiplier, net profit will be positive. If not, the cost of the resource will outweigh the increase in profit.

3.4 Necessary and Sufficient Conditions

The K-T Conditions we have presented in previous sections are necessary conditions for a constrained optimum. That is, for a point to be a candidate optimal solution, it must be possible to find values of λ that satisfy (6.9)-(6.13). If we cannot find such λ , then the candidate point *cannot be* a constrained optimal solution.

Note, however, that as part of the KT conditions we require the constraint gradients, $\nabla g_i(x^*)$, to be independent at the optimal solution; otherwise it is possible, although unlikely, that we could have a point be a constrained optimum and not satisfy the KT conditions.

If a point satisfies the KT conditions, then it is a *candidate* optimal solution. As we have seen, the necessary conditions can hold at a constrained max, constrained min, or a point that is neither. An example showing how this might occur is given below:

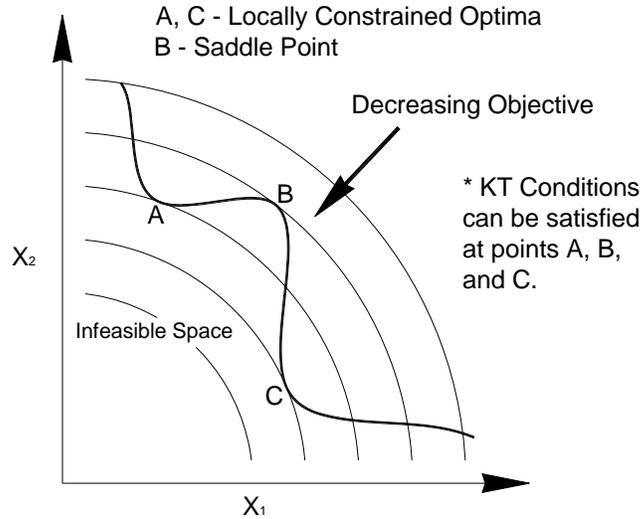


Fig. 6.8. Points where the K-T equations would be satisfied.

For an unconstrained optimum we saw that *sufficient* conditions for a minimum were that $\nabla f = 0$ and, the Hessian, $\nabla^2 f(\mathbf{x})$, is positive definite.

Likewise, for a constrained optimum, sufficient conditions for a point to be a *constrained minimum* are the K-T equations are satisfied (6-9-6.13) and the Hessian of the Lagrangian function with respect to \mathbf{x} , $\nabla_{\mathbf{x}}^2 L(\mathbf{x}^*, \boldsymbol{\lambda}^*)$, is positive definite, where,

$$\nabla_{\mathbf{x}}^2 L(\mathbf{x}^*, \boldsymbol{\lambda}^*) = \nabla^2 f(\mathbf{x}^*) - \sum_{i=1}^m \lambda_i^* \nabla^2 g_i(\mathbf{x}^*) \quad (6.30)$$

Some further discussion is needed, however. If we write the condition of positive definiteness as,

$$\mathbf{y}^T \nabla_{\mathbf{x}}^2 L(\mathbf{x}^*, \boldsymbol{\lambda}^*) \mathbf{y} > 0 \quad (6.31)$$

The vectors \mathbf{y} must satisfy,

$$\mathbf{J}(\mathbf{x}^*) \mathbf{y} > 0 \quad (6.32)$$

Where $\mathbf{J}(\mathbf{x}^*)$ is the Jacobian matrix of the constraints (matrix whose rows are the gradients of the constraints) active at \mathbf{x}^* . These vectors comprise a tangent plane and are orthogonal to the gradients of the active constraints. For more information about the sufficient conditions, see Luenberger, (1984), Fletcher (1987) or Edgar et al. (2001).

4 References

- Fletcher, R., *Practical Methods of Optimization 2nd Ed.*, Wiley, 1987.
Luenberger, D. G., *Linear and Nonlinear Programming, 2nd Ed.*, Addison Wesley, 1984
Edgar, T. F., Himmelblau, D. M., Lasdon, L. S., *Optimization of Chemical Processes, 2nd Ed.*, McGraw-Hill, 2001.

CHAPTER 7

CONSTRAINED OPTIMIZATION 2: SQP AND GRG

1 Introduction

In the previous chapter we examined the necessary and sufficient conditions for a constrained optimum. We did not, however, discuss any algorithms for constrained optimization. That is the purpose of this chapter.

In an oft referenced study done in 1980¹, dozens of nonlinear algorithms were tested on roughly 100 different nonlinear problems. The top-ranked algorithm was SQP. Of the five top algorithms, two were SQP and three were GRG. Although many different algorithms have been proposed, based on these results, we will study only these two.

SQP works by solving for where the KT equations are satisfied. SQP is a very efficient algorithm in terms of the number of function calls needed to get to the optimum. It converges to the optimum by simultaneously improving the objective and tightening feasibility of the constraints. Only the optimal design is guaranteed to be feasible; intermediate designs may be infeasible.

The GRG algorithm works by computing search directions which improve the objective and satisfy the constraints, and then conducting line searches in a very similar fashion to the algorithms we studied in Chapter 3. GRG requires more function evaluations than SQP, but it has the desirable property that it stays feasible once a feasible point is found. If the optimization process is halted before the optimum is reached, the designer is guaranteed to have in hand a better design than the starting design. GRG also appears to be more robust (able to solve a wider variety of problems) than SQP, so for engineering problems it is often the algorithm tried first.

2 The Sequential Quadratic Programming (SQP) Algorithm

The SQP algorithm was developed in the early 1980's by M. J. D. Powell, a mathematician at Cambridge University. Before we begin describing this algorithm, we need to present some background information.

2.1 The Newton Raphson Method for Solving Nonlinear Equations

If we were to sum up how the SQP methods works in one sentence it would be: the SQP algorithm applies the Newton-Raphson method to solve the Kuhn-Tucker equations. Or, in short, SQP does N-R on the K-T! Thus we will begin by reviewing how the Newton Raphson method works.

¹ Schittkowski, K., "Nonlinear Programming codes: Information, Tests, Performance," *Lecture Notes in Economics and Mathematical Systems*, vol. 183, Springer-Verlag, New York, 1980.

2.1.1 One equation with One Unknown

The N-R method is used to find the solution to sets of nonlinear equations. For example, suppose we wish to find the solution to the equation:

$$x + 2 = e^x$$

We cannot solve for x directly. The N-R method solves the equation in an iterative fashion based on results derived from the Taylor expansion.

First, the equation is rewritten in the form,

$$x + 2 - e^x = 0 \tag{7.1}$$

We then provide a starting estimate of the value of x that solves the equation. This point becomes the point of expansion for a Taylor series:

$$F \approx F^0 + \frac{dF^0}{dx}(x - x^0) \tag{7.2}$$

(For reasons that will become apparent later, we will use F instead of f for our functions here.) We would like to drive the value of the function to zero:

$$0 = F^0 + \frac{dF^0}{dx}(x - x^0) \tag{7.3}$$

If we denote $\Delta x = x - x^0$, and solve for Δx in (7.3):

$$\Delta x = \frac{-F^0}{dF/dx} \tag{7.4}$$

We then add Δx to x^0 to obtain a new guess for the value of x that satisfies the equation, obtain the derivative there, get the new function value, and iterate until the function value or *residual*, goes to zero. The process is illustrated in Fig. 7.1 for the example given in (7.1), with a starting guess $x = 2.0$.

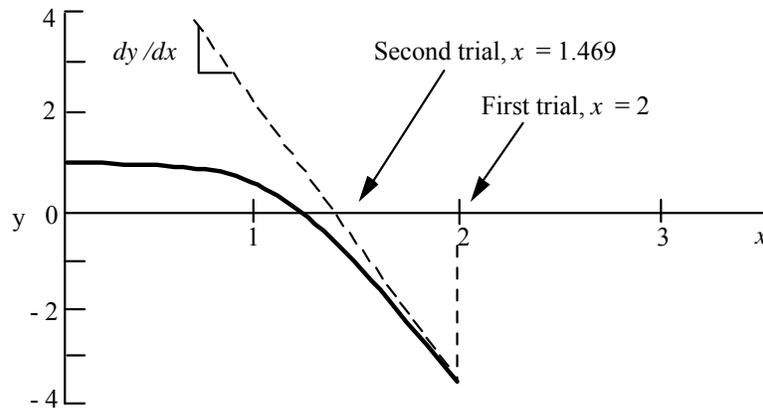


Fig. 7.1 Newton Raphson method on (7.1)

Numerical results are:

K	x	f(x)	df/dx
1	2	-3.389056	-6.389056
2	1.469553	-0.877738	-3.347291
3	1.20732948	-0.13721157	-2.34454106
4	1.14880563	-0.00561748	-2.15442311
5	1.146198212	-0.000010714	-2.146208926
6	1.1461932206	-0.00000000004	-2.1461932254

For simple roots, N-R has *second order convergence*. This means that the number of significant figures in the solution roughly doubles at each iteration. We can see this in the above table, where the value of x at iteration 2 has one significant figure (1); at iteration 3 it has one (1); at iteration 4 it has three (1.14); at iteration 5 it has six (1.14619), and so on. We also see that the error in the residual, as indicated by the number of zeros after the decimal point, also decreases in this fashion, i.e., the number of zeros roughly doubles at each iteration.

2.1.2 Multiple Equations with Multiple Unknowns

The N-R method is easily extended to solve n equation in n unknowns. Writing the Taylor series for the equations in vector form:

$$0 = F_1^0 + (\nabla F_1^0)^T \Delta \mathbf{x}$$

$$0 = F_2^0 + (\nabla F_2^0)^T \Delta \mathbf{x}$$

... ..

$$0 = F_n^0 + (\nabla F_n^0)^T \Delta \mathbf{x}$$

We can rewrite these relationships in matrix form:

$$\begin{bmatrix} (\nabla F_1^0)^\top \\ (\nabla F_2^0)^\top \\ \dots \\ (\nabla F_n^0)^\top \end{bmatrix} \Delta \mathbf{x} = \begin{bmatrix} -F_1^0 \\ -F_2^0 \\ \dots \\ -F_n^0 \end{bmatrix} \quad (7.5)$$

For 2 X 2 System,

$$\begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = \begin{bmatrix} -F_1 \\ -F_2 \end{bmatrix} \quad (7.6)$$

In (7.5) we will denote the vector of residuals as \mathbf{F}^0 . (This violates our notation convention that bold caps represent matrices, not vectors. Just remember that \mathbf{F} is a vector, not a matrix.) We will denote the matrix of coefficients as \mathbf{G} . Equation (7.5) can then be written,

$$\mathbf{G} \Delta \mathbf{x} = -\mathbf{F} \quad (7.7)$$

The solution is obviously

$$\Delta \mathbf{x} = -(\mathbf{G}^{-1}) \mathbf{F} \quad (7.8)$$

2.1.3 Using the N-R method to Solve the Necessary Conditions

In this section we will make a very important connection—we will apply N-R to solve the necessary conditions. Consider for example, a very simple case—an unconstrained problem in two variables. We know the necessary conditions are,

$$\begin{aligned} \frac{\partial f}{\partial x_1} &= 0 \\ \frac{\partial f}{\partial x_2} &= 0 \end{aligned} \quad (7.9)$$

Now suppose we wish to solve these equations using N-R, that is we wish to find \mathbf{x}^* to drive the partial derivatives of f to zero. In terms of notation and discussion this gets a little tricky because the N-R method involves taking derivatives of the equations to be solved, and the equations we wish to solve are composed of derivatives. So when we substitute (7.9) into the N-R method, we end up with *second* derivatives.

For example, if we set $F_1 = \frac{\partial f}{\partial x_1}$ and $F_2 = \frac{\partial f}{\partial x_2}$. Then we can write (7.6) as,

$$\begin{bmatrix} \frac{\partial}{\partial x_1} \left(\frac{\partial f}{\partial x_1} \right) & \frac{\partial}{\partial x_2} \left(\frac{\partial f}{\partial x_1} \right) \\ \frac{\partial}{\partial x_1} \left(\frac{\partial f}{\partial x_2} \right) & \frac{\partial}{\partial x_2} \left(\frac{\partial f}{\partial x_2} \right) \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = \begin{bmatrix} -\left(\frac{\partial f}{\partial x_1} \right) \\ -\left(\frac{\partial f}{\partial x_2} \right) \end{bmatrix} \quad (7.10)$$

or,

$$\begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = \begin{bmatrix} -\frac{\partial f}{\partial x_1} \\ -\frac{\partial f}{\partial x_2} \end{bmatrix} \quad (7.11)$$

which should be familiar from Chapter 3, because (7.11) can be written in vector form as,

$$\mathbf{H}\Delta\mathbf{x} = -\nabla f \quad (7.12)$$

and the solution is,

$$\Delta\mathbf{x} = -(\mathbf{H}^{-1})\nabla f \quad (7.13)$$

We recognize (7.12-7.13) as Newton's method for solving for an unconstrained optimum. Thus we have the important result that *Newton's method is the same as applying N-R on the necessary conditions for an unconstrained problem*. From the properties of the N-R method, we know that if Newton's method converges (and recall that it doesn't always converge), it will do so with second order convergence—which is very fast.

Now just to indicate where we are heading, after we introduce the SQP approximation, the next step is to show that the SQP method is the same as doing N-R on the necessary conditions *for a constrained problem* (with some tweaks to make it efficient and to ensure it converges).

2.2 Constrained Optimization: Equality Constraints

2.2.1 Problem Definition

We will start with a problem which only has equality constraints. We recall that when we only have equality constraints, we do not have to worry about complementary slackness which makes things simpler. So the problem we will focus on is,

$$\text{Min } f(\mathbf{x}) \quad (7.14)$$

$$\text{st. } g_i(\mathbf{x}) - b_i = 0 \quad i = 1, 2, \dots, m \quad (7.15)$$

The necessary conditions for a constrained optimal solution are:

$$\nabla f - \sum_{i=1}^m \lambda_i \nabla g_i = \mathbf{0} \quad (7.16)$$

$$g_i - b_i = 0 \quad i = 1, \dots, m \quad (7.17)$$

2.2.2 The SQP Approximation

As we have previously mentioned in Chapter 6, a problem with a quadratic objective and linear constraints is known as a *quadratic programming problem*. These problems have a special name because the K-T equations are linear and are easily solved. We will make a quadratic programming approximation at the point \mathbf{x}^0 to the problem given by (7.14-7.15)

$$f_a = f^0 + (\nabla f^0)^T \Delta \mathbf{x} + \frac{1}{2} \Delta \mathbf{x}^T \nabla_x^2 L^0 \Delta \mathbf{x} \quad (7.18)$$

$$g_{i,a} = g_i^0 + (\nabla g_i^0)^T \Delta \mathbf{x} = b_i \quad i = 1, \dots, m \quad (7.19)$$

where the subscript a is used in f_a to indicate the approximation. Close examination of (7.18) shows something unexpected. Instead of $\nabla^2 f$ as we would normally have if we were doing a Taylor approximation of the objective, we have $\nabla_x^2 L$, the Hessian of the Lagrangian function with respect to \mathbf{x} . Why is this case? It is directly tied to applying N-R on the K-T, as we will presently show. For now we will just accept that the objective uses the Hessian of the Lagrangian instead of the Hessian of the objective.

We will solve the QP approximation, (7.18-7.19), by solving the K-T equations for this problem, which, as mentioned, are linear. These equations are given by,

$$\nabla f_a - \sum_{i=1}^m \lambda_i \nabla g_{i,a} = \mathbf{0} \quad (7.20)$$

$$g_{i,a} - b_i = 0 \quad \text{for } i = 1, \dots, m \quad (7.21)$$

Since, for example, from (7.18),

$$\nabla f_a = \nabla f^0 + \nabla_x^2 L^0 \Delta \mathbf{x}$$

we can also write these equations in terms of the original problem,

$$\nabla f^0 + \nabla_x^2 L^0 \Delta \mathbf{x} - \sum_{i=1}^m \lambda_i \nabla g_i^0 = \mathbf{0} \quad (7.22)$$

$$g_i^0 + (\nabla g_i^0)^T \Delta \mathbf{x} - b_i = 0 \quad \text{for } i = 1, \dots, m \quad (7.23)$$

These are a linear set of equations we can readily solve, as shown in the example in the next section. For Section 2.2.4, we will want to write these equations even more concisely. If we define the following matrices and vectors,

$$\mathbf{J}^0 = \begin{bmatrix} (\nabla g_1^0)^T \\ (\nabla g_2^0)^T \\ \vdots \\ (\nabla g_m^0)^T \end{bmatrix} \quad (\mathbf{J}^0)^T = [\nabla g_1^0, \nabla g_2^0, \dots, \nabla g_m^0]$$

$$\mathbf{g}^0 = \begin{bmatrix} g_1^0 \\ g_2^0 \\ \vdots \\ g_m^0 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \quad \nabla_x^2 L^0 = \nabla^2 f^0 - \sum_{i=1}^m \lambda_i \nabla^2 g_i^0$$

We can write (7.22-7.23) as,

$$\nabla_x^2 L^0 \Delta \mathbf{x} + (-\mathbf{J}^0)^T \boldsymbol{\lambda} = -\nabla f^0 \quad (7.24)$$

$$\mathbf{J}^0 \Delta \mathbf{x} = -(\mathbf{g}^0 - \mathbf{b}) \quad (7.25)$$

Again, to emphasize, this set of equations represents the solution to (7.18-7.19).

2.2.3 Example 1: Solving the SQP Approximation

Suppose we have as our approximation the following,

$$f_a = 3 + \begin{bmatrix} 3 & 2 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \Delta x_1 & \Delta x_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} \quad (7.26)$$

$$g_a = 5 + \begin{bmatrix} 1 & 3 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = 0$$

We can write out the K-T equations for this approximation as,

$$\nabla f_a = \begin{bmatrix} 3 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} - \lambda \begin{bmatrix} 1 \\ 3 \end{bmatrix} = 0 \quad (7.27)$$

$$g_a = 5 + \begin{bmatrix} 1 & 3 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = 0$$

We can rewrite these equations in matrix form as,

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -3 \\ 1 & 3 & 0 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \lambda \end{bmatrix} = \begin{bmatrix} -3 \\ -2 \\ -5 \end{bmatrix} \quad (7.28)$$

The solution is,

$$\begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \lambda \end{bmatrix} = \begin{bmatrix} -2.6 \\ -0.8 \\ 0.4 \end{bmatrix} \quad (7.29)$$

Observations: This calculation represents the main step in an iteration of the SQP algorithm which solves a *sequence of quadratic programs*. If we wanted to continue, we would add $\Delta \mathbf{x}$ to our current \mathbf{x} , update the Lagrangian Hessian, make a new approximation, solve for that solution, and continue iterating in this fashion.

If we ever reach a point where $\Delta \mathbf{x}$ goes to zero as we solve for the optimum of the approximation, *the original K-T equations are satisfied*. We can see this by examining (7.22-7.23). If $\Delta \mathbf{x}$ is zero, we have,

$$\nabla f + \underbrace{\nabla_x^2 L}_{=0} \Delta \mathbf{x} - \sum_{i=1}^m \lambda_i^* \nabla g_i = \mathbf{0} \quad (7.30)$$

$$g_i + \underbrace{(\nabla g_i)^T}_{=0} \Delta \mathbf{x} - b_i = 0 \quad \text{for } i = 1, \dots, m \quad (7.31)$$

which then match (7.16-7.17).

2.2.4 N-R on the K-T Equations for Problems with Equality Constraints

In this section we wish to look at applying the N-R method to the original K-T equations. The K-T equations for a problem with equality constraints only, as given by (7.16-7.17), are,

$$\nabla f - \sum_{i=1}^m \lambda_i \nabla g_i = \mathbf{0} \quad (7.32)$$

$$g_i - b_i = 0 \quad i = 1, \dots, m \quad (7.33)$$

Now suppose we wish to solve these equations using the N-R method. To implement N-R we would have,

$$\begin{bmatrix} (\nabla_x F_1)^T & (\nabla_\lambda F_1)^T \\ \vdots & \vdots \\ (\nabla_x F_n)^T & (\nabla_\lambda F_n)^T \\ (\nabla_x g_1)^T & (\nabla_\lambda g_1)^T \\ \vdots & \vdots \\ (\nabla_x g_m)^T & (\nabla_\lambda g_m)^T \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} -F_1 \\ \vdots \\ -F_n \\ -(g_1 - b_1) \\ \vdots \\ -(g_m - b_m) \end{bmatrix} \quad (7.34)$$

where F_1 for example, is given by,

$$F_1 = \frac{\partial f}{\partial x_1} - \sum_{i=1}^m \lambda_i \frac{\partial g_i}{\partial x_1} \quad (7.35)$$

If we substitute (7.35) into matrix (7.32), the first row becomes,

$$\left[\frac{\partial^2 f}{\partial x_1^2} - \sum_{i=1}^m \lambda_i \frac{\partial^2 g_i}{\partial x_1^2} \right], \left[\frac{\partial^2 f}{\partial x_2 \partial x_1} - \sum_{i=1}^m \lambda_i \frac{\partial^2 g_i}{\partial x_2 \partial x_1} \right], \dots, \left[\frac{\partial^2 f}{\partial x_n \partial x_1} - \sum_{i=1}^m \lambda_i \frac{\partial^2 g_i}{\partial x_n \partial x_1} \right], \left[-\frac{\partial g_1}{\partial x_1} \right], \left[-\frac{\partial g_2}{\partial x_1} \right], \dots, \left[-\frac{\partial g_m}{\partial x_1} \right]$$

Recalling,

$$\nabla_x^2 L = \nabla^2 f - \sum_{i=1}^m \lambda_i \nabla^2 g_i$$

And using the matrices we defined above,

$$\mathbf{J}^0 = \begin{bmatrix} (\nabla g_1^0)^T \\ (\nabla g_2^0)^T \\ \vdots \\ (\nabla g_m^0)^T \end{bmatrix} \quad (\mathbf{J}^0)^T = [\nabla g_1^0, \nabla g_2^0, \dots, \nabla g_m^0]$$

$$\mathbf{g}^0 = \begin{bmatrix} g_1^0 \\ g_2^0 \\ \vdots \\ g_m^0 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \quad \nabla_x^2 L^0 = \nabla^2 f^0 - \sum_{i=1}^m \lambda_i \nabla^2 g_i^0$$

we can rewrite (7.34) as,

$$\begin{bmatrix} \nabla_x^2 L^0 & (-\mathbf{J}^0)^T \\ \mathbf{J}^0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} - \mathbf{x}^0 \\ \boldsymbol{\lambda} - \boldsymbol{\lambda}^0 \end{bmatrix} = \begin{bmatrix} -\nabla f^0 + (\mathbf{J}^0)^T \boldsymbol{\lambda}^0 \\ -(\mathbf{g}^0 - \mathbf{b}) \end{bmatrix} \quad (7.36)$$

If we do the matrix multiplications we have

$$\begin{aligned}\nabla_x^2 L^0 \Delta \mathbf{x} + (-\mathbf{J}^0)^T (\boldsymbol{\lambda} - \boldsymbol{\lambda}^0) &= -\nabla f^0 + (\mathbf{J}^0)^T \boldsymbol{\lambda}^0 \\ \mathbf{J}^0 \Delta \mathbf{x} &= -(\mathbf{g}^0 - \mathbf{b})\end{aligned}\tag{7.37}$$

and collecting terms,

$$\begin{aligned}\nabla_x^2 L^0 \Delta \mathbf{x} + (-\mathbf{J}^0)^T \boldsymbol{\lambda} &= -\nabla f^0 \\ \mathbf{J}^0 \Delta \mathbf{x} &= -(\mathbf{g}^0 - \mathbf{b})\end{aligned}\tag{7.38}$$

which equations are the same as (7.24-7.25). Thus we see that *doing a N-R iteration on the K-T equations is the same as solving for the optimum of the QP approximation*. This is the reason we use the Hessian of the Lagrangian function rather than the Hessian of the objective in the approximation.

2.3 Constrained Optimization: Inequality and Equality Constraints

In the previous section we considered equality constraints only. We need to extend these results to the general case. We will state this problem as

$$\begin{aligned}\text{Min} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & g_i(\mathbf{x}) - b_i \geq 0 \quad i = 1, \dots, k \\ & g_i(\mathbf{x}) - b_i = 0 \quad i = k + 1, \dots, m\end{aligned}\tag{7.39}$$

The quadratic approximation at point \mathbf{x}^0 is:

$$\begin{aligned}\text{Min} \quad & f_a = f^0 + (\nabla f^0)^T \Delta \mathbf{x} + \frac{1}{2} (\Delta \mathbf{x})^T \nabla_x^2 L^0 \Delta \mathbf{x} \\ \text{s.t.} \quad & g_{i,a} : \quad g_i^0 + (\nabla g_i^0)^T \Delta \mathbf{x} \geq b_i \quad i = 1, 2, \dots, k \\ & \quad \quad \quad g_i^0 + (\nabla g_i^0)^T \Delta \mathbf{x} = b_i \quad i = k + 1, \dots, m\end{aligned}\tag{7.40}$$

Notice that the approximations are a function only of $\Delta \mathbf{x}$. All gradients and the Lagrangian hessian in (7.40) are evaluated at the point of expansion and so represent known quantities.

In the article where Powell describes this algorithm,² he makes a significant statement at this point. Quoting, "The extension of the Newton iteration to take account of inequality constraints on the variables arises from the fact that the value of $\Delta \mathbf{x}$ that solves (7.39) can

² Powell, M.J.D., "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations," *Numerical Analysis, Dundee 1977*, Lecture Notes in Mathematics no. 630, Springer-Verlag, New York, 1978

also be found by solving a quadratic programming problem. Specifically, $\Delta \mathbf{x}$ is the value that makes the quadratic function in (7.40) stationary.”

Further, the value of λ for the K-T conditions is equal to the vector of Lagrange multipliers of the quadratic programming problem. Thus solving the quadratic objective and linear constraints in (7.40) is the *same as solving the N-R iteration on the original K-T equations*.

The main difficulty in extending SQP to the general problem has to do with the complementary slackness condition. This equation is non-linear, and so makes the QP problem nonlinear. We recall that complementary slackness basically enforces that either a constraint is binding or the associated Lagrange multiplier is zero. Thus we can incorporate this condition if we can develop a method to *determine which inequality constraints are binding at the optimum*. An example of a modern solution technique is given by Goldfarb and Idnani.³ This algorithm starts out by solving for the unconstrained optimum to the problem and evaluating which constraints are violated. It then moves to add in these constraints until it is at the optimum. Thus it tends to drive to the optimum from infeasible space.

There are other important details to develop a realistic, efficient SQP algorithm. For example, the QP approximation involves the Lagrangian hessian matrix, which involves second derivatives. As you might expect, we don't evaluate the Hessian directly but approximate it using a quasi-Newton update, such as the BFGS update.

Recall that updates use differences in \mathbf{x} and differences in gradients to estimate second derivatives. To estimate $\nabla_x^2 L$ we will need to use differences in the gradient of the Lagrangian function,

$$\nabla_x L = \nabla f - \sum_{i=1}^m \lambda_i \nabla g_i$$

Note that to evaluate this gradient we need values for λ_i . We will get these from our solution to the QP problem. Since our update stays positive definite, we don't have to worry about the method diverging, like Newton's method does for unconstrained problems.

2.4 Comments on the SQP Algorithm

The SQP algorithm has the following characteristics,

- The algorithm is very fast. It is the most efficient optimization algorithm available today.
- Because it does not rely on a traditional line search, it is often more accurate in identifying an optimum.

³ Goldfarb, D., and A. Idnani, "A Numerically Stable Dual Method for Solving Strictly Convex Quadratic Programs," *Math. Programming*, v. 27, 1983, p.1-33.

- The efficiency of the algorithm is partly because it does not enforce feasibility of the constraints at each step. Rather it gradually enforces feasibility as part of the K-T conditions. It is only guaranteed to be feasible at the optimum.

Relative to engineering problems, there are some drawbacks:

- Because it can go infeasible during optimization—sometimes by relatively large amounts—it can crash engineering models.
- It is more sensitive to numerical noise and/or error in derivatives than GRG.
- If we terminate the optimization process before the optimum is reached, SQP does not guarantee that we will have in-hand a better design than we started with. GRG does guarantee this.

2.5 Summary of Steps for SQP Algorithm

1. Make a QP approximation to the original problem. For the first iteration, use a Lagrangian Hessian equal to the identity matrix.
2. Solve for the optimum to the QP problem. As part of this solution, values for the Lagrange multipliers are obtained.
3. Execute a simple line search by first stepping to the optimum of the QP problem. So the initial step is $\Delta \mathbf{x}$, and $\mathbf{x}^{new} = \mathbf{x}^{old} + \Delta \mathbf{x}$. See if at this point a penalty function, composed of the values of the objective and violated constraints, is reduced. If not, cut back the step size until the penalty function is reduced. The penalty function is given by $P = f + \sum_{i=1}^{vio} \lambda_i |g_i|$ where the summation is done over the set of violated constraints, and the absolute values of the constraints are taken. The Lagrange multipliers act as scaling or weighting factors between the objective and violated constraints.
4. Evaluate the Lagrangian gradient at the new point. Calculate the difference in \mathbf{x} and in the Lagrangian gradient, $\boldsymbol{\gamma}$. Update the Lagrangian Hessian using the BFGS update.
5. Return to Step 1 until $\Delta \mathbf{x}$ is sufficiently small. When $\Delta \mathbf{x}$ approaches zero, the K-T conditions for the original problem are satisfied.

2.6 Example of SQP Algorithm

Find the optimum to the problem,

$$\begin{aligned} \text{Min} \quad & f(\mathbf{x}) = x_1^4 - 2x_2x_1^2 + x_2^2 + x_1^2 - 2x_1 + 5 \\ \text{s.t.} \quad & g(\mathbf{x}) = -(x_1 + 0.25)^2 + 0.75x_2 \geq 0 \end{aligned}$$

starting from the point $[-1, 4]$. A contour plot of the problem is shown in Fig. 7.2. This problem is interesting for several reasons: the objective is quite eccentric at the optimum, the algorithm starts at point where the search direction is pointing away from the optimum, and the constraint boundary at the starting point has a slope opposite to that at the optimum.

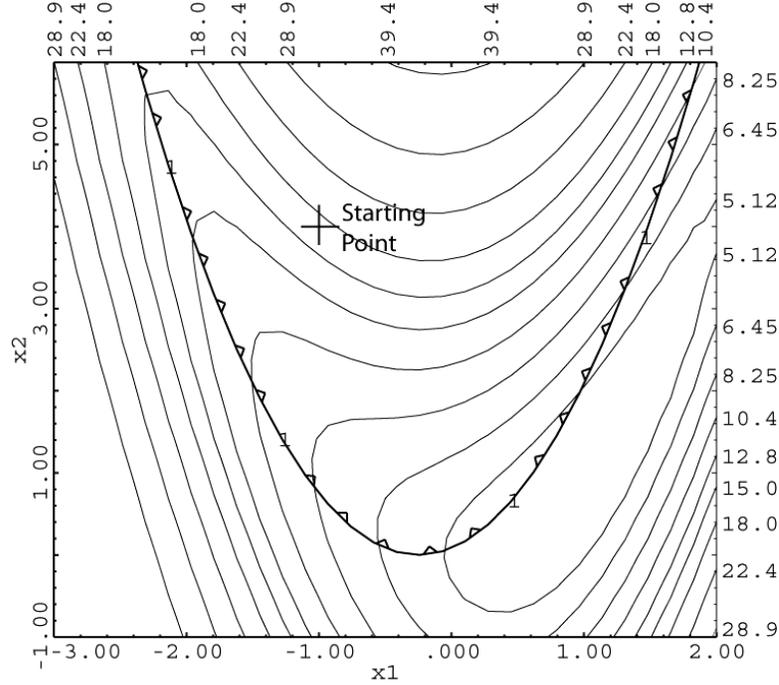


Fig. 7.2. Contour plot of example problem for SQP algorithm.

Iteration 1

We calculate the gradients, etc. at the beginning point. The Lagrangian Hessian is initialized to the identity matrix.

$$\text{At } (\mathbf{x}^0)^T = [-1, 4], f^0 = 17, (\nabla f^0)^T = [8, 6], \nabla^2 L^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$g^0 = 2.4375, (\nabla g^0)^T = [1.5, 0.75]$$

Based on these values, we create the first approximation,

$$f_a = 17.0 + [8 \ 6] \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \Delta x_1 & \Delta x_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix}$$

$$g_a = 2.4375 + [1.5 \ 0.75] \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} \geq 0$$

We will assume the constraint is binding. Then the K-T conditions for the optimum of the approximation are given by the following equations:

$$\nabla f_a - \lambda \nabla g_a = 0$$

$$g_a = 0$$

These equations can be written as,

$$\begin{aligned} 8 + \Delta x_1 - \lambda(1.5) &= 0 \\ 6 + \Delta x_2 - \lambda(0.75) &= 0 \\ 2.4375 + 1.5\Delta x_1 + 0.75\Delta x_2 &= 0 \end{aligned}$$

The solution to this set of equations is $\Delta x_1 = -0.5$, $\Delta x_2 = -2.25$, $\lambda = 5.00$

The proposed step is, $\mathbf{x}^1 = \mathbf{x}^0 + \Delta \mathbf{x} = \begin{bmatrix} -1 \\ 4 \end{bmatrix} + \begin{bmatrix} -0.5 \\ -2.25 \end{bmatrix} = \begin{bmatrix} -1.5 \\ 1.75 \end{bmatrix}$

Before we accept this step, however, we need to check the penalty function,

$$P = f + \sum_{i=1}^{vio} \lambda_i |g_i|$$

to make sure it decreased with the step. At the starting point, the constraint is satisfied, so the penalty function is just the value of the objective, $P = 17$. At the proposed point the objective value is $f = 10.5$ and the constraint is slightly violated with $g = -0.25$. The penalty function is therefore, $P = 10.5 + 5.0 * |-0.25| = 11.75$. Since this is less than 17, we accept the full step. Contours of the first approximation and the path of the first step are shown in Fig. 7.3.

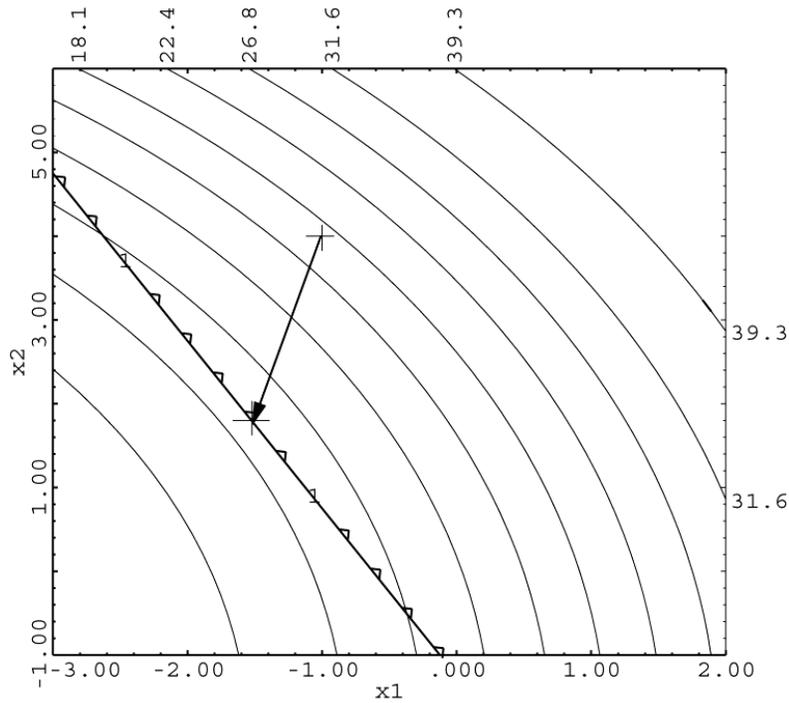


Fig. 7.3 The first SQP approximation and step.

Iteration 2

$$\text{At } (\mathbf{x}^1)^T = [-1.5 \ 1.75], \quad f^1 = 10.5, \quad (\nabla f^1)^T = [-8.0 \ -1.0],$$

$$g^1 = -0.25, \quad (\nabla g^1)^T = [2.5 \ 0.75]$$

We now need to update the Hessian of the Lagrangian. To do this we need the Lagrangian gradient at \mathbf{x}^0 and \mathbf{x}^1 . (Note that we use the same Lagrange multiplier, λ^1 , for both gradients.)

$$\nabla L(\mathbf{x}^0, \lambda^1) = \begin{bmatrix} 8.0 \\ 6.0 \end{bmatrix} - (5.0) \begin{bmatrix} 1.5 \\ 0.75 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 2.25 \end{bmatrix}$$

$$\nabla L(\mathbf{x}^1, \lambda^1) = \begin{bmatrix} -8.0 \\ -1.0 \end{bmatrix} - (5.0) \begin{bmatrix} 2.5 \\ 0.75 \end{bmatrix} = \begin{bmatrix} -20.5 \\ -4.75 \end{bmatrix}$$

$$\gamma^0 = \nabla L(\mathbf{x}^1, \lambda^1) - \nabla L(\mathbf{x}^0, \lambda^1) = \begin{bmatrix} -21.0 \\ -7.0 \end{bmatrix}$$

$$\Delta \mathbf{x}^0 = \begin{bmatrix} -1.5 \\ 1.75 \end{bmatrix} - \begin{bmatrix} -1.0 \\ 4.0 \end{bmatrix} = \begin{bmatrix} -0.5 \\ -2.25 \end{bmatrix}$$

From Chapter 3, we will use the BFGS Hessian update,

$$\mathbf{H}^{k+1} = \mathbf{H}^k + \frac{\gamma^k (\gamma^k)^T}{(\gamma^k)^T \Delta \mathbf{x}^k} - \frac{\mathbf{H}^k \Delta \mathbf{x}^k (\Delta \mathbf{x}^k)^T \mathbf{H}^k}{(\Delta \mathbf{x}^k)^T \mathbf{H}^k \Delta \mathbf{x}^k}$$

Substituting:

$$\nabla^2 L^1 = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix} + \frac{\begin{bmatrix} -21.0 \\ -7.0 \end{bmatrix} \begin{bmatrix} -21.0 & -7.0 \end{bmatrix}}{\begin{bmatrix} -21.0 & -7.0 \end{bmatrix} \begin{bmatrix} -0.5 \\ -2.25 \end{bmatrix}} - \frac{\begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix} \begin{bmatrix} -0.5 \\ -2.25 \end{bmatrix} \begin{bmatrix} -0.5 & -2.25 \end{bmatrix} \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix}}{\begin{bmatrix} -0.5 & -2.25 \end{bmatrix} \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix} \begin{bmatrix} -0.5 \\ -2.25 \end{bmatrix}}$$

$$\nabla^2 L^1 = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix} + \begin{bmatrix} 16.8000 & 5.6000 \\ 5.6000 & 1.8667 \end{bmatrix} - \begin{bmatrix} 0.0471 & 0.2118 \\ 0.2118 & 0.9529 \end{bmatrix}$$

$$\nabla^2 L^1 = \begin{bmatrix} 17.7529 & 5.3882 \\ 5.3882 & 1.9137 \end{bmatrix}$$

The second approximation is therefore,

$$f_a = 10.5 + [-8.0 \quad -1.0] \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} + \frac{1}{2} [\Delta x_1 \quad \Delta x_2] \begin{bmatrix} 17.753 & 5.3882 \\ 5.3882 & 1.9137 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix}$$

$$g_a = -0.25 + [2.5 \quad 0.75] \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} \geq 0$$

As we did before, we will assume the constraint is binding. The K-T equations are,

$$\begin{aligned} -8 + 17.753\Delta x_1 + 5.3882\Delta x_2 - \lambda(2.5) &= 0 \\ -1 + 5.3882\Delta x_1 + 1.9137\Delta x_2 - \lambda(0.75) &= 0 \\ -0.25 + 2.5\Delta x_1 + 0.75\Delta x_2 &= 0 \end{aligned}$$

The solution to this set of equations is $\Delta x_1 = 1.6145$, $\Delta x_2 = -5.048$, $\lambda = -2.615$. Because λ is negative, we need to drop the constraint from the picture. (We can see in Fig. 7.4 below that the constraint is not binding at the optimum.) With the constraint dropped, the solution is, $\Delta x_1 = 2.007$, $\Delta x_2 = -5.131$, $\lambda = 0$. This gives a new \mathbf{x} of,

$$\mathbf{x}^2 = \mathbf{x}^1 + \Delta \mathbf{x} = \begin{bmatrix} -1.5 \\ 1.75 \end{bmatrix} + \begin{bmatrix} 2.007 \\ -5.131 \end{bmatrix} = \begin{bmatrix} 0.507 \\ -3.381 \end{bmatrix}$$

However, when we try to step this far, we find the penalty function has increased from 11.75 to 17.48 (this is the value of the objective only—the violated constraint does not enter in to the penalty function because the Lagrange multiplier is zero). We cut the step back. How much to cut back is somewhat arbitrary. We will make the step 0.5 times the original. The new value of \mathbf{x} becomes,

$$\mathbf{x}^2 = \mathbf{x}^1 + \Delta \mathbf{x} = \begin{bmatrix} -1.5 \\ 1.75 \end{bmatrix} + 0.5 \begin{bmatrix} 2.007 \\ -5.131 \end{bmatrix} = \begin{bmatrix} -0.4965 \\ -0.8155 \end{bmatrix}$$

At which point the penalty function is 7.37. So we accept this step. Contours of the second approximation are shown in Fig. 7.4, along with the step taken.

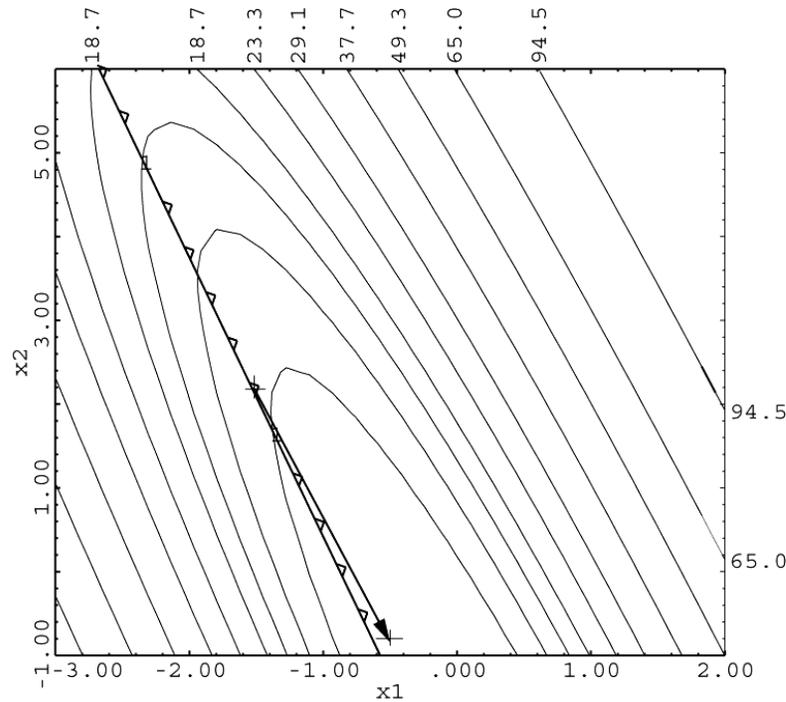


Fig. 7.4 The second approximation and step.

Iteration 3

$$\text{At } (\mathbf{x}^2)^T = [-0.4965 \quad -0.8155], \quad f^2 = 7.367, \quad (\nabla f^2)^T = [-5.102 \quad -2.124],$$

$$g^2 = -0.6724, \quad (\nabla g^2)^T = [0.493 \quad 0.75]$$

$$\nabla L(\mathbf{x}^1, \lambda^2) = \begin{bmatrix} -8.0 \\ -1.0 \end{bmatrix} - (0) \begin{bmatrix} 2.5 \\ 0.75 \end{bmatrix} = \begin{bmatrix} -8.0 \\ -1.0 \end{bmatrix}$$

$$\nabla L(\mathbf{x}^2, \lambda^2) = \begin{bmatrix} -5.102 \\ -2.124 \end{bmatrix} - (0) \begin{bmatrix} 0.493 \\ 0.75 \end{bmatrix} = \begin{bmatrix} -5.102 \\ -2.124 \end{bmatrix}$$

$$\boldsymbol{\gamma}^1 = \nabla L(\mathbf{x}^2, \lambda^2) - \nabla L(\mathbf{x}^1, \lambda^2) = \begin{bmatrix} 2.898 \\ -1.124 \end{bmatrix}$$

$$\Delta \mathbf{x}^1 = \begin{bmatrix} -0.4965 \\ -0.8155 \end{bmatrix} - \begin{bmatrix} -1.5 \\ 1.75 \end{bmatrix} = \begin{bmatrix} 1.004 \\ -2.5655 \end{bmatrix}$$

Based on these vectors, the new Lagrangian Hessian is,

$$\nabla^2 L^2 = \begin{bmatrix} 17.7529 & 5.3882 \\ 5.3882 & 1.9137 \end{bmatrix} + \begin{bmatrix} 1.4497 & -0.5623 \\ -0.5623 & 0.2181 \end{bmatrix} - \begin{bmatrix} 5.8551 & 0.7320 \\ 0.7320 & 0.0915 \end{bmatrix}$$

$$\nabla^2 L^2 = \begin{bmatrix} 13.3475 & 4.0939 \\ 4.0939 & 2.0403 \end{bmatrix}$$

So our next approximation is,

$$f_a = 7.367 + \begin{bmatrix} -5.102 & -2.124 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \Delta x_1 & \Delta x_2 \end{bmatrix} \begin{bmatrix} 13.3475 & 4.0939 \\ 4.0939 & 2.0403 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix}$$

$$g_a = -0.6724 + \begin{bmatrix} 0.493 & 0.75 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} \geq 0$$

The K-T equations, assuming the constraint is binding, are,

$$-5.102 + 13.3475\Delta x_1 + 4.0939\Delta x_2 - \lambda(0.493) = 0$$

$$-2.124 + 4.0939\Delta x_1 + 2.0403\Delta x_2 - \lambda(0.75) = 0$$

$$-0.6724 + 0.493\Delta x_1 + 0.75\Delta x_2 = 0$$

The solution to this set of equations is $\Delta x_1 = 0.1399$, $\Delta x_2 = 0.8046$, $\lambda = 0.1205$.

Our new proposed point is, $\mathbf{x}^3 = \mathbf{x}^2 + \Delta \mathbf{x} = \begin{bmatrix} -0.4965 \\ -0.8155 \end{bmatrix} + \begin{bmatrix} 0.1399 \\ 0.8046 \end{bmatrix} = \begin{bmatrix} -0.3566 \\ -0.0109 \end{bmatrix}$

At this point the penalty function has decreased from 7.37 to 5.85. We accept the full step. A contour plot of the third approximation is shown in Fig. 7.5.

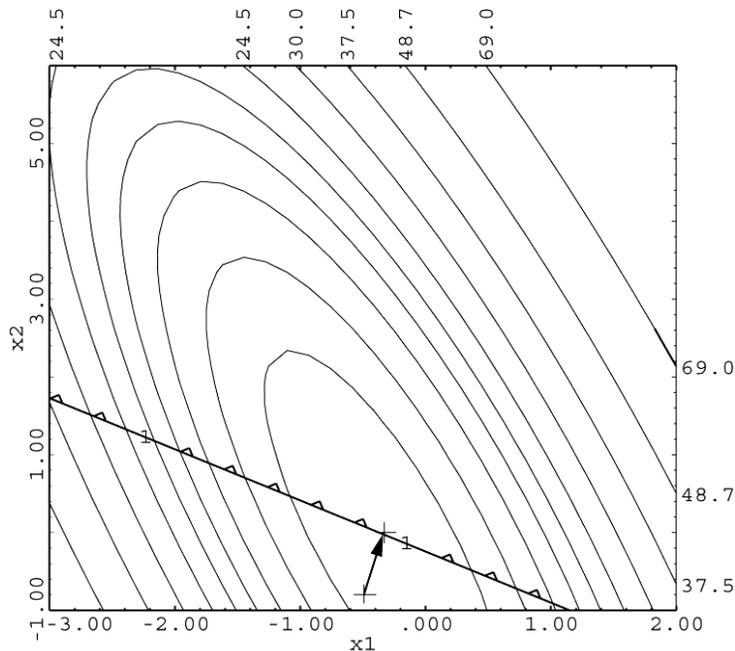


Fig. 7.5 The third approximation and step.

Iteration 4

$$\text{At } (\mathbf{x}^3)^T = [-0.3566 \quad -0.0109], \quad f^3 = 5.859, \quad (\nabla f^3)^T = [-2.9101 \quad -0.2761],$$

$$g^3 = -0.01954, \quad (\nabla g^3)^T = [0.2132 \quad 0.75]$$

$$\nabla L(\mathbf{x}^2, \lambda^3) = \begin{bmatrix} -5.102 \\ -2.124 \end{bmatrix} - (0.1205) \begin{bmatrix} 0.493 \\ 0.75 \end{bmatrix} = \begin{bmatrix} -5.161 \\ -2.214 \end{bmatrix}$$

$$\nabla L(\mathbf{x}^3, \lambda^3) = \begin{bmatrix} -2.910 \\ -0.2761 \end{bmatrix} - (0.1205) \begin{bmatrix} 0.2132 \\ 0.75 \end{bmatrix} = \begin{bmatrix} -2.936 \\ -0.3665 \end{bmatrix}$$

$$\gamma^2 = \nabla L(\mathbf{x}^3, \lambda^3) - \nabla L(\mathbf{x}^2, \lambda^3) = \begin{bmatrix} 2.225 \\ 1.8475 \end{bmatrix}$$

$$\Delta \mathbf{x}^2 = \begin{bmatrix} -0.3566 \\ -0.0109 \end{bmatrix} - \begin{bmatrix} -0.4965 \\ -0.8155 \end{bmatrix} = \begin{bmatrix} 0.1399 \\ 0.8046 \end{bmatrix}$$

Based on these vectors, the new Lagrangian Hessian is,

$$\nabla^2 L^3 = \begin{bmatrix} 13.3475 & 4.0939 \\ 4.0939 & 2.0403 \end{bmatrix} + \begin{bmatrix} 2.7537 & 2.2865 \\ 2.2865 & 1.8986 \end{bmatrix} - \begin{bmatrix} 10.6397 & 4.5647 \\ 4.5647 & 1.9584 \end{bmatrix}$$

$$\nabla^2 L^3 = \begin{bmatrix} 5.4616 & 1.8157 \\ 1.8157 & 1.9805 \end{bmatrix}$$

Our new approximation is,

$$f_a = 5.859 + [-2.910 \quad -0.2761] \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \Delta x_1 & \Delta x_2 \end{bmatrix} \begin{bmatrix} 5.4616 & 1.8157 \\ 1.8157 & 1.9805 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix}$$

$$g_a = -0.0195 + [0.2132 \quad 0.75] \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} \geq 0$$

The K-T equations, assuming the constraint is binding, are,

$$-2.910 + 5.4616\Delta x_1 + 1.8157\Delta x_2 - \lambda(0.2132) = 0$$

$$-0.2761 + 1.8157\Delta x_1 + 1.9805\Delta x_2 - \lambda(0.75) = 0$$

$$-0.0195 + 0.2132\Delta x_1 + 0.75\Delta x_2 = 0$$

The solution to this problem is, $\Delta x_1 = 0.6099$, $\Delta x_2 = -0.1474$, $\lambda = 0.7192$. Since λ is positive, our assumption about the constraint was correct. Our new proposed point is,

$$\mathbf{x}^4 = \mathbf{x}^3 + \Delta \mathbf{x} = \begin{bmatrix} -0.3566 \\ -0.0109 \end{bmatrix} + \begin{bmatrix} 0.6099 \\ -0.1474 \end{bmatrix} = \begin{bmatrix} 0.2533 \\ -0.1583 \end{bmatrix}$$

At this point the penalty function is 4.87, a decrease from 5.85, so we take the full step. The contour plot is given in Fig. 7.6

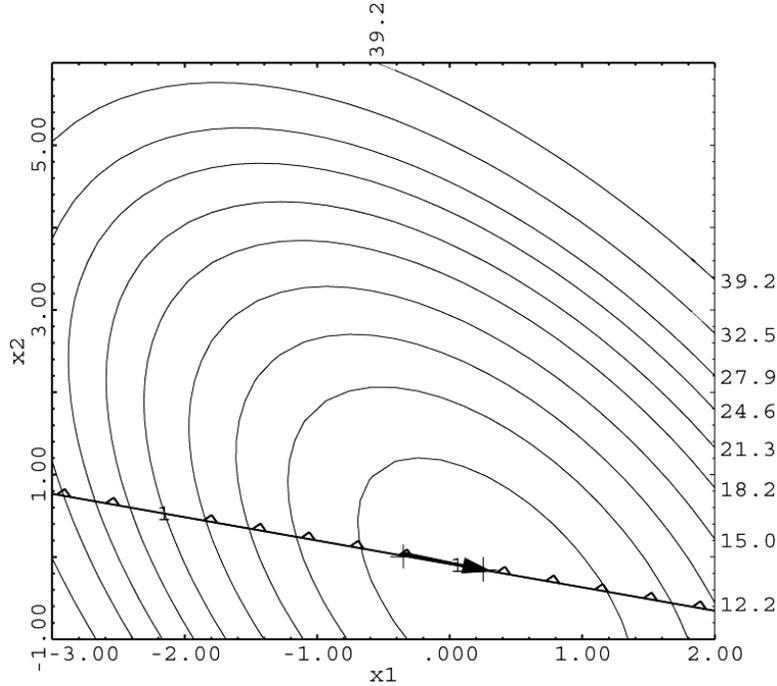


Fig. 7.6 The fourth approximation and step.

Iteration 5

At $(\mathbf{x}^4)^T = [0.2533 \quad -0.1583]$, $f^4 = 4.6071$, $(\nabla f^4)^T = [-1.268 \quad -0.4449]$,

$$g^4 = -0.3724, \quad (\nabla g^4)^T = [-1.007 \quad 0.75]$$

$$\nabla L(\mathbf{x}^3, \lambda^4) = \begin{bmatrix} -2.9101 \\ -0.2761 \end{bmatrix} - (0.7192) \begin{bmatrix} 0.2132 \\ 0.75 \end{bmatrix} = \begin{bmatrix} -3.063 \\ -0.8155 \end{bmatrix}$$

$$\nabla L(\mathbf{x}^4, \lambda^4) = \begin{bmatrix} -1.268 \\ -0.4449 \end{bmatrix} - (0.7192) \begin{bmatrix} -1.007 \\ 0.75 \end{bmatrix} = \begin{bmatrix} -0.5438 \\ -0.9843 \end{bmatrix}$$

$$\gamma^3 = \nabla L(\mathbf{x}^4, \lambda^4) - \nabla L(\mathbf{x}^3, \lambda^4) = \begin{bmatrix} 2.519 \\ -0.1688 \end{bmatrix}$$

$$\Delta \mathbf{x}^3 = \begin{bmatrix} 0.2533 \\ -0.1583 \end{bmatrix} - \begin{bmatrix} -0.3566 \\ -0.0109 \end{bmatrix} = \begin{bmatrix} 0.6099 \\ -0.1474 \end{bmatrix}$$

Based on these vectors, the new Lagrangian Hessian is,

$$\nabla^2 L^4 = \begin{bmatrix} 5.4616 & 1.8157 \\ 1.8157 & 1.9805 \end{bmatrix} + \begin{bmatrix} 4.0644 & -0.2724 \\ -0.2724 & 0.0183 \end{bmatrix} - \begin{bmatrix} 5.3681 & 1.4290 \\ 1.4290 & 0.3804 \end{bmatrix}$$

$$\nabla^2 L^4 = \begin{bmatrix} 4.1578 & 0.1144 \\ 0.1144 & 1.6184 \end{bmatrix}$$

Our new approximation is,

$$f_a = 4.6071 + \begin{bmatrix} -1.268 & -0.4449 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \Delta x_1 & \Delta x_2 \end{bmatrix} \begin{bmatrix} 4.1578 & 0.1144 \\ 0.1144 & 1.6184 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix}$$

$$g_a = -0.3724 + \begin{bmatrix} -1.007 & 0.75 \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} \geq 0$$

The K-T equations, assuming the constraint is binding, are,

$$\begin{aligned} -1.268 + 4.1578\Delta x_1 + 0.1144\Delta x_2 - \lambda(-1.007) &= 0 \\ -0.4449 + 0.1144\Delta x_1 + 1.6184\Delta x_2 - \lambda(0.75) &= 0 \\ -0.3724 - 1.007\Delta x_1 + 0.75\Delta x_2 &= 0 \end{aligned}$$

The solution to this problem is, $\Delta x_1 = 0.0988$, $\Delta x_2 = 0.6292$, $\lambda = 0.7797$. Since λ is positive, our assumption about the constraint was correct. Our new proposed point is,

$$\mathbf{x}^5 = \mathbf{x}^4 + \Delta \mathbf{x} = \begin{bmatrix} 0.2533 \\ -0.1583 \end{bmatrix} + \begin{bmatrix} 0.0988 \\ 0.6292 \end{bmatrix} = \begin{bmatrix} 0.3521 \\ 0.4709 \end{bmatrix}$$

At this point the penalty function is 4.55, a decrease from 4.87, so we take the full step. The contour plot is given in Fig. 7.7

We would continue in this fashion until $\Delta \mathbf{x}$ goes to zero. We would then know the original K-T equations were satisfied. The solution to this problem occurs at,

$$(\mathbf{x}^*)^T = [0.495 \quad 0.739], \quad f^* = 4.50$$

Using OptdesX, SQP requires 25 calls to the analysis program. GRG takes 50 calls.

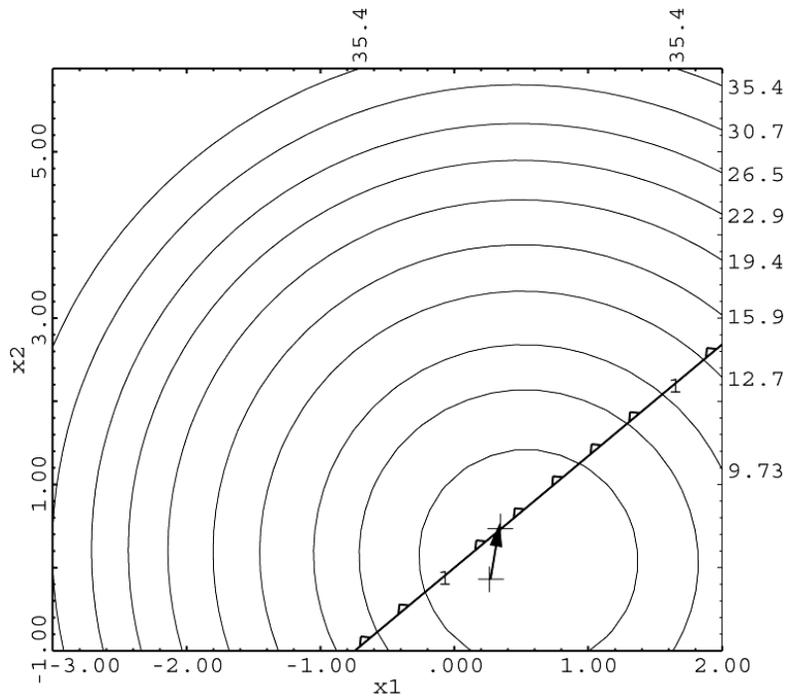


Fig. 7.7 The fifth approximation and step.

A summary of all the steps is overlaid on the original problem in Fig. 7.8.

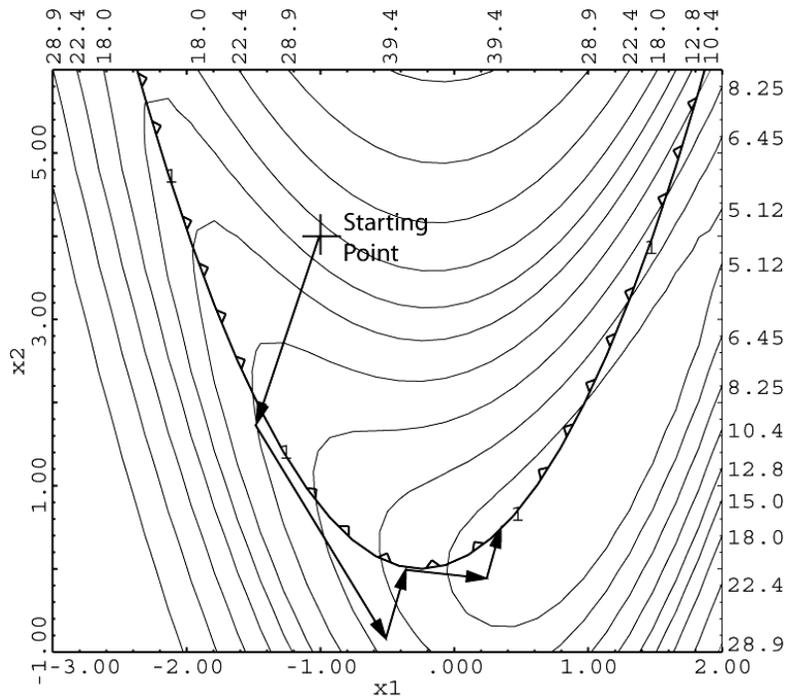


Fig. 7.8 The path of the SQP algorithm.

3 The Generalized Reduced Gradient (GRG) Algorithm

3.1 Introduction

In the previous section we learned that SQP works by solving for the point where the K-T equations are satisfied. SQP gradually enforces feasibility of the constraints as part of the K-T equations. In this section we will learn how GRG works. We will find it is very different from SQP. If started inside feasible space, GRG goes downhill until it runs into fences—constraints—and then corrects the search direction such that it follows the fences downhill. At every step it enforces feasibility. The strategy of GRG in following fences works well for engineering problems because most engineering optimums are constrained.

3.2 Explicit vs. Implicit Elimination

Suppose we have the following optimization problem,

$$\text{Min } f(\mathbf{x}) = x_1^2 + 3x_2^2 \quad (7.41)$$

$$\text{s.t. } g(\mathbf{x}) = 2x_1 + x_2 - 6 = 0 \quad (7.42)$$

A contour plot is given in Fig. 7.9a.

From previous discussions about modeling in Chapter 2, we know there are two approaches to this problem—we can solve it as a problem in two variables with one equality constraint, or we can use the equality constraint to eliminate a variable and the constraint. We will use the second approach. Using (7.42) to solve for x_2 ,

$$x_2 = 6 - 2x_1$$

Substituting into the objective function, (7.41), we have,

$$\text{Min } f(\mathbf{x}) = x_1^2 + 3(6 - 2x_1)^2 \quad (7.43)$$

Mathematically, solving the problem given by (7.41-7.42) is the same as solving the problem in (7.43). We have used the constraint to *explicitly* eliminate a variable and a constraint.

Once we solve for the optimal value of x_1 , we will obviously have to back substitute to get the value of x_2 using 7.42. The solution in x_1 is illustrated in Fig. 7.9b, where the sensitivity plot for 7.43 is given (because we only have one variable, we can't show a contour plot). The derivative $\frac{df}{dx_1}$ of (7.43) would be considered to be the *reduced gradient* relative to the original problem.

Usually we cannot make an explicit substitution as we did in this example. So we eliminate variables *implicitly*. We show how this can be done in the next section.

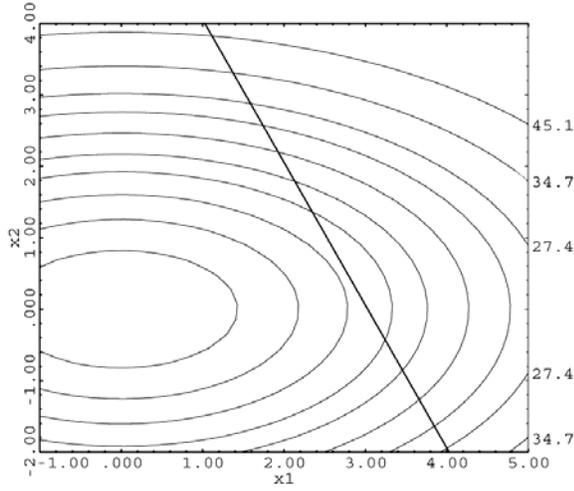


Fig. 7.9 a) Contour plot in x_1, x_2 with equality constraint. The optimum is at $\mathbf{x}^T = [2.7693 \quad 0.4613]$.

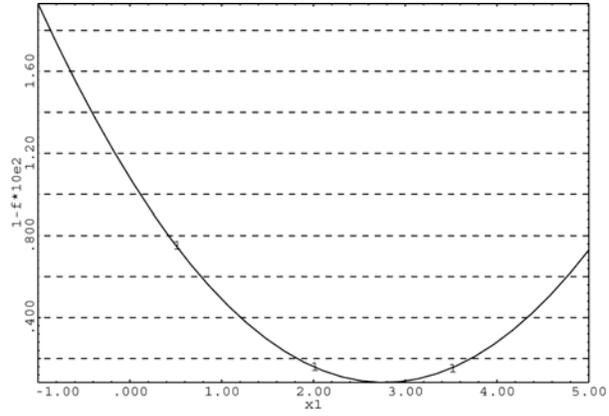


Fig. 7.9 b) Sensitivity plot for 7.43. The optimum is at $x_1 = 2.7693$

3.3 Implicit Elimination

In this section we will look at how we can eliminate variables implicitly. We do this by considering *differential* changes in the objective and constraints. We will start by considering a simple problem of two variables with one equality constraint,

$$\text{Min } f(\mathbf{x}) \quad \mathbf{x}^T = [x_1 \quad x_2] \tag{7.44}$$

$$\text{s.t. } g(\mathbf{x}) - b = 0 \tag{7.45}$$

Suppose we are at a feasible point. Thus the equality constraint is satisfied. We wish to move to improve the objective function. The differential change is given by,

$$df = \frac{\partial f}{\partial x_1} dx_1 + \frac{\partial f}{\partial x_2} dx_2 \tag{7.46}$$

to keep the constraint satisfied the differential change must be zero:

$$dg = \frac{\partial g}{\partial x_1} dx_1 + \frac{\partial g}{\partial x_2} dx_2 = 0 \tag{7.47}$$

Solving for dx_2 in (7.47) gives:

$$dx_2 = \frac{-\partial g / \partial x_1}{\partial g / \partial x_2} dx_1$$

substituting into (7.46) gives,

$$df = \left[\frac{\partial f}{\partial x_1} - \frac{\partial f}{\partial x_2} \left(\frac{\partial g / \partial x_1}{\partial g / \partial x_2} \right) \right] dx_1 \quad (7.48)$$

where the term in brackets is the reduced gradient.

$$\text{i.e.,} \quad \frac{df_R}{dx_1} = \left[\frac{\partial f}{\partial x_1} - \frac{\partial f}{\partial x_2} \left(\frac{\partial g / \partial x_1}{\partial g / \partial x_2} \right) \right] \quad (7.49)$$

If we substitute Δx for dx , then the equations are only approximate. *We are stepping tangent to the constraint in a direction that improves the objective function.*

3.4 GRG Algorithm with Equality Constraints Only

We can extend the concepts of the previous section to the general problem which we represent in vector notation. Suppose now we consider the general problem with equality constraints,

$$\begin{aligned} \text{Min} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & g_i(\mathbf{x}) - b_i = 0 \quad i = 1, \dots, m \end{aligned}$$

We have n design variables and m equality constraints. We begin by partitioning the design variables into $(n-m)$ independent variables, \mathbf{z} , and m dependent variables \mathbf{y} . The independent variables will be used to improve the objective function, and the dependent variables will be used to satisfy the binding constraints. If we partition the gradient vectors as well we have,

$$\begin{aligned} \nabla f(\mathbf{z})^T &= \left[\frac{\partial f(\mathbf{x})}{\partial z_1} \quad \frac{\partial f(\mathbf{x})}{\partial z_2} \quad \dots \quad \frac{\partial f(\mathbf{x})}{\partial z_{n-m}} \right] \\ \nabla f(\mathbf{y})^T &= \left[\frac{\partial f(\mathbf{x})}{\partial y_1} \quad \frac{\partial f(\mathbf{x})}{\partial y_1} \quad \dots \quad \frac{\partial f(\mathbf{x})}{\partial y_m} \right] \end{aligned}$$

We will also define independent and dependent matrices of the partial derivatives of the constraints:

$$\frac{\partial \psi}{\partial \mathbf{z}} = \begin{bmatrix} \frac{\partial g_1}{\partial z_1} & \frac{\partial g_1}{\partial z_2} & \dots & \frac{\partial g_1}{\partial z_{n-m}} \\ \frac{\partial g_m}{\partial z_1} & \frac{\partial g_m}{\partial z_2} & \dots & \frac{\partial g_m}{\partial z_{n-m}} \end{bmatrix} \quad \frac{\partial \psi}{\partial \mathbf{y}} = \begin{bmatrix} \frac{\partial g_1}{\partial y_1} & \frac{\partial g_1}{\partial y_2} & \dots & \frac{\partial g_m}{\partial y_m} \\ \frac{\partial g_m}{\partial y_1} & \frac{\partial g_m}{\partial y_2} & \dots & \frac{\partial g_m}{\partial y_m} \end{bmatrix}$$

We can write the differential changes in the objective and constraints in vector form as:

$$df = f(\mathbf{z})^\top d\mathbf{z} + f(\mathbf{y})^\top d\mathbf{y} \quad (7.50)$$

$$d\psi = \frac{\partial\psi}{\partial\mathbf{z}} d\mathbf{z} + \frac{\partial\psi}{\partial\mathbf{y}} d\mathbf{y} = \mathbf{0} \quad (7.51)$$

Noting that $\frac{\partial\psi}{\partial\mathbf{y}}$ is a square matrix, and solving (7.51) for $d\mathbf{y}$,

$$d\mathbf{y} = -\frac{\partial\psi^{-1}}{\partial\mathbf{y}} \frac{\partial\psi}{\partial\mathbf{z}} d\mathbf{z} \quad (7.52)$$

substituting (7.52) into (7.50),

$$df = \nabla f(\mathbf{z})^\top d\mathbf{z} - \nabla f(\mathbf{y})^\top \frac{\partial\psi^{-1}}{\partial\mathbf{y}} \frac{\partial\psi}{\partial\mathbf{z}} d\mathbf{z}$$

or
$$\nabla f_R^\top = \nabla f(\mathbf{z})^\top - \nabla f(\mathbf{y})^\top \frac{\partial\psi^{-1}}{\partial\mathbf{y}} \frac{\partial\psi}{\partial\mathbf{z}} \quad (7.53)$$

where ∇f_R^\top is the reduced gradient. *The reduced gradient is the direction of steepest ascent that stays tangent to the binding constraints.*

3.5 GRG Example 1: One Equality Constraint

We will illustrate the theory of the previous section with the following example. For this example we will have three variables and one equality constraint. We state the problem as,

$$\text{Min } f = 4x_1^2 + x_2^2 + 3x_3^2$$

$$\text{s.t. } g = 2x_1 + 4x_2 - x_3 = 10$$

Step 1: Evaluate the objective and constraints at the starting point.

The starting point will be $\mathbf{x}^T = [2 \quad 2 \quad 2]$, at which point $f = 32$ and $g = 10$. So the constraint is satisfied.

Step 2: Partition the variables.

We have one binding constraint so we will need one dependent variable. We will arbitrarily choose x_1 as the dependent variable, so $\mathbf{y} = [x_1]$. The independent variables will therefore be $\mathbf{z}^T = [x_2 \quad x_3]$. Thus,

$$\mathbf{z} = \begin{bmatrix} x_2 \\ x_3 \end{bmatrix} \quad \mathbf{y} = [x_1] \quad \nabla f(\mathbf{z}) = \begin{bmatrix} \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \end{bmatrix} = \begin{bmatrix} 2x_2 \\ 6x_3 \end{bmatrix}_{@2,2} = \begin{bmatrix} 4 \\ 12 \end{bmatrix} \quad \nabla f(\mathbf{y}) = \left[\frac{\partial f}{\partial x_1} \right] = [8x_1]_{@2} = [16]$$

$$\frac{\partial \psi}{\partial \mathbf{z}} = \begin{bmatrix} \frac{\partial g}{\partial x_2} & \frac{\partial g}{\partial x_3} \end{bmatrix} = [4 \quad -1] \quad \frac{\partial \psi}{\partial \mathbf{y}} = \left[\frac{\partial g}{\partial x_1} \right] = [2]$$

Step 3: Compute the reduced gradient.

We now have the information we need to compute the reduced gradient:

$$\begin{aligned} \nabla f_R^T &= \nabla f(\mathbf{z})^T - \nabla f(\mathbf{y})^T \frac{\partial \psi^{-1}}{\partial \mathbf{y}} \frac{\partial \psi}{\partial \mathbf{z}} \\ \nabla f_R^T &= [4 \quad 12] - [16] \begin{bmatrix} 1 \\ 2 \end{bmatrix} [4 \quad -1] \\ &= [-28 \quad 20] \end{aligned}$$

Step 4: Compute the direction of search.

We will step in the direction of steepest descent, i.e., the negative reduced gradient direction, which is the direction of steepest descent which stays tangent to the constraint.

$$\mathbf{s} = \begin{bmatrix} 28 \\ -20 \end{bmatrix} \quad \text{or, normalized, } \mathbf{s} = \begin{bmatrix} 0.8137 \\ -0.5812 \end{bmatrix}$$

Step 5: Do a line search in the independent variables

We will use our regular formula,

$$\mathbf{z}^{new} = \mathbf{z}^{old} + \alpha \mathbf{s}$$

We will arbitrarily pick a starting step length $\alpha = 0.5$

$$\begin{bmatrix} x_2^{new} \\ x_3^{new} \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} + 0.5 \begin{bmatrix} 0.8137 \\ -0.5812 \end{bmatrix} = \begin{bmatrix} 2.4068 \\ 1.7094 \end{bmatrix}$$

Step 6: Solve for the value of the dependent variable.

We do this using (7.52) above, only we will substitute $\Delta \mathbf{y}$ for $d\mathbf{y}$:

$$\Delta \mathbf{y} = - \frac{\partial \psi^{-1}}{\partial \mathbf{y}} \frac{\partial \psi}{\partial \mathbf{z}} \Delta \mathbf{z}$$

$$\begin{aligned} [\Delta x_1] &= -\frac{\partial \psi^{-1}}{\partial \mathbf{y}} \frac{\partial \psi}{\partial \mathbf{z}} \begin{bmatrix} \Delta x_2 \\ \Delta x_3 \end{bmatrix} \\ &= -\begin{bmatrix} 1 \\ 2 \end{bmatrix} \begin{bmatrix} 4 & -1 \end{bmatrix} \begin{bmatrix} 0.4069 \\ -0.2906 \end{bmatrix} \\ &= -0.9590 \end{aligned}$$

So the new value of x_1 is,

$$\begin{aligned} x_1^{new} &= x_1^{old} + \Delta x \\ &= 2 - 0.9590 \\ &= 1.041 \end{aligned}$$

Our new point is $\mathbf{x}^T = [1.041 \quad 2.4069 \quad 1.7094]$ at which point $f = 18.9$ and $g = 10$. We observe that the objective has decreased from 32 to 18.9 and the constraint is still satisfied. This only represents one step in the line search. We would continue the line search until we reach a minimum.

3.6 GRG Algorithm with Equality and Inequality Constraints

In this section we will consider the general problem with both inequality and equality constraints,

$$\begin{aligned} \text{Min} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & g_i(\mathbf{x}) - b_i \leq 0 \quad i = 1, \dots, k \\ & g_i(\mathbf{x}) - b_i = 0 \quad i = k + 1, \dots, m \end{aligned}$$

The extension of the GRG algorithm to include inequality constraints involves some additional complexity, because the derivation of GRG is based on equality constraints. We therefore convert inequalities into equalities by adding *slack variables*. So for example,

$$x_1 + 5x_2 \leq 6 \text{ is changed to } x_1 + 5x_2 + s_1 = 6$$

where s_1 is the slack variable and must be positive for the constraint to be feasible. The slack variable is zero when the constraint is binding. The word “slack” comes from the idea the variable “takes up the slack” between the function value and the right hand side.

The GRG algorithm described here is an *active constraint* algorithm—only the binding inequality constraints are used to determine the search direction. The non-binding constraints enter into the problem only if they become binding or violated.

With these changes the equations of Section 3.4 can be used. In particular, (7.53) is used to compute the reduced gradient.

3.7 Steps of the GRG Algorithm for the General Problem

1. Evaluate the objective function and all constraints at the current point.
2. For any binding inequality constraints, add a slack variable, s_i
3. Partition the variables into independent variables and dependent variables. We will need one dependent variable for each binding constraint. Any variable at either its upper or lower limit should become an independent variable.
4. Compute the reduced gradient using (7.53).
5. Calculate a direction of search. We can use any method to calculate the search direction that relies on gradients since the reduced gradient is a gradient. For example, we can use a quasi-Newton update.
6. Do a line search in the independent variables. For each step, find the corresponding values in the dependent variables using (7.52) with $\Delta \mathbf{z}$ and $\Delta \mathbf{y}$ substituted for $d\mathbf{z}$ and $d\mathbf{y}$.
7. At each step in the line search, drive back to the constraint boundaries for any violated constraints using Newton-Raphson to adjust the dependent variables. If an independent variable hits its bound, set it equal to its bound.

The N-R iteration is given by $\Delta \mathbf{y} = -\frac{\partial \psi^{-1}}{\partial \mathbf{y}}(\mathbf{g} - \mathbf{b})$ We note we already have the matrix

$\frac{\partial \psi^{-1}}{\partial \mathbf{y}}$ from the calculation of the reduced gradient.

8. The line search may terminate either of 4 ways
 - 1) The minimum in the direction of search is found (using, for example, quadratic interpolation).
 - 2) A dependent variable hits its upper or lower limit.
 - 3) A formerly non-binding constraint becomes binding.
 - 4) N-R fails to converge. In this case we must cut back the step size until N-R does converge.
9. If at any point the reduced gradient in step 4 is equal to $\mathbf{0}$, the K-T conditions are satisfied.

3.8 GRG Example 2: Two Inequality Constraints

In this problem we have two inequality constraints and will therefore need to add in slack variables.

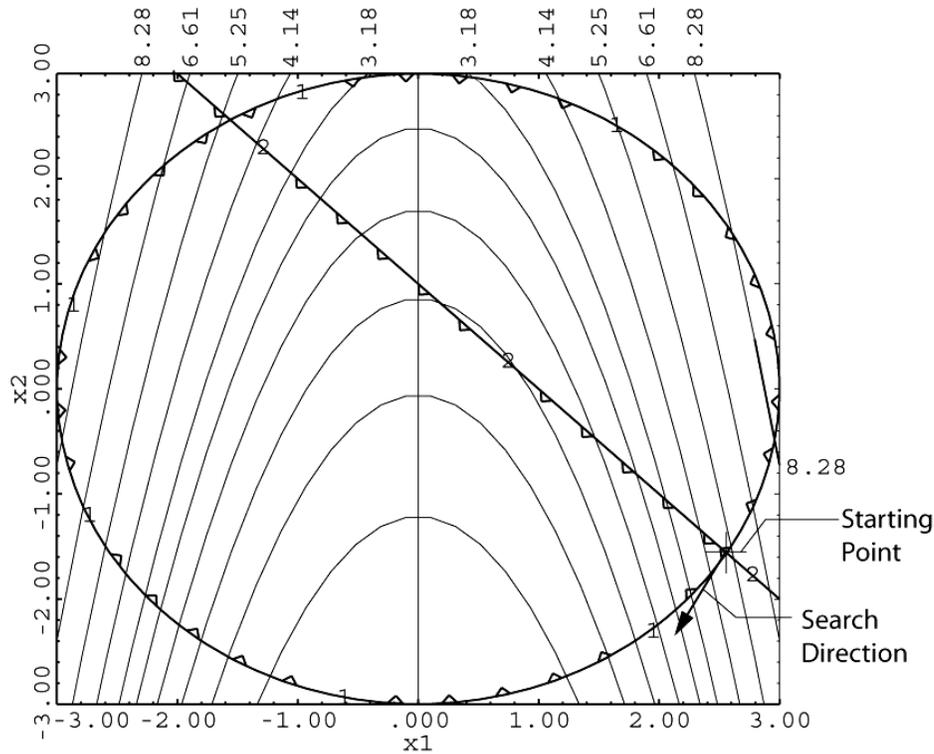


Fig. 7.10 Example problem for GRG algorithm

$$\begin{aligned} \text{Min. } & f(\mathbf{x}) = x_1^2 + x_2 \\ \text{s.t.: } & g_1(\mathbf{x}) = x_1^2 + x_2^2 - 9 \leq 0 \\ & g_2(\mathbf{x}) = x_1 + x_2 - 1 \leq 0 \end{aligned}$$

Suppose, to make things interesting, we are starting at $\mathbf{x}^T = [2.56155, -1.56155]$ where both constraints are binding.

Step 1: Evaluate functions.

$$f(\mathbf{x}) = 5.0 \quad g_1(\mathbf{x}) = 0.0 \quad g_2(\mathbf{x}) = 0.0$$

Step 2: Add in slack variables.

We note that both constraints are binding so we will add in two slack variables. s_1, s_2 .

Step 3: Partition the variables

Since the slack variables are at their lower limits ($=0$) they will become the independent variables; x_1, x_2 will be the dependent variables.

$$\mathbf{z}^T = [s_1 \quad s_2] \quad \mathbf{y}^T = [x_1 \quad x_2]$$

Step 4: Compute the reduced gradient

$$\nabla f(\mathbf{z})^T = [0.0 \quad 0.0] \quad \nabla f(\mathbf{y})^T = [5.123 \quad 1.0]$$

$$\frac{\partial \psi}{\partial \mathbf{z}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \frac{\partial \psi}{\partial \mathbf{y}} = \begin{bmatrix} 5.123 & -3.123 \\ 1.0 & 1.0 \end{bmatrix}$$

$$\frac{\partial \psi^{-1}}{\partial \mathbf{y}} = \begin{bmatrix} 0.1213 & 0.3787 \\ -0.1213 & 0.6213 \end{bmatrix}$$

thus
$$\frac{\partial \psi^{-1}}{\partial \mathbf{y}} \frac{\partial \psi}{\partial \mathbf{z}} = \begin{bmatrix} 0.1213 & 0.3787 \\ -0.1213 & 0.6213 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.1213 & 0.3787 \\ -0.1213 & 0.6213 \end{bmatrix}$$

$$\begin{aligned} \nabla f_r^T &= [0.0 \quad 0.0] - [5.123 \quad 1] \begin{bmatrix} 0.1213 & 0.3787 \\ -0.1213 & 0.6213 \end{bmatrix} \\ &= [0.0 \quad 0.0] - [0.50 \quad 2.56] \\ &= [-0.50 \quad -2.56] \end{aligned}$$

Step 5: Calculate a search direction.

We want to move in the negative gradient direction, so our search direction will be $\mathbf{s}^T = [0.50 \quad 2.56]$. This is the direction for the independent variables (the slacks). When normalized this direction is $\mathbf{s}^T = [0.19 \quad 0.98]$.

Step 6: Conduct the line search in the independent variables

We will start our line search, denoting the current point as \mathbf{z}^0 ,

$$\mathbf{z}^1 = \mathbf{z}^0 + \alpha \mathbf{s}^0$$

Suppose we pick $\alpha = 1.0$. Then

$$\begin{aligned} \mathbf{z}^1 &= \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix} + (1.0) \begin{bmatrix} 0.19 \\ 0.98 \end{bmatrix} \\ \mathbf{z}^1 &= \begin{bmatrix} 0.19 \\ 0.98 \end{bmatrix} \end{aligned}$$

Step 7: Adjust the dependent variables

To find the change in the dependent variables, we use (7.52)

$$\Delta \mathbf{y} = \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = - \begin{bmatrix} \frac{\partial \psi^{-1}}{\partial \mathbf{y}} \end{bmatrix} \begin{bmatrix} \frac{\partial \psi}{\partial \mathbf{z}} \end{bmatrix} \Delta \mathbf{z}$$

$$= \begin{bmatrix} 0.1213 & 0.3787 \\ -0.1213 & 0.6213 \end{bmatrix} \begin{bmatrix} 0.19 \\ 0.98 \end{bmatrix}$$

$$= \begin{bmatrix} -0.394 \\ -0.586 \end{bmatrix}$$

$$x_1^1 = 2.56155 - 0.394 = 2.168$$

$$x_2^1 = -1.56155 - 0.586 = -2.148$$

at which point $f(\mathbf{x}) = 2.522$

Have we violated any constraints?

$$g_1(\mathbf{x}) = x_1^2 + x_2^2 - 9 = (2.168)^2 + (-2.148)^2 - 9 = 0.31 \text{ (violated)}$$

$$g_2(\mathbf{x}) = x_1 + x_2 - 1 = 2.168 - 2.148 - 1 = -0.98 \text{ (satisfied)}$$

We need to drive back to where the violated constraint is satisfied. We will use N-R to do this. Since we don't want to drive back to where both constraints are binding, we will set the residual for constraint 2 to zero.

N-R Iteration 1:

$$\mathbf{y}^{(n)} = \mathbf{y}^{(0)} - \frac{\partial \psi^{-1}}{\partial \mathbf{y}} (\mathbf{g} - \mathbf{b})$$

$$= \begin{bmatrix} 2.168 \\ -2.148 \end{bmatrix} - \begin{bmatrix} 0.1213 & 0.3787 \\ -0.1213 & 0.6213 \end{bmatrix} \begin{bmatrix} 0.31 \\ 0.0 \end{bmatrix}$$

$$= \begin{bmatrix} 2.130 \\ -2.110 \end{bmatrix}$$

at this point

$$g_1 = (2.130)^2 + (2.110)^2 - 9 = -0.011$$

$$g_2 = -0.98$$

N-R Iteration 2:

$$= \begin{bmatrix} 2.130 \\ -2.110 \end{bmatrix} - \begin{bmatrix} 0.1213 & 0.3787 \\ -0.1213 & 0.6213 \end{bmatrix} \begin{bmatrix} -0.011 \\ 0.0 \end{bmatrix}$$

$$= \begin{bmatrix} 2.1313 \\ -2.113 \end{bmatrix}$$

evaluating constraints:

$$g_1 = (2.1313)^2 + (-2.113)^2 - 9 = 0$$

$$g_2 = -0.98$$

We are now feasible again. We have taken one step in the line search!

Our new point is $\mathbf{x} = \begin{bmatrix} 2.1313 \\ -2.113 \end{bmatrix}$ at which point the objective is 2.43, and all constraints are satisfied.

We would continue the line search until we run into a new constraint boundary, a dependent variable hits a bound, or we can no longer get back on the constraint boundaries (which is not an issue in this example, since the constraint is linear).

4 References

M. J. D. Powell, "Algorithms for Nonlinear Constraints That Use Lagrangian Functions," *Mathematical Programming*, 14, (1978) 224-248.

Powell, M.J.D., "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations," *Numerical Analysis, Dundee 1977*, Lecture Notes in Mathematics no. 630, Springer-Verlag, New York, 1978.

L.S. Lasdon et. al, "Design and Testing of a Generalized Reduced Gradient Code for Nonlinear Programming," *ACM Transactions on Mathematical Software*, Vol. 4 No. 1 March 1978

Gabriele G.A., and K.M. Ragsdell, "The Generalized Reduced Gradient Method: A Reliable Tool for Optimal Design," *Trans ASME, J. Eng. Industry*, May 1977.

CHAPTER 8

ROBUST DESIGN

1 Introduction

In the “real” world, almost all designs are subject to variation. Such variation can arise from multiple sources, including manufacturing processes, material properties, changing operating conditions, or the environment. We can also have uncertainty associated with our computer model. We may not know some assumed values as well as we would like (e.g. heat transfer coefficients, friction coefficients), and our assumptions about boundary conditions might also be faulty. For example, loads or temperatures might be different than we assumed.

The consequences of variation are almost always bad. Variation in product dimensions can lead to assemblies which assemble poorly or not at all, or function improperly. Failure to take into account variation can lead to product failure, poor performance and customer dissatisfaction. A famous quality researcher, Genichi Taguchi, has promoted the idea that any deviation from a desired target value results in a loss to the customer.

Optimized designs may be particularly vulnerable to variation. This is because optimized designs often include active or binding constraints. Such constraints are on the verge of being violated. Slight variations in problem parameters can cause designs to become infeasible.

Thus it should be clear that we should not only be interested in an optimal design, but also in an optimal design which is *robust*. A robust design is a design which can tolerate variation. Fortunately, a general approach to robust design can be formulated in terms of optimization techniques, further extending the usefulness of these methods. In this chapter we will learn how to apply optimization methods to determine a robust design.

We will define variation in terms of tolerances which give upper and lower limits on the expected deviation of uncertain quantities about their nominal values. We consider a design to be robust if it can tolerate variability, within the ranges of the tolerances, and still function properly. The term “function properly” will be taken to mean the constraints remain feasible when subjected to variation. We define this type of robustness as *feasibility robustness*.

2 Worst-case Tolerances

2.1 Introduction

We will begin by considering *worst-case* tolerances. With a worst-case tolerance analysis, we assume all tolerances can simultaneously be at the values which cause the largest variation. We ignore the sign of the variation, assuming it always adds. This gives us a conservative estimate of the worst situation we should encounter.

2.2 Background

We will consider a design problem of the form,

$$\begin{aligned} \text{Min} \quad & f(\mathbf{x}, \mathbf{p}) \\ \text{s.t.} \quad & g_i(\mathbf{x}, \mathbf{p}) \leq b_i \quad i = 1, \dots, m \end{aligned}$$

where

\mathbf{x} is an n dimensional vector of design variables

\mathbf{p} is a l dimensional vector of constant *parameters*, i.e., unmapped analysis variables.

We will group the right-hand-side values, b_i , into a vector \mathbf{b} .

For a given set of nominal values for \mathbf{x} , \mathbf{p} , and \mathbf{b} , there can be fluctuations $\Delta\mathbf{x}$, $\Delta\mathbf{p}$, and $\Delta\mathbf{b}$ about these nominal values. We would like the design to be feasible even if these fluctuations occur. As we will see, in a constrained design space, *the effect of variation is to reduce the size of the feasible region*.

2.3 Two Approaches to Robust Optimal Design

Several researchers have incorporated worst-case tolerances into the design process, using a “tolerance box” approach, as illustrated in Fig. 8.1. A tolerance box is defined for the design variables; the robust optimum is the design that is as close to the nominal optimum as possible and keeps the entire box in the feasible region. A main drawback is that it does not allow us to specify tolerances on parameters.

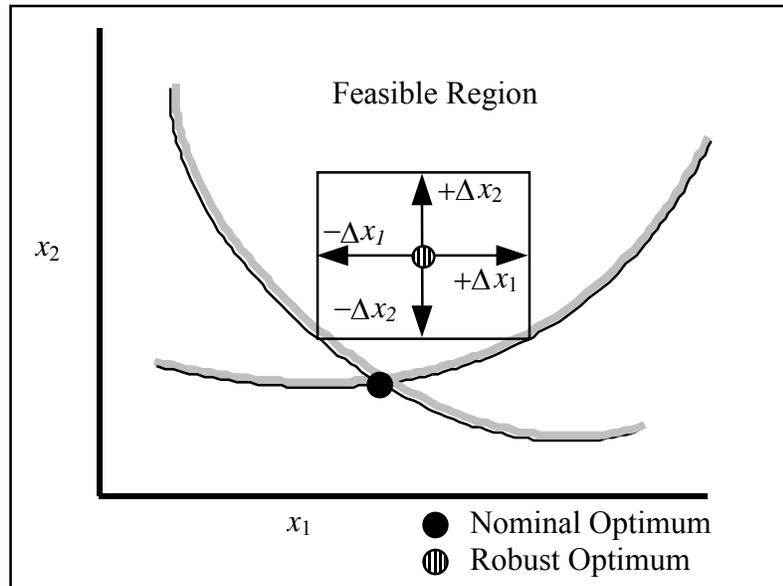


Fig 8.1. Tolerance box approach for robust design with worst-case tolerances.

In contrast to the tolerance box approach, the method we will develop relies on “transmitted variation.” As will be explained, we *transmit the variation from the variables and parameters to the constraints*, and then correct the nominal optimum so that it is feasible with respect to the constraints with the transmitted variation added in. This method is

illustrated in Fig. 8.2. The same optimization methods used to find the nominal optimum can be used to find the robust optimum, and tolerances may be placed on any model value, whether a variable or a parameter.

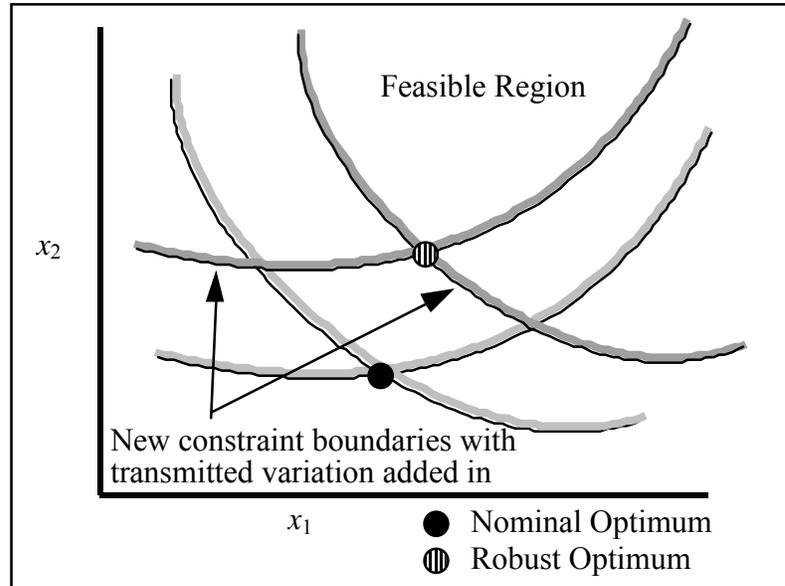


Fig 8.2. Transmitted variation approach for robust design with worst-case tolerances.

2.4 Calculating Transmitted Variation: Linear Analysis

Worst-case tolerance analysis assumes that all fluctuations may occur simultaneously in the worst possible combination. The effect of variations on a function can be estimated from a first order Taylor series, as follows:

$$\Delta g_i = \sum_{j=1}^n \left| \frac{\partial g_i}{\partial x_j} \Delta x_j \right| + \sum_{j=1}^m \left| \frac{\partial g_i}{\partial p_j} \Delta p_j \right| \quad (8.1)$$

where the bars indicate that the absolute value is taken. With the absolute value, (8.1) allows the tolerances to assume any sign and therefore computes the worst possible effect of the tolerances. We will refer to Δg_i as the “functional variation.” For constraints, we must also add in variation of the right hand side Δb_i ,

$$\Delta_i = \Delta g_i + \Delta b_i \quad (8.2)$$

We will refer to Δ_i as the “total constraint variation.”

2.5 Developing a Robust Optimal Design

2.5.1 Compensating for Variation

Robustness can be developed for worst-case tolerances by adjusting the value of the constraint functions by the amount of the total constraint variation during optimization. For a less than constraint, we add the variation; for a greater than constraint, we subtract the variation. In both cases, this has the effect of reducing the size of the feasible region, with a corresponding degradation in the value of the objective. Thus a less than constraint becomes,

$$g_i + \Delta_i \leq b_i \quad (8.3)$$

A greater than constraint becomes,

$$g_i - \Delta_i \geq b_i \quad (8.4)$$

Alternatively, we can consider that the variation has reduced (or increased) the right side, depending on whether we have a less than or greater than constraint, respectively:

$$g_i \leq b_i - \Delta_i \quad (8.5)$$

$$g_i \geq b_i + \Delta_i \quad (8.6)$$

2.5.2 An Efficient Solution Method

Adding in the transmitted variation can be computationally expensive because the transmitted variation is a function of derivatives, and these would have to be evaluated every time the constraint is evaluated.

To reduce computation, we propose the following process,

1. Drive to the nominal optimum.
2. Calculate the transmitted variation.
3. Adjust the constraint right hand sides by the amount of transmitted variation.
4. Assuming the transmitted variation is a constant, re-optimize to find the robust optimum.

The assumptions that are built into this method are,

- the robust optimum is close to the nominal optimum.
- the derivatives are constant, i.e. second derivatives are equal to zero.

These assumptions are consistent with assuming a linear Taylor expansion, (8.1), for the transmitted variation.

2.5.3 An Example: The Two-bar Truss

We will illustrate the method on our familiar example, the Two-bar Truss. Now, however, we will add in tolerances on all the analysis variables and the right hand side for stress. (The right hand side for stress is the yield strength, a material property, and so has variation associated with it. The other right hand sides are set by the user and are not uncertain.) Data regarding the truss are given in Table 1.

Table 1 Worst-case Tolerance Data for the Two-bar Truss

Description	Nominal Value	Worst-case Tolerance
Height, H	Design Variable	0.5 in
Width, B	60 in	0.5 in
Diameter, d	Design Variable	0.1 in
Thickness, t	0.15 in	0.01 in
Modulus	30000 ksi	1500 ksi
Density	0.3 lb/in ³	0.01 lb/in ³
Load, P	66 kips	3 kips
Yield Strength <i>Right Hand Side</i>	100 ksi	5 ksi
Buckling <i>Right Hand Side</i>	0.0	0.0
Deflection <i>Right Hand Side</i>	0.25 in	0.0

Fig. 8.3 is a contour plot showing the design space for this problem. As a first step, we drive to the nominal optimum to the problem, which occurs at the intersection of the boundaries for stress and deflection, shown as a solid circle in the figure.

We then calculate the transmitted variation, given by Eq (8.1), using derivatives that are already available from the nominal optimization. If we calculate the worst-case variation for each constraint using (8.3), and subtract this value from the constraint right hand sides, as in (8.5), the new constraint boundaries are shown as 1*, 2*, and 3* in the figure. The decrease in the feasible region caused by including variation is shaded.

The final step is to drive to the robust optimum, given by the shaded circle in the figure. The optimal value of the objective has increased from 15.8 to 18.0 pounds.

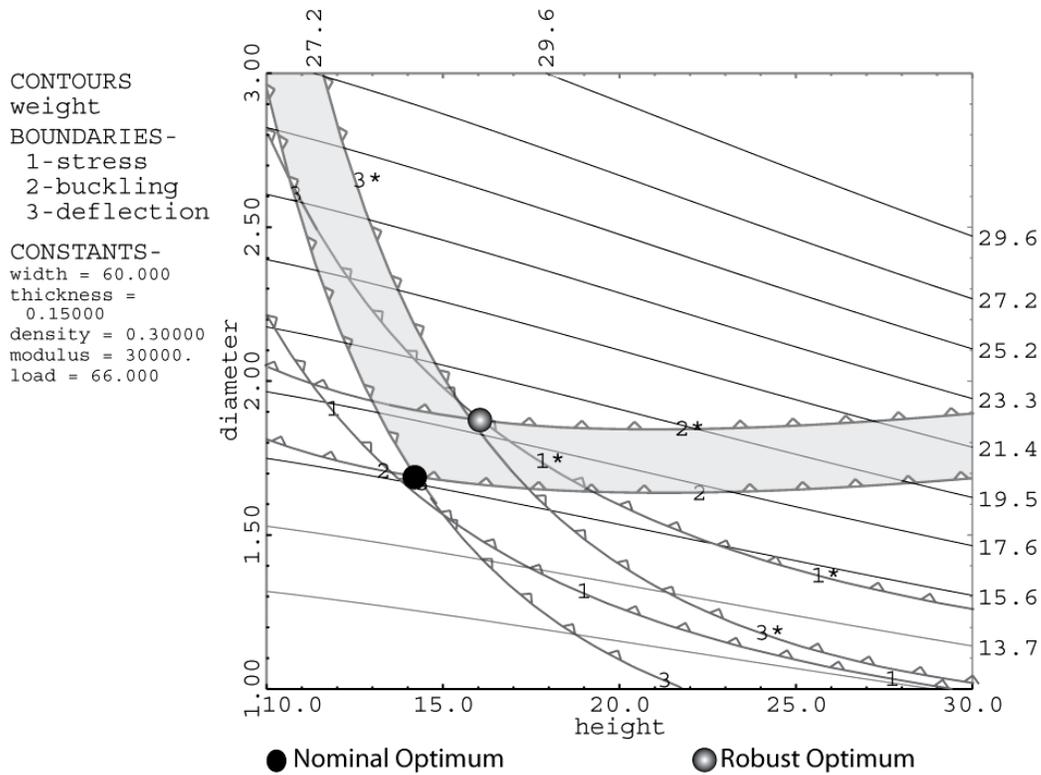


Fig. 8.3 Decrease in feasible region caused by including worst-case tolerances.

2.5.4 A Numerical Example: The Two-bar Truss

To illustrate how this method might be implemented “by hand,” we will determine the effect on the optimum of the Two-bar Truss problem caused by just two tolerances. For the nominal optimization we have design variables height, diameter, and thickness. We wish to see the effect on the optimum of adding tolerances on the load and the width,

$$\begin{aligned}\Delta_{\text{load}} &= 2 \text{ kips} \\ \Delta_{\text{width}} &= 1 \text{ inch}\end{aligned}$$

The first step is to drive to the nominal optimum. The optimum occurs with height = 30, diameter = 2.204, thickness = 0.067, and an optimal weight of 11.88 lbs, with stress and buckling as binding constraints. Next we obtain the derivatives at the optimum using the Gradients window (note that we evaluate un-scaled gradients of all functions with respect to all variables):

	weight	stress	stress-buckling	deflection
height	0.1980491	-1.666198	1.667515	-0.003332284
diameter	5.390443	-45.35029	-136.0059	-0.09070057
thickness	176.2924	-1482.736	-1485.508	-2.965472
width	0.09902378	0.8331339	2.500016	0.004998807
density	39.60950	0.000000	0.000000	0.000000
modulus	0.000000	0.000000	-0.003333768	-6.665068e-06
load	0.000000	1.514788	1.514788	0.003029576

Dismiss Hardcopy

The next step is to calculate the transmitted variation to the stress, buckling, and deflection constraints.

$$\Delta_{stress} = |(0.833) \cdot (1)| + |(1.514) \cdot (2)| = 3.861$$

$$\Delta_{buckling} = |(2.500) \cdot (1)| + |(1.514) \cdot (2)| = 5.528$$

$$\Delta_{deflection} = |(0.005) \cdot (1)| + |(0.00303) \cdot (2)| = 0.01106$$

The third step requires that the constraint right hand sides be adjusted. For this problem,

$$\text{stress} \leq 100 - 3.861 = 96.139 \text{ ksi}$$

$$\text{buckling} \leq 0 - 5.528 = -5.528$$

$$\text{deflection} \leq 0.25 - 0.01106 = 0.23894 \text{ in}$$

When we re-optimize in accordance with these new constraints, the new optimum is, height = 29.95, diameter = 2.22, thickness = 0.070, with a weight of 12.36 pounds, and with stress and buckling, again, as binding constraints.

2.6 Verifying the Robust Design: Monte Carlo Simulation

We have discussed a method to develop a robust design. How can we tell if the design is really robust, i.e., how can we be sure that any design within the tolerance bounds will remain feasible? One approach is Monte Carlo simulation, which refers to using a computer to simulate variation in a design. The computer introduces variation within the bounds of the tolerances for each variable. It then calculates the functions. We do this many times--in effect, we have the computer build thousands of designs--and we keep track of how many designs violate the constraints. For a worst-case analysis, the number of infeasible designs should be zero.

Since these are worst-case tolerances, we will assign load and width to have *uniform distributions*. For a uniform distribution, the ends of the range have the same probability of

occurring as the middle. So, for example, the load would be assumed to be uniformly distributed with a lower bound of 64 and an upper bound of 68.

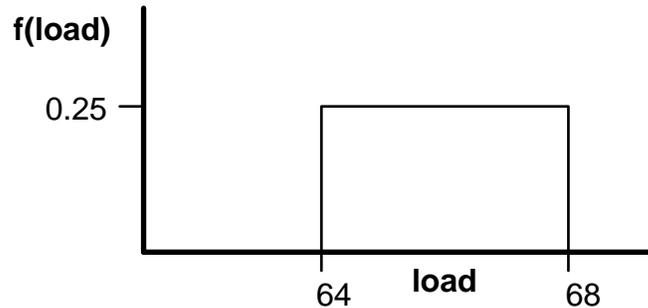


Fig. 8.5 Uniform distribution for load.

The width has a similar distribution, only the lower bound is 59 and the upper bound is 61. We then have the computer generate many designs where load and width are uniformly distributed between their tolerances, and we count the infeasible designs.

The output from running a Monte Carlo simulation (100,000 trials) on the robust design is shown below,

```
no. of trials= 100000
mean values, variables given
 29.952      60.000      2.2203      .69680E-01   .30000
 30000.      66.000
mean values, variables calc
 29.951      60.002      2.2203      .69680E-01   .30000
 30000.      66.002
standard deviations, variables given
 .00000E+00   .57735      .00000E+00   .00000E+00   .00000E+00
 .00000E+00   1.1547
standard deviations, variables calc
 .49268E-04   .57735      .37418E-05   .15760E-06   .63290E-06
 .00000E+00   1.1517
mean values, functions
 96.103      -5.5265      .19223
std devs, functions
 1.7410      2.2150      .43622E-02

infeasible designs for function 1 = 0
infeasible designs for function 2 = 0
infeasible designs for function 3 = 0

total number of infeasible designs = 0
```

Out of 100,000 simulations, there are no infeasible designs.

It is instructive to compare these results to the non-robust design. If we run the same simulation, with the same tolerances, for the nominal optimum, we get,

```
no. of trials= 100000
mean values, variables given
 30.000      60.000      2.2044      .67404E-01   .30000
 30000.      66.000
mean values, variables calc
 30.000      60.002      2.2044      .67404E-01   .30000
```

```

30000.      66.002
standard deviations, variables given
.00000E+00 .57735      .00000E+00 .00000E+00 .00000E+00
.00000E+00 1.1547
standard deviations, variables calc
.00000E+00 .57735      .43317E-05 .13389E-06 .63290E-06
.00000E+00 1.1517
mean values, functions
99.982     -.32641E-01 .19999
std devs, functions
1.8111     2.2673     .45353E-02

infeasible designs for function 1 = 49812
infeasible designs for function 2 = 49213
infeasible designs for function 3 = 0

total number of infeasible designs = 56366

```

Out of 100,000 simulations, 56,366 designs had at least one infeasible constraint.

3 Statistical Tolerances

3.1 Introduction

Worst-case analysis is almost always overly conservative. There are some conditions, such as thermal expansion, which must be treated as worst-case. Often, however, it is reasonable to assume that fluctuations are independent random variables. When this is the case, it is very unlikely they will simultaneously occur in the worst possible combination. With a statistical tolerance analysis, the low probability of a worst-case combination can be taken into account. By allowing a small number of *rejects*--infeasible designs--the designer can use larger tolerances, or, as will be shown, back away from the optimum design a smaller amount than for a worst-case analysis.

3.2 Background

For this situation, variables with tolerances will be treated as random variables. Typically a random variable is described by a distribution type and distribution characteristics such as the mean and variance (or standard deviation, which is the square root of variance). We will consider all of the variables which have tolerances to be random variables described by normal distributions, with a mean at the nominal value and specified standard deviation.

3.3 Calculating Transmitted Variation: Linear Statistical Analysis

Just as with worst-case tolerances, we will transmit the variation of the variables into the constraints. Once again, we will rely on a first-order Taylor series, only this time we will find the mean and variance of the series. The mean of a first-order Taylor series is just the nominal value; the variance is given by:

$$\sigma_{g_i}^2 = \sum_{j=1}^n \left(\frac{\partial g_i}{\partial x_j} \sigma_{x_j} \right)^2 + \sum_{j=1}^l \left(\frac{\partial g_i}{\partial p_j} \sigma_{p_j} \right)^2 \quad (8.7)$$

$$\sigma_i^2 = \sigma_{b_i}^2 + \sigma_{g_i}^2 \quad (8.8)$$

We will refer to σ_i^2 as the “total constraint variance.”

Thus we will consider that randomness in the variables *induces* randomness in the functions. The constraint boundaries are no longer described by a sharp boundary but rather by a distribution, with the mean at the nominal value, and with one tail inside the feasible region and one tail outside the feasible region. This is illustrated in Fig. 8.6a.

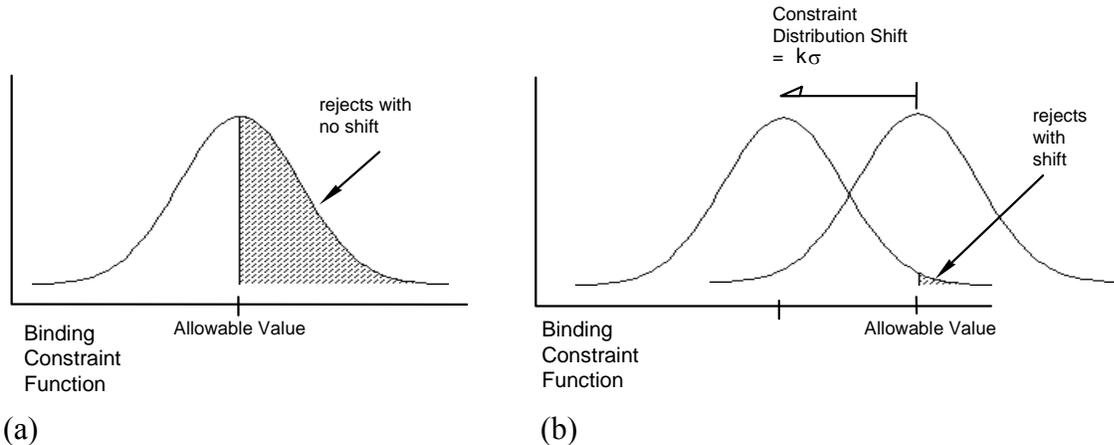


Fig. 8.6 The distribution of a constraint function. (a) The distribution with no shift. (b) The distribution with a shift to reduce the number of rejects.

We note that the distributions shown in Fig. 8.6 are normal. This is an assumption of this method. An important theorem in statistics, the Central Limit Theorem, states that sums and differences of random variables, regardless of their distributions, will tend to be normal. With engineering models, variables combine not only as sums and differences but as products, quotients, powers, etc. This means the assumption that the functions are normally distributed will only be approximately satisfied.

3.4 Developing a Robust Optimal Design with Statistical Tolerances

3.4.1 Compensating for Statistical Variation

As with worst-case tolerances, we will modify the constraints to take into account the transmitted variation. Thus each constraint becomes,

$$g_i + k\sigma_i \leq b_i \tag{8.9}$$

Alternatively, we can consider that the variation has reduced the right hand side,

$$g_i \leq b_i - k\sigma_i \tag{8.10}$$

where k is the number of standard deviations we decide to shift. Unlike the worst-case approach where we wanted to be feasible 100% of the time, with the statistical approach we set the level of feasibility we desire. For a normal distribution,

Value of k (number of standard deviations)	Percentage of Designs that will be Feasible (Normal Distribution)
1	84.13
2	97.725
3	99.865
4	99.9968

Shifting a constraint to control the number of rejects is illustrated in the Fig. 8.6b.

To obtain a robust optimum based on a statistical analysis, we follow essentially the same steps as with the worst-case approach,

1. Drive to the nominal optimum.
2. Calculate the transmitted variance.
3. Reduce the allowable value for a constraint by the amount of k times the standard deviation for a *less than* constraint; increase it for a *greater than* constraint. This has the effect of shifting the constraint distribution into the feasible region, as shown in Fig. 8.6b.
4. Assuming the transmitted variation is a constant, re-optimize to find the robust optimum.
5. Estimate the overall estimated feasibility as the product of the feasibilities for the binding constraints.

The assumptions of this approach include,

- Variables are independent and normally distributed.
- The robust optimum is close to the nominal optimum.
- Derivatives are constant, i.e., second derivatives are equal to zero. This assumption is consistent with assuming a linear Taylor expansion, (8.7), for the transmitted variation.
- Constraints are normally distributed.
- Constraints are not correlated. This means that for a random perturbation, the probability of one constraint being violated is independent of other constraints. This allows us to multiply the probability of each constraint together (step 5 above) to get the overall feasibility.

These assumptions are not always completely met, so we consider this method as a means of *estimating the order of magnitude of the number of rejects*. This means we will determine whether the per cent rejects will be 10%, 1%, 0.1%, etc. This level of accuracy is usually on a par with the accuracy of tolerance data available during the design stage.

3.4.2 An Example: The Two-bar Truss

In this section we will apply the method to the Two-bar truss. The tolerances are given in Table 3 below. For comparison to a worst-case tolerance analysis, the standard deviations are

made to be one third the worst-case tolerances given in the previous section. This means a worst-case tolerance band would be $\pm 3\sigma$.

Table 3 Statistical Tolerance Data for the Two-bar Truss

Description	Nominal Value	Standard Deviation
Height, H	Design Variable	1.67 in
Width, B	60 in	0.167 in
Diameter, d	Design Variable	0.033 in
Thickness, t	0.15 in	0.0033 in
Modulus	30000 ksi	500 ksi
Density	0.3 lb/in ³	0.0033 lb/in ³
Load, P	66 kips	1 kips
Yield Strength <i>Right Hand Side</i>	100 ksi	1.67 ksi
Buckling <i>Right Hand Side</i>	0.0	0.0
Deflection <i>Right Hand Side</i>	0.25 in	0.0

We desire each constraint to be feasible 99.865% of the time. To effect this, we calculate the variance for each constraint using (8.7). In the case of stress, which has a tolerance on the right hand side, we add in that variance using (8.9). We then subtract 3σ from the constraint right hand sides, as shown in Eq (8.10).

Fig. 8.7 is a contour plot showing the design space for this problem. The shaded area shows the decrease in the feasible region caused by the tolerances. The new constraint boundaries are shown as 1*, 2* and 3*. Comparing to Fig. 8.3, we see that the decrease is smaller than for worst-case tolerances. The optimal value of the objective has increased from 15.8 to 16.8 pounds.

We have two binding constraints that should each be feasible 99.865% of the time. The predicted overall feasibility is computed to be $0.99865 \times 0.99865 = 0.9973$ or 99.73%. Monte Carlo simulation of the robust optimum gives a feasibility of 99.8%.

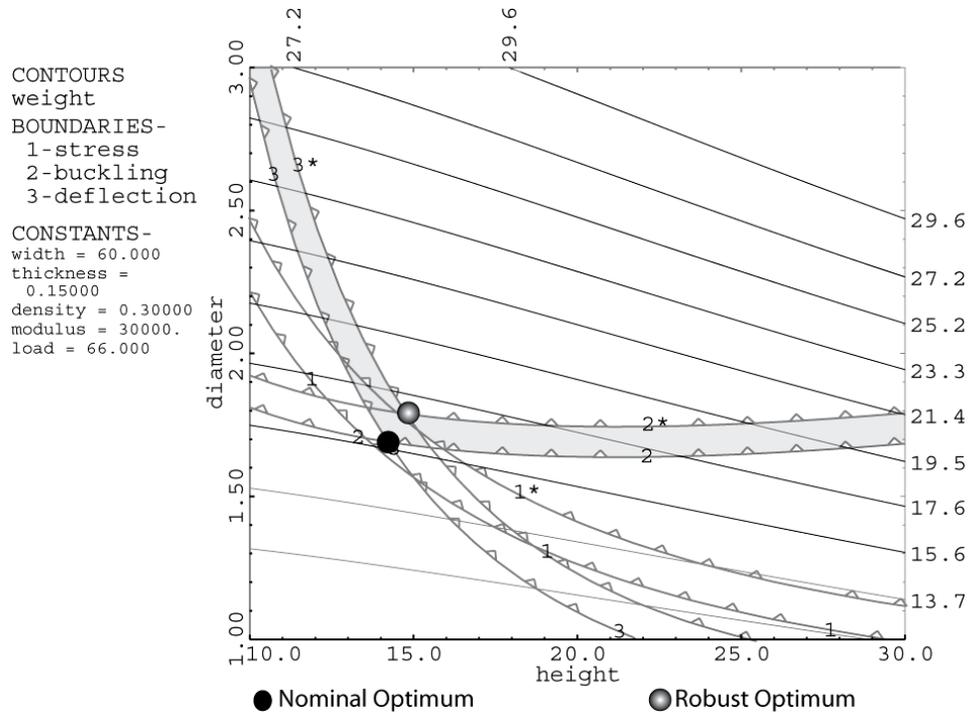


Fig. 8.7 Effect of statistical tolerances on optimum for Two-bar Truss problem. Shaded area is decrease in feasible region caused by including statistical tolerances. Compare to Fig. 8.3.

3.4.3 A Numerical Example: The Two-bar Truss

As we did for worst-case tolerances, we will show how to implement this method by hand. For this problem we have as design variables height, diameter, and thickness. The optimum occurs with height = 30, diameter = 2.2044, thickness = 0.06740, and an optimal weight of 11.88 lbs, with stress and buckling as binding constraints.

We wish to see the effect on the optimum of adding tolerances on the load and the width,

$$\sigma_{load} = 0.6667 \text{ kips}$$

$$\sigma_{width} = 0.3333 \text{ inch}$$

The first step is to drive to the nominal optimum, which we have already done. The second step is to calculate the transmitted variation using (8.7) with derivatives at the optimum:

Unscaled Gradients All Functions/All Variables				
	weight	stress	stress-buckling	deflection
height	0.1980491	-1.666198	1.667515	-0.003332284
diameter	5.390443	-45.35029	-136.0059	-0.09070057
thickness	176.2924	-1482.736	-1485.508	-2.965472
width	0.09902378	0.8331339	2.500016	0.004998807
density	39.60950	0.000000	0.000000	0.000000
modulus	0.000000	0.000000	-0.003333768	-6.665068e-06
load	0.000000	1.514788	1.514788	0.003029576

Dismiss Hardcopy

We estimate the variance of the functions as:

$$\sigma_{stress}^2 = (0.833 \times 0.333)^2 + (1.514 \times 0.667)^2 = 1.094$$

$$\sigma_{buckling}^2 = (2.500 \times 0.333)^2 + (1.514 \times 0.667)^2 = 1.713$$

$$\sigma_{deflection}^2 = (0.005 \times 0.333)^2 + (0.00303 \times 0.667)^2 = 0.000006857$$

We will shift the constraints by 3σ . These amounts are,

$$3\sigma_{stress} = 3.138$$

$$3\sigma_{buckling} = 3.926$$

$$3\sigma_{deflection} = 0.007856$$

We subtract these amounts from the right hand sides, as in (8.10),

$$\text{stress} \leq 100 - 3.138 = 96.862 \text{ ksi}$$

$$\text{buckling} \leq 0 - 3.926 = -3.926$$

$$\text{deflection} \leq 0.25 - 0.007856 = 0.24214 \text{ in}$$

When we re-optimize the new optimum is, height = 29.98, diameter = 2.21, thickness = 0.0694, with a weight of 12.27 pounds, and with stress and buckling as binding constraints.

3.5 Verifying the Robust Design with Monte Carlo Simulation

We verify these values using a Monte Carlo simulation similar to the one describe for worst-case tolerances, only the independent variables are given normal distributions instead of uniform distributions.

The output from running this program is given below,

```
no. of trials= 100000
mean values, variables given
```

```

29.980      60.000      2.2122      .69350E-01  .30000
30000.      66.000
mean values, variables calc
29.980      60.000      2.2122      .69350E-01  .30000
30000.      65.997
standard deviations, variables given
.00000E+00  .33330      .00000E+00  .00000E+00  .00000E+00
.00000E+00  .66670
standard deviations, variables calc
.38388E-04  .33262      .52221E-05  .47925E-07  .63290E-06
.00000E+00  .66709
mean values, functions
96.860      -3.9271      .19373
std devs, functions
1.0160      1.2838      .25398E-02

infeasible designs for function 1 = 118
infeasible designs for function 2 = 118
infeasible designs for function 3 = 0

total number of infeasible designs = 183

```

We have 0.183% infeasible designs. We predicted,

$$1 - (0.99865 \times 0.99865) = 0.00270 = 0.27\%$$

This is well within our desired order of magnitude accuracy.

4 Minimizing Variance: Sensitivity Robustness

4.1 Introduction

Up to this point we have considered only *feasibility robustness*: we wanted to develop designs that could tolerate variation and still work. We developed a method based on a linear approximation of transmitted variation.

For worst-case analysis, we estimate transmitted variation by,

$$\Delta g_i = \sum_{j=1}^n \left| \frac{\partial g_i}{\partial x_j} \Delta x_j \right| + \sum_{j=1}^m \left| \frac{\partial g_i}{\partial p_j} \Delta p_j \right|$$

For a statistical analysis, we estimate transmitted variation by,

$$\sigma_{g_i}^2 = \sum_{j=1}^n \left(\frac{\partial g_i}{\partial x_j} \sigma_{x_j} \right)^2 + \sum_{j=1}^m \left(\frac{\partial g_i}{\partial p_j} \sigma_{p_j} \right)^2$$

Besides feasibility robustness, we might also be interested in *sensitivity robustness*, which refers to *reducing the sensitivity of the design to variation*. This can be achieved by minimizing the transmitted variation as an objective in our optimization problem, either as

the sole objective in the problem or in combination with other objectives related to performance. The transmitted variation might also be a constraint.

The idea here is to find a region in the design space where *the derivatives of the function are small*. Thus we can reduce the effect of the tolerances *without reducing the tolerances themselves*. This idea is very similar to the central concept of Taguchi methods. Taguchi showed that it was possible to reduce variation in a product without reducing tolerances (which usually costs money) by moving to a different spot in the design space, where the design was less sensitive to the tolerances. In contrast to the computer models we are using, Taguchi based his method on using Design of Experiments to obtain models experimentally.

Minimizing variance can be computationally expensive, since we are minimizing a function which is composed of derivatives. To obtain a search direction, we will need to take second derivatives.

4.2 Example of Minimizing Variation

We will illustrate the concept of reducing transmitted variation by considering the design of a check valve—a device made to restrict flow to one direction only.

A diagram of a check valve is shown in Fig. 8.8. The purpose of the valve is to allow fluid flow in only one direction. Fluid can flow through the valve from left to right when the pressure of the flow overcomes the force exerted on the ball by the spring. The pressure required to unseat the ball is called the “cracking pressure.” It is desirable to minimize cracking pressure to reduce pressure drop across the valve; however, cracking pressure must be sufficient to prevent backflow. Design variables, parameters and tolerances for this problem are given in Table 4.

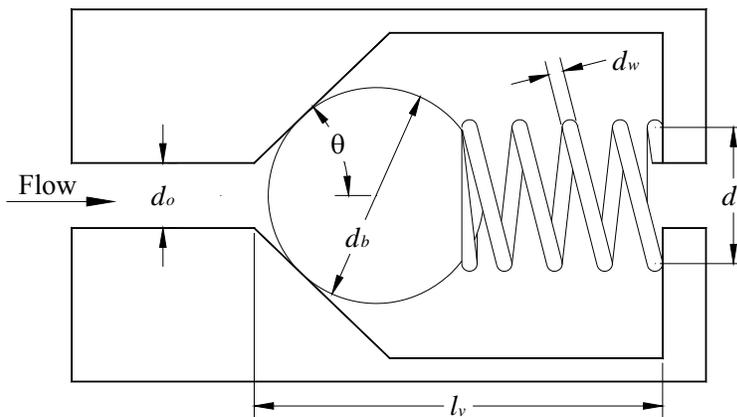


Fig. 8.8 Diagram of a check valve.

The design problem is to choose values for the variables to improve the sensitivity robustness of cracking pressure (i.e. reduce its sensitivity to variation), subject to constraints on cracking pressure, stress at full deflection, ball travel (spring compression) and various diameter ratios.

Table 4. Variables, Parameters and Tolerances for Check Valve

Design Variables	Tolerance	Standard Deviation
Seat angle, θ (degrees)	± 1.0	0.3333
Ball diameter, d_b (cm)	± 0.025	0.008333
Spring coil diameter, d_c (cm)	± 0.05	0.016667
Spring wire diameter, d_w (cm)	± 0.005	0.0016667
Spring length unloaded (cm)	± 0.1	0.03333
Spring number of coils	± 0.5	0.16667
Parameters		
Length of valve, l_v (cm)	± 0.1	0.03333
Diameter of orifice, d_v (cm)	± 0.025	0.008333
Shear modulus (kPa/m ²)	$\pm 7.e5$	2.333e5

We will need to combine sensitivity robustness with feasibility robustness to insure that cracking pressure is at least 15 kPa (about 2.2 psi) for 99% of all valves. This can be accomplished in two steps, where we first minimize variance and then add constraint shifts and re-optimize to achieve feasibility robustness. The starting design is given in the first column of Table 5.

After determining the minimum variance design, the next step was to shift it to obtain feasibility robustness. Constraint shifts based on a linear estimate of variance were inadequate for this problem so variance was calculated using a second order model. The shifted design is given in column 2 of Table 5.

Also shown in Table 5 is a comparison design. In order to determine the effect of minimizing variance we wanted to compare it to some sort of baseline design. We chose as a comparison design the design that results from maximizing ball travel as the objective, ignoring variance, and with all other constraints the same. The comparison design was then shifted to obtain feasibility robustness.

Table 5. Starting, Minimum Variance and Comparison Designs

Design Variables	Starting Design	Shifted Min Variance Design	Comparison Design*
Seat angle, θ (degrees)	45.	34.9	43.5
Ball diameter, d_b (cm)	1.25	1.47	0.720
Spring coil diameter, d_c (cm)	1.0	1.13	0.535
Spring wire diameter, d_w (cm)	0.075	0.0755	0.0379
Spring length unloaded (cm)	3.0	2.00	2.77
Spring number of coils	10	8	12
Parameters			
Length of valve, l_v (cm)	2.5	2.5	2.5
Diameter of orifice, d_v (cm)	0.635	0.635	0.635
Shear modulus (kPa/m ²)	8.3e7	8.3e7	8.3e7
Objective			

Variance of cracking pressure (kPa)	8.70	2.46	5.63
Constraints			
Cracking pressure ≥ 15 (kPa)	77	20.02 (b)	25.37 (b)
$0.5 \leq$ Ball diam/coil diam ≤ 0.8	0.8 (b)	0.77 (b)	0.74 (b)
$4 \leq$ Coil diam/wire diam ≤ 16	13.3	15.03 (b)	14.1 (b)
Ball travel ≥ 0.5 (cm)	0.8	0.60 (b)	1.63
Stress ≤ 900000 (kPa)	494000	295000	746600 (b)
Predicted Feasibility	N/A	96.06%	96.06%
Actual Feasibility	N/A	96.42%	96.59%

The symbol “(b)” indicates a binding constraint

*the objective for the comparison design was to maximize ball travel.

Monte Carlo simulation was used to verify robustness, and the results are shown in the last two rows of the table. During the simulation, pressure values were recorded so that the distributions could be graphed. These are shown in Fig. 8.9. The improvement of the minimum variance design over the comparison design is clearly evident.

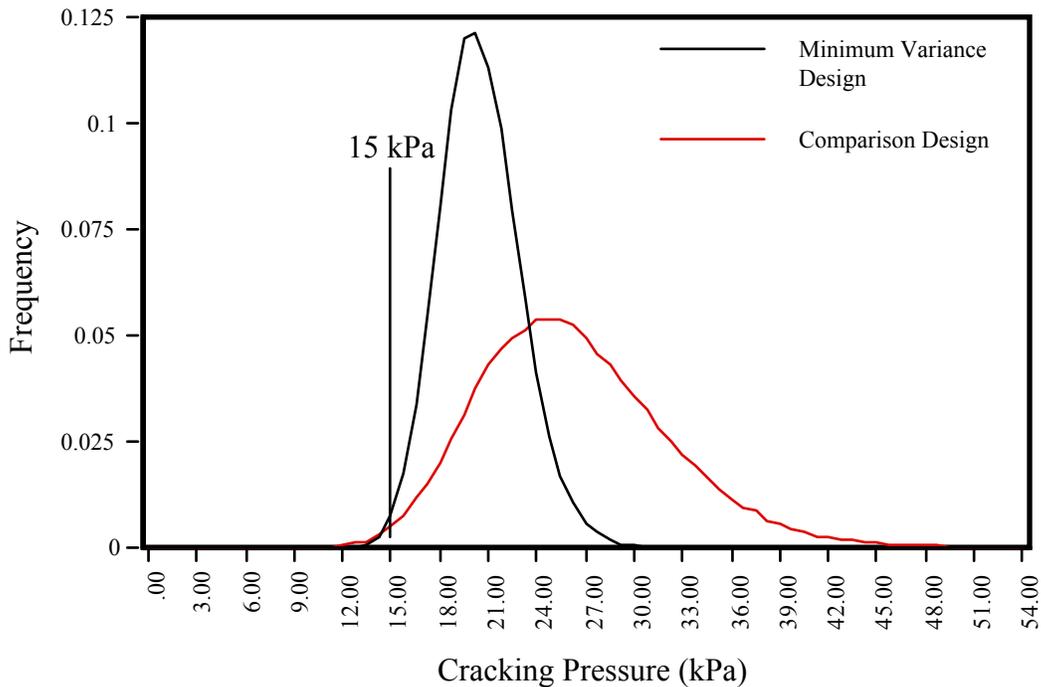


Fig. 8.9 Minimum variance and comparison design distributions for cracking pressure

5 References

A. Parkinson, C. Sorensen, and N. Pourhassan, “A General Approach for Robust Optimal Design,” *ASME J. of Mechanical Design*, Vol. 115, March 1993, pg. 74

A. Parkinson, “Robust Mechanical Design Using Engineering Models,” invited paper for special 50th anniversary issue of *J. of Mechanical Design*, vol. 117, p. 48-54, June 1995.

M. Phadke, *Quality Engineering Using Robust Design*, PTR Prentice Hall, 1989.