10 Introduction to Optimization

Find $\mathbf{x}$ that minimizes $F(\mathbf{x})$ subject to $g(\mathbf{x}) = 0$, $h(\mathbf{x}) \geq 0$

10.1 Introduction

Optimization is the term often used for minimizing or maximizing a function. It is sufficient to consider the problem of minimization only; maximization of $F(\mathbf{x})$ is achieved by simply minimizing $-F(\mathbf{x})$. In engineering, optimization is closely related to design. The function $F(\mathbf{x})$, called the *merit function* or *objective function*, is the quantity that we wish to keep as small as possible, such as the cost or weight. The components of $\mathbf{x}$, known as the *design variables*, are the quantities that we are free to adjust. Physical dimensions (lengths, areas, angles, etc.) are common examples of design variables.

Optimization is a large topic with many books dedicated to it. The best we can do in limited space is to introduce a few basic methods that are good enough for problems that are reasonably well behaved and do not involve too many design variables. By omitting the more sophisticated methods, we may actually not miss all that much. All optimization algorithms are unreliable to a degree – any one of them may work on one problem and fail on another. As a rule of the thumb, by going up in sophistication we gain computational efficiency, but not necessarily reliability.

The algorithms for minimization are iterative procedures that require starting values of the design variables $\mathbf{x}$. If $F(\mathbf{x})$ has several local minima, the initial choice of $\mathbf{x}$ determines which of these will be computed. There is no guaranteed way of finding the global optimal point. One suggested procedure is to make several computer runs using different starting points and pick the best result.

More often than not, the design variables are also subjected to restrictions, or *constraints*, which may have the form of equalities or inequalities. As an example, take the minimum weight design of a roof truss that has to carry a certain loading. Assume that the layout of the members is given, so that the design variables are the cross-sectional areas of the members. Here the design is dominated by inequality

constraints that consist of prescribed upper limits on the stresses and possibly the displacements.

The majority of available methods are designed for *unconstrained optimization*, where no restrictions are placed on the design variables. In these problems, the minima, if they exit, are stationary points (points where the gradient vector of $F(\mathbf{x})$ vanishes). In the more difficult problem of *constrained optimization*, the minima are usually located where the $F(\mathbf{x})$ surface meets the constraints. There are special algorithms for constrained optimization, but they are not easily accessible because of their complexity and specialization. One way to tackle a problem with constraints is to use an unconstrained optimization algorithm, but modify the merit function so that any violation of constrains is heavily penalized.

Consider the problem of minimizing $F(\mathbf{x})$ where the design variables are subject to the constraints

$$g_i(\mathbf{x}) = 0, \quad i = 1, 2, \ldots, M$$
$$h_j(\mathbf{x}) \leq 0, \quad j = 1, 2, \ldots, N$$

We choose the new merit function be

$$F^*(\mathbf{x}) = F(\mathbf{x}) + \mu P(\mathbf{x}) \tag{10.1a}$$

where

$$P(\mathbf{x}) = \sum_{i=1}^{M} [g_i(x)]^2 + \sum_{j=1}^{N} \left\{\max\left[0, h_j(\mathbf{x})\right]\right\}^2 \tag{10.1b}$$

is the *penalty function* and μ is a multiplier. The function $\max(a, b)$ returns the larger of a and b. It is evident that $P(\mathbf{x}) = 0$ if no constraints are violated. Violation of a constraint imposes a penalty proportional to the square of the violation. Hence, the minimization algorithm tends to avoid the violations, the degree of avoidance being dependent on the magnitude of μ. If μ is small, optimization will proceed faster because there is more "space" in which the procedure can operate, but there may be significant violation of constraints. On the other hand, large μ can result in a poorly conditioned procedure, but the constraints will be tightly enforced. It is advisable to run the optimization program with a μ that is on the small side. If the results show unacceptable constraint violation, increase μ and run the program again, starting with the results of the previous run.

An optimization procedure may also become ill conditioned when the constraints have widely different magnitudes. This problem can be alleviated by *scaling* the offending constraints, that is, multiplying the constraint equations by suitable constants.

It is not always necessary (or even advisable) to employ an iterative minimization algorithm. In problems where the derivatives of $F(\mathbf{x})$ can be readily computed and inequality constraints are absent, the optimal point can always be found directly by calculus. For example, if there are no constraints, the coordinates of the point where $F(\mathbf{x})$ is minimized are given by the solution of the simultaneous (usually nonlinear)

equations $\nabla F(\mathbf{x}) = \mathbf{0}$. The direct method for finding the minimum of $F(\mathbf{x})$ subject to equality constraints $g_i(\mathbf{x}) = 0$, $i = 1, 2, \ldots, m$ is to form the function

$$F^*(\mathbf{x}, \lambda) = F(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x}) \tag{10.2a}$$

and solve the equations

$$\nabla F^*(\mathbf{x}) = \mathbf{0} \qquad g_i(\mathbf{x}) = 0, \quad i = 1, 2, \ldots, m \tag{10.2b}$$

for $\mathbf{x}$ and λ_i. The parameters λ_i are known as the *Lagrangian multipliers*. The direct method can also be extended to inequality constraints, but the solution of the resulting equations is not straightforward because of lack of uniqueness.

10.2 Minimization along a Line

Consider the problem of minimizing a function $f(x)$ of a single variable x with the constraints $c \le x \le d$. A hypothetical plot of the function is shown in Fig. 10.1. There are two minimum points: a stationary point characterized by $f'(x) = 0$ that represents a local minimum, and a global minimum at the constraint boundary. It appears that finding the global minimum is simple. All the stationary points could be located by finding the roots of $df/dx = 0$, and each constraint boundary may be checked for a global minimum by evaluating $f(c)$ and $f(d)$. Then why do we need an optimization algorithm? We need it if $f(\mathbf{x})$ is difficult or impossible to differentiate – for example, if f represents a complex computer algorithm.

Bracketing

Before a minimization algorithm can be entered, the minimum point must be bracketed. The procedure of bracketing is simple: start with an initial value of x_0 and move *downhill* computing the function at $x_1, x_2, x_3, \ldots$ until we reach the point x_n where $f(x)$ increases for the first time. The minimum point is now bracketed in the interval (x_{n-2}, x_n). What should the step size $h_i = x_{i+1} - x_i$ be? It is not a good idea to have a

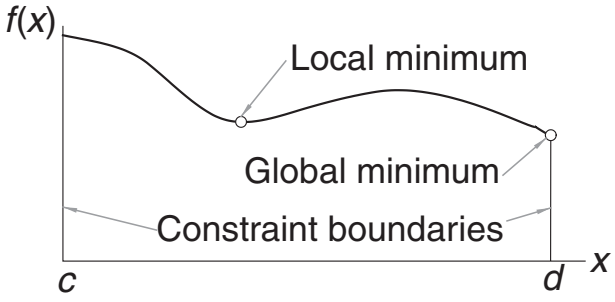


Figure 10.1. Example of local and global minima.

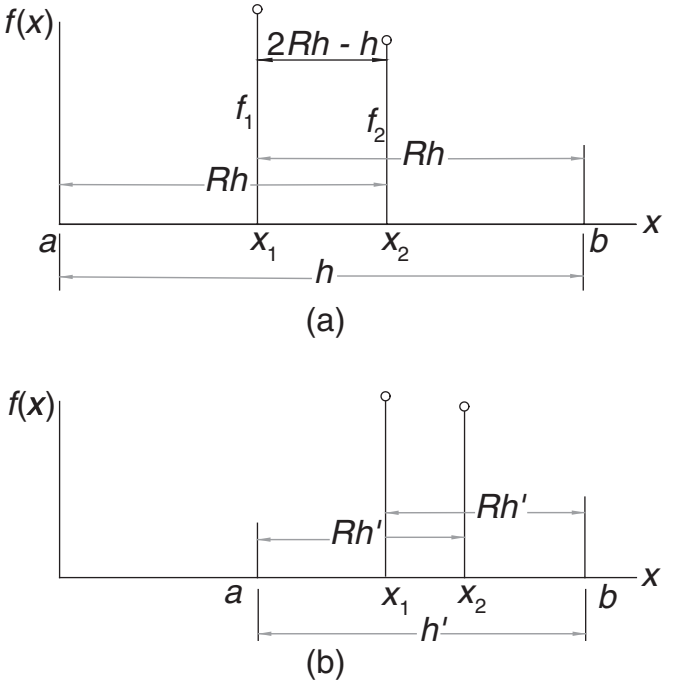


Figure 10.2. Golden section telescoping.

constant h_i, because it often results in too many steps. A more efficient scheme is to increase the size with every step, the goal being to reach the minimum quickly, even if its resulting bracket is wide. In our algorithm we chose to increase the step size by a constant factor, that is, we use $h_{i+1} = ch_i$, $c > 1$.

Golden Section Search

The golden section search is the counterpart of bisection used in finding roots of equations. Suppose that the minimum of $f(x)$ has been bracketed in the interval (a, b) of length h. To telescope the interval, we evaluate the function at $x_1 = b - Rh$ and $x_2 = a + Rh$, as shown in Fig. 10.2(a). The constant R is to be determined shortly. If $f_1 > f_2$ as indicated in the figure, the minimum lies in (x_1, b); otherwise, it is located in (a, x_2).

Assuming that $f_1 > f_2$, we set $a \leftarrow x_1$ and $x_1 \leftarrow x_2$, which yields a new interval (a, b) of length $h' = Rh$, as illustrated in Fig. 10.2(b). To carry out the next telescoping operation, we evaluate the function at $x_2 = a + Rh'$ and repeat the process.

The procedure works only if Figs. 10.1(a) and (b) are similar, that is, if the same constant R locates x_1 and x_2 in both figures. Referring to Fig. 10.2(a), we note that $x_2 - x_1 = 2Rh - h$. The same distance in Fig. 10.2(b) is $x_1 - a = h' - Rh'$. Equating the two, we get

$$2Rh - h = h' - Rh'$$

Substituting $h' = Rh$ and cancelling h yields

$$2R - 1 = R(1 - R)$$

the solution of which is the *golden ratio*.[1]

$$R = \frac{-1 + \sqrt{5}}{2} = 0.618\,033\,989\ldots \tag{10.3}$$

Note that each telescoping decreases the interval containing the minimum by the factor R, which is not as good as the factor is 0.5 in bisection. However, the golden search method achieves this reduction with *one function evaluation*, whereas two evaluations would be needed in bisection.

The number of telescopings required to reduce h from $|b - a|$ to an error tolerance ε is given by

$$|b - a|\, R^n = \varepsilon$$

which yields

$$n = \frac{\ln(\varepsilon / |b - a|)}{\ln R} = -2.078\,087 \ln \frac{\varepsilon}{|b - a|} \tag{10.4}$$

■ goldSearch

This module contains the bracketing and the golden section search algorithms. For the factor that multiplies successive search intervals in `bracket`, we chose $c = 1 + R$.

```
## module goldSearch
''' a,b = bracket(f,xStart,h)
    Finds the brackets (a,b) of a minimum point of the
    user-supplied scalar function f(x).
    The search starts downhill from xStart with a step
    length h.

    x,fMin = search(f,a,b,tol=1.0e-6)
    Golden section method for determining x that minimizes
    the user-supplied scalar function f(x).
    The minimum must be bracketed in (a,b).
'''
from math import log

def bracket(f,x1,h):
    c = 1.618033989
    f1 = f(x1)
    x2 = x1 + h; f2 = f(x2)
```

[1] R is the ratio of the sides of a "golden rectangle," considered by ancient Greeks to have the perfect proportions.

```
  # Determine downhill direction and change sign of h if needed
    if f2 > f1:
        h = -h
        x2 = x1 + h; f2 = f(x2)
      # Check if minimum between x1 - h and x1 + h
        if f2 > f1: return x2,x1 - h
  # Search loop
    for i in range (100):
        h = c*h
        x3 = x2 + h; f3 = f(x3)
        if f3 > f2: return x1,x3
        x1 = x2; x2 = x3
        f1 = f2; f2 = f3
    print ''Bracket did not find a minimum''

def search(f,a,b,tol=1.0e-9):
    nIter = -2.078087*log(tol/abs(b-a)) # Eq. (10.4)
    R = 0.618033989
    C = 1.0 - R
  # First telescoping
    x1 = R*a + C*b; x2 = C*a + R*b
    f1 = f(x1); f2 = f(x2)
  # Main loop
    for i in range(nIter):
        if f1 > f2:
            a = x1
            x1 = x2; f1 = f2
            x2 = C*a + R*b; f2 = f(x2)
        else:
            b = x2
            x2 = x1; f2 = f1
            x1 = R*a + C*b; f1 = f(x1)
    if f1 < f2: return x1,f1
    else: return x2,f2
```

EXAMPLE 10.1

Use `goldSearch` to find x that minimizes

$$f(x) = 1.6x^3 + 3x^2 - 2x$$

subject to the constraint $x \geq 0$. Compare the result with the analytical solution.

Solution This is a constrained minimization problem. Either the minimum of $f(x)$ is a stationary point in $x \geq 0$, or it is located at the constraint boundary $x = 0$. We handle the constraint with the penalty function method by minimizing $f(x) + \mu \left[\min(0, x)\right]^2$.

Starting at $x = 1$ and choosing $h = 0.01$ for the first step size in `bracket` (both choices being rather arbitrary), we arrive at the following program:

```
#!/usr/bin/python
## example10_1
from goldSearch import *

def f(x):
    mu = 1.0         # Constraint multiplier
    c = min(0.0, x)  # Constraint function
    return 1.6*x**3 + 3.0*x**2 - 2.0*x + mu*c**2

xStart = 1.0
h = 0.01
x1,x2 = bracket(f,xStart,h)
x,fMin = search(f,x1,x2)
print ''x ='',x
print ''f(x) ='',fMin
raw_input (''\nPress return to exit'')
```

The result is

```
x = 0.27349402621
f(x) = -0.28985978555
```

Because the minimum was found to be a stationary point, the constraint was not active. Therefore, the penalty function was superfluous, but we did not know that at the beginning.

The locations of stationary points are obtained analytically by solving

$$f'(x) = 4.8x^2 + 6x - 2 = 0$$

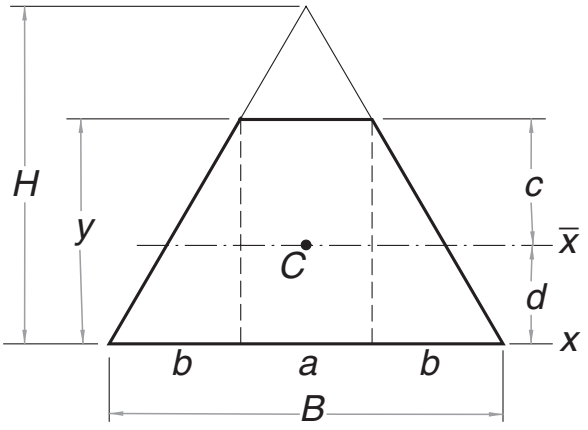
The positive root of this equation is $x = 0.273\,49\,4$. As this is the only positive root, there are no other stationary points in $x \geq 0$ that we must check out. The only other possible location of a minimum is the constraint boundary $x = 0$. But here $f(0) = 0$ is larger than the function at the stationary point, leading to the conclusion that the global minimum occurs at $x = 0.273\,49\,4$.

EXAMPLE 10.2

The trapezoid shown is the cross section of a beam. It is formed by removing the top from a triangle of base $B = 48$ mm and height $H = 60$ mm. The problem is to find the height y of the trapezoid that maximizes the section modulus

$$S = I_{\bar{x}}/c$$

where $I_{\bar{x}}$ is the second moment of the cross-sectional area about the axis that passes through the centroid C of the cross section. By optimizing the section modulus, we



minimize the maximum bending stress $\sigma_{\max} = M/S$ in the beam, M being the bending moment.

Solution Considering the area of the trapezoid as a composite of a rectangle and two triangles, the section modulus is found through the following sequence of computations:

Base of rectangle	$a = B(H - y)/H$
Base of triangle	$b = (B - a)/2$
Area	$A = (B + a)\, y/2$
First moment of area about x-axis	$Q_x = (ay)\, y/2 + 2\left(by/2\right) y/3$
Location of centroid	$d = Q_x/A$
Distance involved in S	$c = y - d$
Second moment of area about x-axis	$I_x = ay^3/3 + 2\left(by^3/12\right)$
Parallel axis theorem	$I_{\bar{x}} = I_x - Ad^2$
Section modulus	$S = I_{\bar{x}}/c$

We could use the formulas in the table to derive S as an explicit function of y, but that would involve a lot of error-prone algebra and result in an overly complicated expression. It makes more sense to let the computer do the work.

The program we used and its output are listed next. As we wish to maximize S with a minimization algorithm, the merit function is $-S$. There are no constraints in this problem.

```
#!/usr/bin/python
## example10_2
from goldSearch import *

def f(y):
    B = 48.0
    H = 60.0
    a = B*(H - y)/H
```

```
    b = (B - a)/2.0
    A = (B + a)*y/2.0
    Q = (a*y**2)/2.0 + (b*y**2)/3.0
    d = Q/A
    c = y - d
    I = (a*y**3)/3.0 + (b*y**3)/6.0
    Ibar = I - A*d**2
    return -Ibar/c

yStart = 60.0  # Starting value of y
h = 1.0        # Size of first step used in bracketing
a,b = bracket(f,yStart,h)
yOpt,fOpt = search(f,a,b)
print ''Optimal y ='',yOpt
print ''Optimal S ='',-fOpt
print ''S of triangle ='',-f(60.0)
raw_input(''Press return to exit'')

Optimal y = 52.1762738732
Optimal S = 7864.43094136
S of triangle = 7200.0
```

The printout includes the section modulus of the original triangle. The optimal section shows a 9.2% improvement over the triangle.

10.3 Powell's Method

Introduction

We now look at optimization in n-dimensional design space. The objective is to minimize $F(\mathbf{x})$, where the components of $\mathbf{x}$ are the n independent design variables. One way to tackle the problem is to use a succession of one-dimensional minimizations to close in on the optimal point. The basic strategy is

- Choose a point $\mathbf{x}_0$ in the design space.
- Loop with $i = 1, 2, 3, \ldots$

  Choose a vector $\mathbf{v}_i$.

  Minimize $F(\mathbf{x})$ along the line through $\mathbf{x}_{i-1}$ in the direction of $\mathbf{v}_i$. Let the minimum point be $\mathbf{x}_i$.

  if $|\mathbf{x}_i - \mathbf{x}_{i-1}| < \varepsilon$ exit loop

  $\mathbf{x}_i \leftarrow \mathbf{x}_{i+1}$

- end loop

The minimization along a line can be accomplished with any one-dimensional optimization algorithm (such as the golden section search). The only question left open is how to choose the vectors $\mathbf{v}_i$.

Conjugate Directions

Consider the quadratic function

$$F(\mathbf{x}) = c - \sum_i b_i x_i + \frac{1}{2}\sum_i \sum_j A_{ij} x_i x_j$$
$$= c - \mathbf{b}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} \tag{10.5}$$

Differentiation with respect to x_i yields

$$\frac{\partial F}{\partial x_i} = -b_i + \sum_j A_{ij} x_j$$

which can be written in vector notation as

$$\nabla F = -\mathbf{b} + \mathbf{A}\mathbf{x} \tag{10.6}$$

where ∇F is called the *gradient* of F.

Now consider the change in the gradient as we move from point $\mathbf{x}_0$ in the direction of a vector $\mathbf{u}$. The motion takes place along the line

$$\mathbf{x} = \mathbf{x}_0 + s\mathbf{u}$$

where s is the distance moved. Substitution into Eq. (10.6) yields the expression for the gradient at $\mathbf{x}$:

$$\nabla F|_{\mathbf{x}_0 + s\mathbf{u}} = -\mathbf{b} + \mathbf{A}(\mathbf{x}_0 + s\mathbf{u}) = \nabla F|_{\mathbf{x}_0} + s\,\mathbf{A}\mathbf{u}$$

Note that the change in the gradient is $s\,\mathbf{A}\mathbf{u}$. If this change is perpendicular to a vector $\mathbf{v}$, that is, if

$$\mathbf{v}^T\mathbf{A}\mathbf{u} = 0 \tag{10.7}$$

the directions of $\mathbf{u}$ and $\mathbf{v}$ are said to be mutually *conjugate* (noninterfering). The implication is that once we have minimized $F(\mathbf{x})$ in the direction of $\mathbf{v}$, we can move along $\mathbf{u}$ without ruining the previous minimization.

For a quadratic function of n independent variables it is possible to construct n mutually conjugate directions. Therefore, it would take precisely n line minimizations along these directions to reach the minimum point. If $F(\mathbf{x})$ is not a quadratic function, Eq. (10.5) can be treated as a local approximation of the merit

function, obtained by truncating the Taylor series expansion of $F(\mathbf{x})$ about $\mathbf{x}_0$ (see Appendix A1):

$$F(\mathbf{x}) \approx F(\mathbf{x}_0) + \nabla F(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \mathbf{H}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

Now the conjugate directions based on the quadratic form are only approximations, valid in the close vicinity of $\mathbf{x}_0$. Consequently, it would take several cycles of n line minimizations to reach the optimal point.

The various conjugate gradient methods use different techniques for constructing conjugate directions. The *zero-order methods* work with $F(\mathbf{x})$ only, whereas the *first-order methods* utilize both $F(\mathbf{x})$ and ∇F. The first-order methods are computationally more efficient, of course, but the input of ∇F, if it is available at all, can be very tedious.

Powell's Algorithm

Powell's method is a zero-order method, requiring the evaluation of $F(\mathbf{x})$ only. The basic algorithm is

- Choose a point $\mathbf{x}_0$ in the design space.
- Choose the starting vectors $\mathbf{v}_i$, $1, 2, \ldots, n$ (the usual choice is $\mathbf{v}_i = \mathbf{e}_i$, where $\mathbf{e}_i$ is the unit vector in the x_i-coordinate direction).
- cycle

  do with $i = 1, 2, \ldots, n$

  Minimize $F(\mathbf{x})$ along the line through $\mathbf{x}_{i-1}$ in the direction of $\mathbf{v}_i$. Let the minimum point be $\mathbf{x}_i$.

  end do

  $\mathbf{v}_{n+1} \leftarrow \mathbf{x}_0 - \mathbf{x}_n$

  Minimize $F(\mathbf{x})$ along the line through $\mathbf{x}_0$ in the direction of $\mathbf{v}_{n+1}$. Let the minimum point be $\mathbf{x}_{n+1}$.

  if $|\mathbf{x}_{n+1} - \mathbf{x}_0| < \varepsilon$ exit loop

  do with $i = 1, 2, \ldots, n$

  $\mathbf{v}_i \leftarrow \mathbf{v}_{i+1}$ ($\mathbf{v}_1$ is discarded, the other vectors are reused)

  end do

- end cycle

Powell demonstrated that the vectors $\mathbf{v}_{n+1}$ produced in successive cycles are mutually conjugate, so that the minimum point of a quadratic surface is reached in precisely n cycles. In practice, the merit function is seldom quadratic, but as long as it can be approximated locally by Eq. (10.5), Powell's method will work. Of course, it usually takes more than n cycles to arrive at the minimum of a nonquadratic function. Note that it takes n line minimizations to construct each conjugate direction.

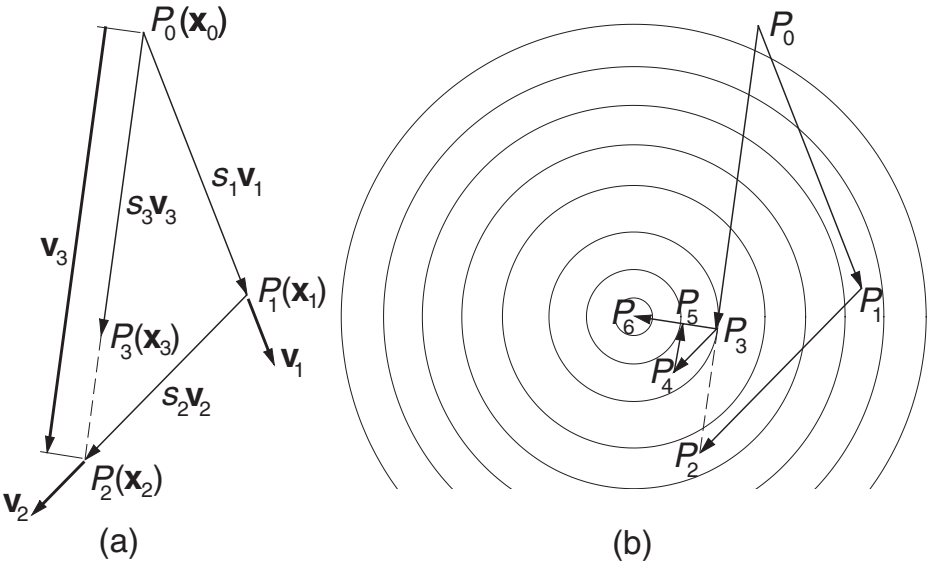


Figure 10.3. The method of Powell

Figure 10.3(a) illustrates one typical cycle of the method in a two-dimensional design space ($n = 2$). We start with point $\mathbf{x}_0$ and vectors $\mathbf{v}_1$ and $\mathbf{v}_2$. Then we find the distance s_1 that minimizes $F(\mathbf{x}_0 + s\mathbf{v}_1)$, finishing up at point $\mathbf{x}_1 = \mathbf{x}_0 + s_1\mathbf{v}_1$. Next, we determine s_2 that minimizes $F(\mathbf{x}_1 + s\mathbf{v}_2)$, which takes us to $\mathbf{x}_2 = \mathbf{x}_1 + s_2\mathbf{v}_2$. The last search direction is $\mathbf{v}_3 = \mathbf{x}_2 - \mathbf{x}_0$. After finding s_3 by minimizing $F(\mathbf{x}_0 + s\mathbf{v}_3)$, we get to $\mathbf{x}_3 = \mathbf{x}_0 + s_3\mathbf{v}_3$, completing the cycle.

Figure 10.3(b) shows the moves carried out in two cycles superimposed on the contour map of a quadratic surface. As explained before, the first cycle starts at point P_0 and ends up at P_3. The second cycle takes us to P_6, which is the optimal point. The directions P_0P_3 and P_3P_6 are mutually conjugate.

Powell's method does have a major flaw that has to be remedied – if $F(\mathbf{x})$ is not a quadratic, the algorithm tends to produce search directions that gradually become linearly dependent, thereby ruining the progress toward the minimum. The source of the problem is the automatic discarding of $\mathbf{v}_1$ at the end of each cycle. It has been suggested that it is better to throw out the direction that resulted in the *largest decrease* of $F(\mathbf{x})$, a policy that we adopt. It seems counterintuitive to discard the best direction, but it is likely to be close to the direction added in the next cycle, thereby contributing to linear dependence. As a result of the change, the search directions cease to be mutually conjugate, so that a quadratic form is not minimized in n cycles any more. This is not a significant loss because in practice $F(\mathbf{x})$ is seldom a quadratic.

Powell suggested a few other refinements to speed up convergence. Because they complicate the book keeping considerably, we did not implement them.

■ powell

The algorithm for Powell's method is listed here. It utilizes two arrays: `df` contains the decreases of the merit function in the first n moves of a cycle, and the matrix `u` stores the corresponding direction vectors $\mathbf{v}_i$(one vector per row).

```
## module powell
''' xMin,nCyc = powell(F,x,h=0.1,tol=1.0e-6)
    Powell's method of minimizing user-supplied function F(x).
    x    = starting point
    h    = initial search increment used in 'bracket'
    xMin = mimimum point
    nCyc = number of cycles
'''
from numpy import identity,array,dot,zeros,argmax
from goldSearch import *
from math import sqrt

def powell(F,x,h=0.1,tol=1.0e-6):

    def f(s): return F(x + s*v)    # F in direction of v

    n = len(x)                     # Number of design variables
    df = zeros(n)                  # Decreases of F stored here
    u = identity(n)                # Vectors v stored here by rows
    for j in range(30):            # Allow for 30 cycles:
        xOld = x.copy()            # Save starting point
        fOld = F(xOld)
      # First n line searches record decreases of F
        for i in range(n):
            v = u[i]
            a,b = bracket(f,0.0,h)
            s,fMin = search(f,a,
            df[i] = fOld - fMin
            fOld = fMin
            x = x + s*v
      # Last line search in the cycle
        v = x - xOld
        a,b = bracket(f,0.0,h)
        s,fLast = search(f,a,b)
        x = x + s*v
      # Check for convergence
        if sqrt(dot(x-xOld,x-xOld)/n) < tol: return x,j+1
      # Identify biggest decrease & update search directions
        iMax = argmax(df)
```

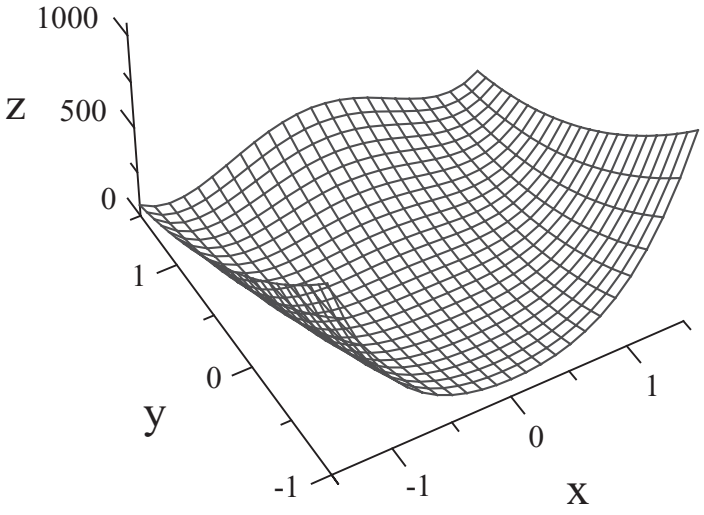
```
        for i in range(iMax,n-1):
            u[i] = u[i+1]
        u[n-1] = v
    print "Powell did not converge"
```

EXAMPLE 10.3

Find the minimum of the function[2]

$$F = 100(y - x^2)^2 + (1 - x)^2$$

with Powell's method starting at the point $(-1, 1)$. This function has an interesting topology. The minimum value of F occurs at the point $(1, 1)$. As seen in the figure, there is a hump between the starting and minimum points that the algorithm must negotiate.



Solution The program that solves this unconstrained optimization problem is

```
#!/usr/bin/python
## example10_3
from powell import *
from numpy import array

def F(x): return 100.0*(x[1] - x[0]**2)**2 + (1 - x[0])**2

xStart = array([-1.0, 1.0])
xMin,nIter = powell(F,xStart)
```

[2] From T. E. Shoup and F. Mistree, *Optimization Methods with Applications for Personal Computers* (Prentice-Hall, 1987).

```
print ''x ='',xMin
print ''F(x) ='',F(xMin)
print ''Number of cycles ='',nIter
raw_input (''Press return to exit'')
```

As seen in the printout, the minimum point was obtained after 12 cycles.

```
x = [ 1.  1.]
F(x) = 3.71750701585e-029
Number of cycles = 12
Press return to exit
```

EXAMPLE 10.4

Use `powell` to determine the smallest distance from the point (5, 8) to the curve $xy = 5$.

Solution This is a constrained optimization problem: minimize $F(x, y) = (x - 5)^2 + (y - 8)^2$ (the square of the distance) subject to the equality constraint $xy - 5 = 0$. The following program uses Powell's method with penalty function:

```
#!/usr/bin/python
## example10_4
from powell import *
from numpy import array
from math import sqrt

def F(x):
    mu = 1.0                        # Penalty multiplier
    c = x[0]*x[1] - 5.0             # Constraint equation
    return  distSq(x) + mu*c**2     # Penalized merit function

def distSq(x): return (x[0] - 5)**2 + (x[1] - 8)**2

xStart = array([1.0, 5.0])
x,numIter = powell(F,xStart,0.01)
print ''Intersection point ='',x
print ''Minimum distance ='', sqrt(distSq(x))
print ''xy ='', x[0]*x[1]
print ''Number of cycles ='',numIter
raw_input (''Press return to exit'')
```

As mentioned before, the value of the penalty function multiplier μ (called `mu` in the program) can have profound effect on the result. We chose $\mu = 1$ (as in the program listing) with the following result:

```
Intersection point = [ 0.73306761  7.58776385]
Minimum distance = 4.28679958767
```

```
xy = 5.56234387462
Number of cycles = 5
```

The small value of μ favored speed of convergence over accuracy. Because the violation of the constraint $xy = 5$ is clearly unacceptable, we ran the program again with $\mu = 10\,000$ and changed the starting point to (0.73306761, 7.58776385), the end point of the first run. The results shown next are now acceptable:

```
Intersection point = [ 0.65561311  7.62653592]
Minimum distance = 4.36040970945
xy = 5.00005696357
Number of cycles = 5
```

Could we have used $\mu = 10\,000$ in the first run? In this case, we would be lucky and obtain the minimum in 17 cycles. Hence, we save seven cycles by using two runs. However, a large μ often causes the algorithm to hang up, so that it is generally wise to start with a small μ.

Check

Because we have an equality constraint, the optimal point can readily be found by calculus. The function in Eq. (10.2a) is (here λ is the Lagrangian multiplier)

$$F^*(x, y, \lambda) = (x-5)^2 + (y-8)^2 + \lambda(xy-5)$$

so that Eqs. (10.2b) become

$$\begin{aligned}
\frac{\partial F^*}{\partial x} &= 2(x-5) + \lambda y = 0 \\
\frac{\partial F^*}{\partial y} &= 2(y-8) + \lambda x = 0 \\
g(x) &= xy - 5 = 0
\end{aligned}$$

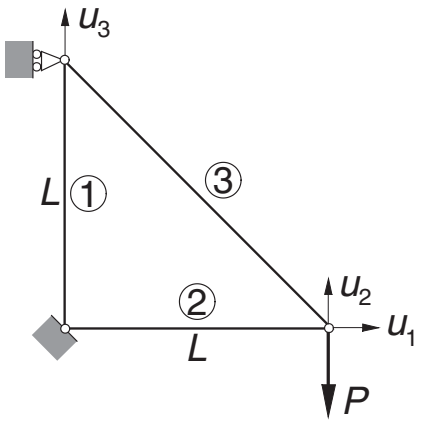
which can be solved with the Newton–Raphson method (the function `newtonRaphson2` in Section 4.6). In the following program we used the notation $\mathbf{x} = \begin{bmatrix} x & y & \lambda \end{bmatrix}^T$.

```
## example10_4_check
from numpy import array
from newtonRaphson2 import *

def F(x):
    return array([2.0*(x[0] - 5.0) + x[2]*x[1],  \
                  2.0*(x[1] - 8.0) + x[2]*x[0],  \
                  x[0]*x[1] - 5.0])

xStart = array([1.0, 5.0, 1.0])
print "x = ", newtonRaphson2(F,xStart)
raw_input (''Press return to exit'')
```

The result is

```
x =  [ 0.6556053   7.62653992  1.13928328]
```

EXAMPLE 10.5



The displacement formulation of the truss shown results in the following simultaneous equations for the joint displacements **u**:

$$\frac{E}{2\sqrt{2}L}\begin{bmatrix} 2\sqrt{2}A_2 + A_3 & -A_3 & A_3 \\ -A_3 & A_3 & -A_3 \\ A_3 & -A_3 & 2\sqrt{2}A_1 + A_3 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ -P \\ 0 \end{bmatrix}$$

where E represents the modulus of elasticity of the material and A_i is the cross-sectional area of member i. Use Powell's method to minimize the structural volume (i.e., the weight) of the truss while keeping the displacement u_2 below a given value δ.

Solution Introducing the dimensionless variables

$$v_i = \frac{u_i}{\delta} \qquad x_i = \frac{E\delta}{PL}A_i$$

the equations become

$$\frac{1}{2\sqrt{2}}\begin{bmatrix} 2\sqrt{2}x_2 + x_3 & -x_3 & x_3 \\ -x_3 & x_3 & -x_3 \\ x_3 & -x_3 & 2\sqrt{2}x_1 + x_3 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} \tag{a}$$

The structural volume to be minimized is

$$V = L(A_1 + A_2 + \sqrt{2}A_3) = \frac{PL^2}{E\delta}(x_1 + x_2 + \sqrt{2}x_3)$$

In addition to the displacement constraint $|u_2| \le \delta$, we should also prevent the cross-sectional areas from becoming negative by applying the constraints $A_i \ge 0$. Thus, the optimization problem becomes: Minimize

$$F = x_1 + x_2 + \sqrt{2}x_3$$

with the inequality constraints

$$|v_2| \le 1 \qquad x_i \ge 0 \ \ (i = 1, 2, 3)$$

Note that in order to obtain v_2 we must solve Eqs. (a).

Here is the program:

```
#!/usr/bin/python
## example10_5
from powell import *
from numpy import array
from math import sqrt
from gaussElimin import *

def F(x):
    global v, weight
    mu = 100.0
    c = 2.0*sqrt(2.0)
    A = array([[c*x[1] + x[2], -x[2],  x[2]],              \
               [-x[2],          x[2], -x[2]],              \
               [ x[2],         -x[2],  c*x[0] + x[2]]])/c
    b = array([0.0, -1.0, 0.0])
    v = gaussElimin(A,b)
    weight = x[0] + x[1] + sqrt(2.0)*x[2]
    penalty = max(0.0,abs(v[1]) - 1.0)**2   \
            + max(0.0,-x[0])**2              \
            + max(0.0,-x[1])**2              \
            + max(0.0,-x[2])**2
    return weight + penalty*mu

xStart = array([1.0, 1.0, 1.0])
x,numIter = powell(F,xStart)
print "x = ",x
print "v = ",v
print "Relative weight F = ",weight
print "Number of cycles = ",numIter
raw_input ("Press return to exit")
```

The first run of the program started with $\mathbf{x} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T$ and used $\mu = 100$ for the penalty multiplier. The results were

```
x =  [ 3.73870376  3.73870366  5.28732564]
v =  [-0.26747239 -1.06988953 -0.26747238]
Relative weight F =  14.9548150471
Number of cycles =  10
```

Because the magnitude of v_2 is excessive, the penalty multiplier was increased to 10,000 and the program run again using the output **x** from the last run as the input. As seen next, v_2 is now much closer to the constraint value.

```
x =  [ 3.99680758  3.9968077   5.65233961]
v =  [-0.25019968 -1.00079872 -0.25019969]
Relative weight F =  15.9872306185
Number of cycles =  11
```

In this problem, the use of $\mu = 10{,}000$ at the outset would not work. You are invited to try it.

10.4 Downhill Simplex Method

The downhill simplex method is also known as the *Nelder–Mead method*. The idea is to employ a moving simplex in the design space to surround the optimal point and then shrink the simplex until its dimensions reach a specified error tolerance. In n-dimensional space, a simplex is a figure of $n + 1$ vertices connected by straight lines and bounded by polygonal faces. If $n = 2$, a simplex is a triangle; if $n = 3$, it is a tetrahedron.

The allowed moves of the simplex are illustrated in Fig. 10.4 for $n = 2$. By applying these moves in a suitable sequence, the simplex can always hunt down the minimum point, enclose it, and then shrink around it. The direction of a move is determined by the values of $F(\mathbf{x})$ (the function to be minimized) at the vertices. The vertex with the highest value of F is labeled *Hi*, and *Lo* denotes the vertex with the lowest value. The

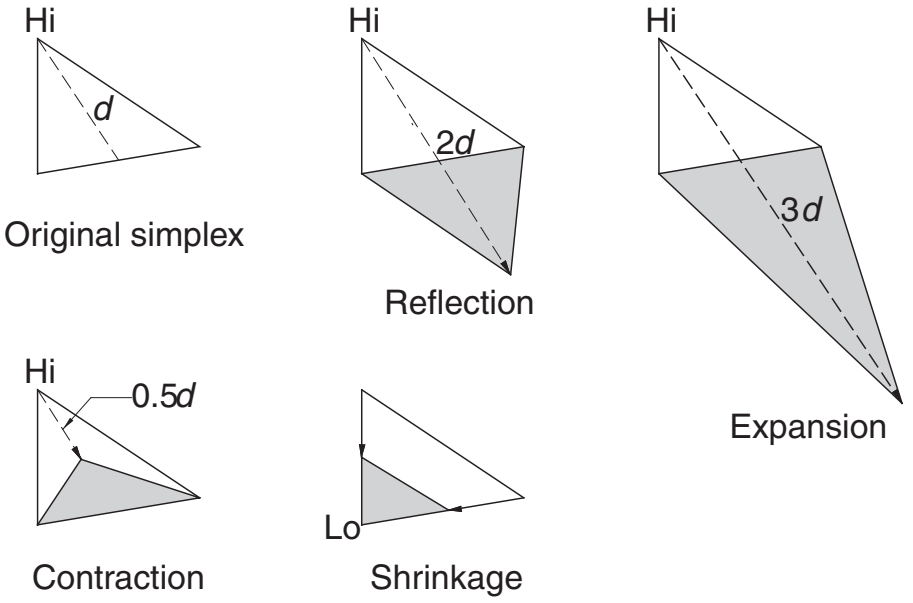


Figure 10.4. A simplex in two dimensions illustrating the allowed moves.

magnitude of a move is controlled by the distance d measured from the *Hi* vertex to the centroid of the opposing face (in the case of the triangle, the middle of the opposing side).

The outline of the algorithm is:

- Choose a starting simplex.
- Cycle until $d \leq \varepsilon$ (ε being the error tolerance)
  - Try reflection.
    - If the new vertex is lower than previous *Hi*, accept reflection.
    - If the new vertex is lower than previous *Lo*, try expansion.
    - If the new vertex is lower than previous *Lo*, accept expansion.
    - If reflection is accepted, start next cycle.
  - Try contraction.
    - If the new vertex is lower than *Hi*, accept contraction and start next cycle.
  - Shrinkage.
- end cycle

The downhill simplex algorithm is much slower than Powell's method in most cases, but makes up for it in robustness. It often works in problems where Powell's method hangs up.

■ downhill

The implementation of the downhill simplex method is given here. The starting simplex has one of its vertices at $\mathbf{x}_0$ and the others at $\mathbf{x}_0 + \mathbf{e}_i b$ ($i = 1, 2, \ldots, n$), where $\mathbf{e}_i$ is the unit vector in the direction of the x_i-coordinate. The vector $\mathbf{x}_0$ (called `xStart` in the program) and the edge length b of the simplex are input by the user.

```
## module downhill
''' x = downhill(F,xStart,side=0.1,tol=1.0e-6)
    Downhill simplex method for minimizing the user-supplied
    scalar function F(x) with respect to the vector x.
    xStart = starting vector x.
    side   = side length of the starting simplex (default = 0.1).
'''
from numpy import zeros,dot,argmax,argmin,sum
from math import sqrt

def downhill(F,xStart,side,tol=1.0e-6):
    n = len(xStart)                 # Number of variables
    x = zeros((n+1,n))
    f = zeros(n+1)

  # Generate starting simplex
    x[0] = xStart
```

```
  for i in range(1,n+1):
      x[i] = xStart
      x[i,i-1] = xStart[i-1] + side
# Compute values of F at the vertices of the simplex
  for i in range(n+1): f[i] = F(x[i])

# Main loop
  for k in range(500):
    # Find highest and lowest vertices
      iLo = argmin(f)
      iHi = argmax(f)
    # Compute the move vector d
      d = (-(n+1)*x[iHi] + sum(x))/n
    # Check for convergence
      if sqrt(dot(d,d)/n) < tol: return x[iLo]

    # Try reflection
      xNew = x[iHi] + 2.0*d
      fNew = F(xNew)
      if fNew <= f[iLo]:        # Accept reflection
          x[iHi] = xNew
          f[iHi] = fNew
        # Try expanding the reflection
          xNew = x[iHi] + d
          fNew = F(xNew)
          if fNew <= f[iLo]:    # Accept expansion
              x[iHi] = xNew
              f[iHi] = fNew
      else:
        # Try reflection again
          if fNew <= f[iHi]:    # Accept reflection
              x[iHi] = xNew
              f[iHi] = fNew
          else:
            # Try contraction
              xNew = x[iHi] + 0.5*d
              fNew = F(xNew)
              if fNew <= f[iHi]: # Accept contraction
                  x[iHi] = xNew
                  f[iHi] = fNew
              else:
                # Use shrinkage
                  for i in range(len(x)):
                      if i != iLo:
```

```
                        x[i] = (x[i] - x[iLo])
                        f[i] = F(x[i])
    print "Too many iterations in downhill"
    print "Last values of x were"
    return x[iLo]
```

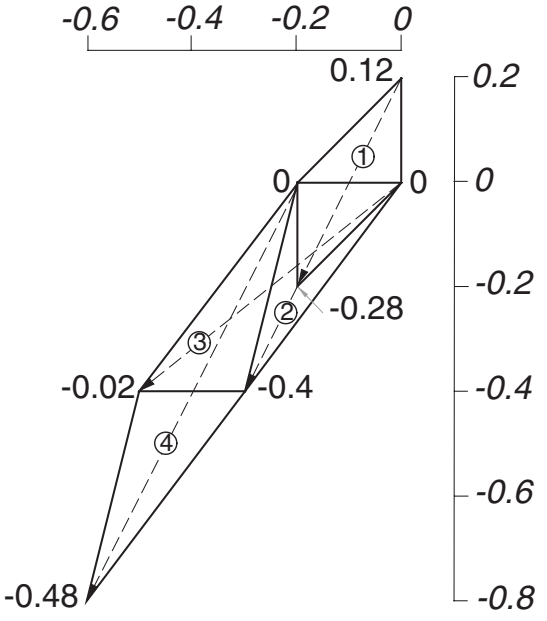
EXAMPLE 10.6

Use the downhill simplex method to minimize

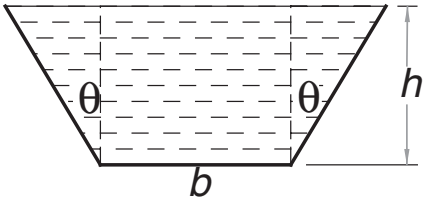
$$F = 10x_1^2 + 3x_2^2 - 10x_1x_2 + 2x_1$$

The coordinates of the vertices of the starting simplex are (0, 0), (0, −0.2), and (0.2, 0). Show graphically the first four moves of the simplex.

Solution The figure shows the design space (the x_1-x_2 plane). The numbers in the figure are the values of F at the vertices. The move numbers are enclosed in circles. The starting move (move 1) is a reflection, followed by an expansion (move 2). The next two moves are reflections. At this stage, the simplex is still moving downhill. Contraction will not start until move 8, when the simplex has surrounded the optimal point at (−0.6, −1.0).



EXAMPLE 10.7



The figure shows the cross section of a channel carrying water. Use the downhill simplex to determine h, b, and θ that minimize the length of the wetted perimeter while maintaining a cross-sectional area of 8 m^2. (Minimizing the wetted perimeter results in least resistance to the flow.) Check the answer by calculus.

Solution The cross-sectional area of the channel is

$$A = \frac{1}{2}\left[b + (b + 2h\tan\theta)\right]h = (b + h\tan\theta)h$$

and the length of the wetted perimeter is

$$S = b + 2(h\sec\theta)$$

The optimization problem is to minimize S subject to the constraint $A - 8 = 0$. Using the penalty function to take care of the equality constraint, the function to be minimized is

$$S^* = b + 2h\sec\theta + \mu\left[(b + h\tan\theta)h - 8\right]^2$$

Letting $\mathbf{x} = \begin{bmatrix} b & h & \theta \end{bmatrix}^T$ and starting with $\mathbf{x}_0 = \begin{bmatrix} 4 & 2 & 0 \end{bmatrix}^T$, we arrive at the following program:

```
#!/usr/bin/python
## example10_7
from numpy import array
from math import cos,tan,pi
from downhill import *

def S(x):
    global perimeter,area
    mu = 10000.0
    perimeter = x[0] + 2.0*x[1]/cos(x[2])
    area = (x[0] + x[1]*tan(x[2]))*x[1]
    return perimeter + mu*(area - 8.0)**2

xStart = array([4.0, 2.0, 0.0])
x = downhill(S,xStart)
area = (x[0] + x[1]*tan(x[2]))*x[1]
print "b = ",x[0]
print "h = ",x[1]
print "theta (deg) = ",x[2]*180.0/pi
print "area = ",area
print "perimeter = ",perimeter
raw_input("Finished. Press return to exit")
```

The results are

```
b =  2.4816069148
h =  2.14913738694
theta (deg) =  30.0000185796
area =  7.99997671775
perimeter =  7.44482803952
```

Check

Because we have an equality constraint, the problem can be solved by calculus with help from a Lagrangian multiplier. Referring to Eqs. (10.2a), we have $F = S$ and $g = A - 8$, so that

$$\begin{aligned} F^* &= S + \lambda(A - 8) \\ &= b + 2(h\sec\theta) + \lambda\left[(b + h\tan\theta)h - 8\right] \end{aligned}$$

Therefore, Eqs. (10.2b) become

$$\begin{aligned} \frac{\partial F^*}{\partial b} &= 1 + \lambda h = 0 \\ \frac{\partial F^*}{\partial h} &= 2\sec\theta + \lambda(b + 2h\tan\theta) = 0 \\ \frac{\partial F^*}{\partial \theta} &= 2h\sec\theta\tan\theta + \lambda h^2\sec^2\theta = 0 \\ g &= (b + h\tan\theta)h - 8 = 0 \end{aligned}$$

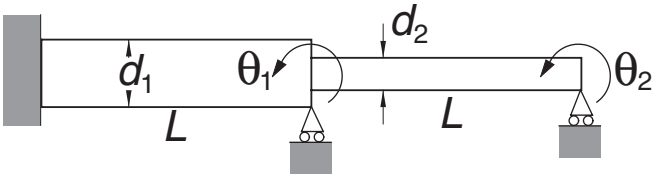
which can be solved with `newtonRaphson2` as shown next.

```
#!/usr/bin/python
## example10_7_check
from numpy import array,zeros
from math import tan,cos
from newtonRaphson2 import *

def f(x):
    f = zeros(4)
    f[0] = 1.0 + x[3]*x[1]
    f[1] = 2.0/cos(x[2]) + x[3]*(x[0] + 2.0*x[1]*tan(x[2]))
    f[2] = 2.0*x[1]*tan(x[2])/cos(x[2]) + x[3]*(x[1]/cos(x[2]))**2
    f[3] = (x[0] + x[1]*tan(x[2]))*x[1] - 8.0
    return f

xStart = array([3.0, 2.0, 0.0, 1.0])
print "x =",newtonRaphson2(f,xStart)
raw_input ("Press return to exit")
```

The solution $\mathbf{x} = \begin{bmatrix} b & h & \theta & \lambda \end{bmatrix}^T$ is

```
x = [ 2.48161296  2.14913986  0.52359878 -0.46530243]
```

EXAMPLE 10.8



The fundamental circular frequency of the stepped shaft is required to be higher than ω_0 (a given value). Use the downhill simplex to determine the diameters d_1 and d_2 that minimize the volume of the material without violating the frequency constraint. The approximate value of the fundamental frequency can be computed by solving the eigenvalue problem (obtainable from the finite element approximation)

$$\begin{bmatrix} 4(d_1^4 + d_2^4) & 2d_2^4 \\ 2d_2^4 & 4d_2^4 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \frac{4\gamma L^4 \omega^2}{105E} \begin{bmatrix} 4(d_1^2 + d_2^2) & -3d_2^2 \\ -3d_2^2 & 4d_2^2 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

where

$$\begin{aligned} \gamma &= \text{mass density of the material} \\ \omega &= \text{circular frequency} \\ E &= \text{modulus of elasticity} \\ \theta_1, \theta_2 &= \text{rotations at the simple supports} \end{aligned}$$

Solution We start by introducing the dimensionless variables $x_i = d_i/d_0$, where d_0 is an arbitrary "base" diameter. As a result, the eigenvalue problem becomes

$$\begin{bmatrix} 4(x_1^4 + x_2^4) & 2x_2^4 \\ 2x_2^4 & 4x_2^4 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \lambda \begin{bmatrix} 4(x_1^2 + x_2^2) & -3x_2^2 \\ -3x_2^2 & 4x_2^2 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \tag{a}$$

where

$$\lambda = \frac{4\gamma L^4 \omega^2}{105 E d_0^2}$$

In the program listed next we assume that the constraint on the frequency ω is equivalent to $\lambda \geq 0.4$.

```
## example10_8
from numpy import array
from stdForm import *
from inversePower import *
from downhill import *
```

```
def F(x):
    global eVal
    mu = 1.0e6
    eVal_min = 0.4
    A = array([[4.0*(x[0]**4 + x[1]**4), 2.0*x[1]**4],  \
               [2.0*x[1]**4, 4.0*x[1]**4]])
    B = array([[4.0*(x[0]**2 + x[1]**2), -3.0*x[1]**2],   \
               [-3*x[1]**2, 4.0*x[1]**2]])
    H,t = stdForm(A,B)
    eVal,eVec = inversePower(H,0.0)
    return x[0]**2 + x[1]**2 + mu*(max(0.0,eVal_min - eVal))**2

xStart = array([1.0,1.0])
x = downhill(F,xStart,0.1)
print "x = ", x
print "eigenvalue = ",eVal
raw_input ("Press return to exit")
```

Although a 2×2 eigenvalue problem can be solved easily, we avoid the work involved by employing functions that have been already prepared – `stdForm` to turn the eigenvalue problem into standard form, and `inversePower` to compute the eigenvalue closest to zero.

The results shown here were obtained with $x_1 = x_2 = 1$ as the starting values and 10^6 for the penalty multiplier. The downhill simplex method is robust enough to alleviate the need for multiple runs with increasing penalty multiplier.

```
x =  [ 1.07512696  0.79924677]
eigenvalue =  0.399997757238
```

PROBLEM SET 10.1

1. ■ The Lennard–Jones potential between two molecules is

$$V = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

   where ε and σ are constants, and r is the distance between the molecules. Use the module `goldSearch` to find σ/r that minimizes the potential, and verify the result analytically.
2. ■ One wave function of the hydrogen atom is

$$\psi = C\left(27 - 18\sigma + 2\sigma^2\right)e^{-\sigma/3}$$

where

$$\sigma = zr/a_0$$

$$C = \frac{1}{81\sqrt{3\pi}}\left(\frac{z}{a_0}\right)^{2/3}$$

$z =$ nuclear charge

$a_0 =$ Bohr radius

$r =$ radial distance

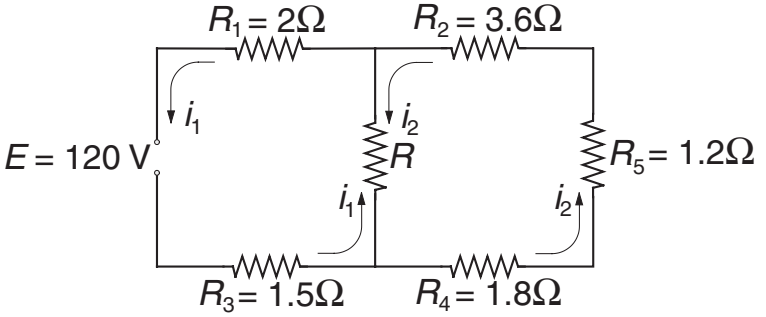
Find σ where ψ is at a minimum. Verify the result analytically.

3. ■ Determine the parameter p that minimizes the integral

$$\int_0^{\pi} \sin x \cos px \, dx$$

Hint: use numerical quadrature to evaluate the integral.

4. ■



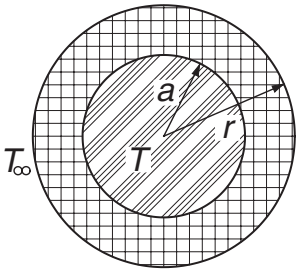
Kirchoff's equations for the two loops of the electrical circuit are

$$R_1 i_1 + R_3 i_1 + R(i_1 - i_2) = E$$

$$R_2 i_2 + R_4 i_2 + R_5 i_2 + R(i_2 - i_1) = 0$$

Find the resistance R that maximizes the power dissipated by R. *Hint*: Solve Kirchoff's equations numerically with one of the functions in Chapter 2.

5. ■



A wire carrying an electric current is surrounded by rubber insulation of outer radius r. The resistance of the wire generates heat, which is conducted through the insulation and convected into the surrounding air. The temperature of the

wire can be shown to be

$$T = \frac{q}{2\pi}\left(\frac{\ln(r/a)}{k} + \frac{1}{hr}\right) + T_\infty$$

where

$$\begin{aligned} q &= \text{rate of heat generation in wire} = 50\ \text{W/m} \\ a &= \text{radius of wire} = 5\ \text{mm} \\ k &= \text{thermal conductivity of rubber} = 0.16\ \text{W/m} \cdot \text{K} \\ h &= \text{convective heat-transfer coefficient} = 20\ \text{W/m}^2 \cdot \text{K} \\ T_\infty &= \text{ambient temperature} = 280\ \text{K} \end{aligned}$$

Find r that minimizes T.

6. ■ Minimize the function

$$F(x, y) = (x-1)^2 + (y-1)^2$$

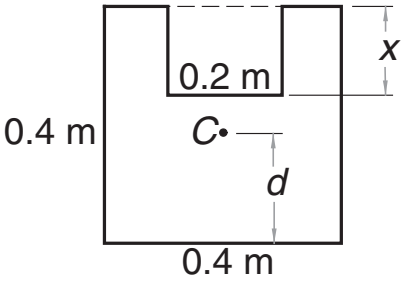
subject to the constraints $x + y \geq 1$ and $x \geq 0.6$.

7. ■ Find the minimum of the function

$$F(x, y) = 6x^2 + y^3 + xy$$

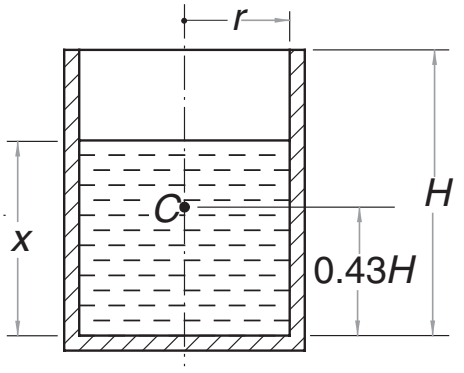
in $y \geq 0$. Verify the result analytically.

8. ■ Solve Prob. 7 if the constraint is changed to $y \geq -2$.
9. ■ Determine the smallest distance from the point $(1, 2)$ to the parabola $y = x^2$.
10. ■



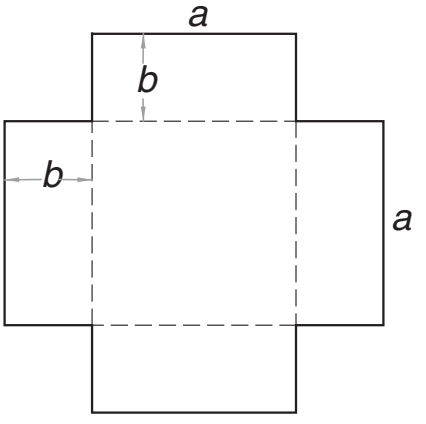
Determine x that minimizes the distance d between the base of the area shown and its centroid C.

11. ■



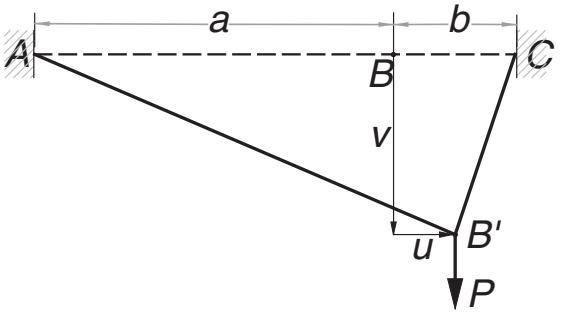
The cylindrical vessel of mass M has its center of gravity at C. The water in the vessel has a depth x. Determine x so that the center of gravity of the vessel–water combination is as low as possible. Use $M = 115$ kg, $H = 0.8$ m, and $r = 0.25$ m.

12. ■



The sheet of cardboard is folded along the dashed lines to form a box with an open top. If the volume of the box is to be 1.0 m^3, determine the dimensions a and b that would use the least amount of cardboard. Verify the result analytically.

13. ■



The elastic cord ABC has an extensional stiffness k. When the vertical force P is applied at B, the cord deforms to the shape $AB'C$. The potential energy of the

system in the deformed position is

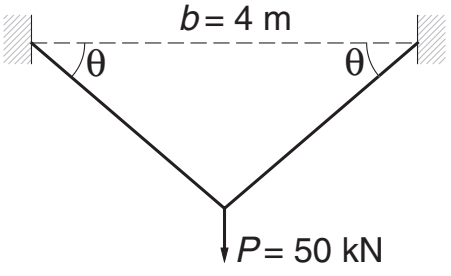
$$V = -Pv + \frac{k(a+b)}{2a}\delta_{AB} + \frac{k(a+b)}{2b}\delta_{BC}$$

where

$$\delta_{AB} = \sqrt{(a+u)^2 + v^2} - a$$
$$\delta_{BC} = \sqrt{(b-u)^2 + v^2} - b$$

are the elongations of AB and BC. Determine the displacements u and v by minimizing V (this is an application of the principle of minimum potential energy: a system is in stable equilibrium if its potential energy is at minimum). Use $a = 150$ mm, $b = 50$ mm, $k = 0.6$ N/mm, and $P = 5$ N.

14. ■



Each member of the truss has a cross-sectional area A. Find A and the angle θ that minimize the volume

$$V = \frac{bA}{\cos\theta}$$

of the material in the truss without violating the constraints

$$\sigma \le 150 \text{ MPa} \qquad \delta \le 5 \text{ mm}$$

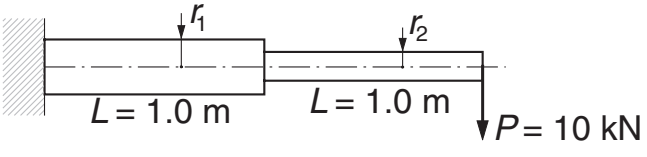
where

$$\sigma = \frac{P}{2A\sin\theta} = \text{stress in each member}$$
$$\delta = \frac{Pb}{2EA\sin 2\theta \sin\theta} = \text{ displacement at the load } P$$

and $E = 200 \times 10^9$.

15. ■ Solve Prob. 14 if the allowable displacement is changed to 2.5 mm.

16. ■



The cantilever beam of circular cross section is to have the smallest volume possible subject to constraints

$$\sigma_1 \le 180 \text{ MPa} \qquad \sigma_2 \le 180 \text{ MPa} \qquad \delta \le 25 \text{ mm}$$

where

$$\sigma_1 = \frac{8PL}{\pi r_1^3} = \text{ maximum stress in left half}$$

$$\sigma_2 = \frac{4PL}{\pi r_2^3} = \text{maximum stress in right half}$$

$$\delta = \frac{PL^3}{3\pi E}\left(\frac{7}{r_1^4} + \frac{1}{r_2^4}\right) = \text{displacement at free end}$$

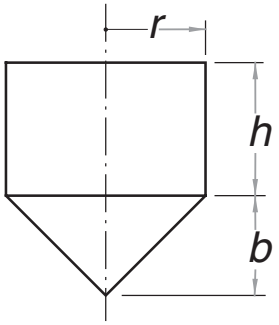
and $E = 200$ GPa. Determine r_1 and r_2.

17. ■ Find the minimum of the function

$$F(x, y, z) = 2x^2 + 3y^2 + z^2 + xy + xz - 2y$$

and confirm the result analytically.

18. ■

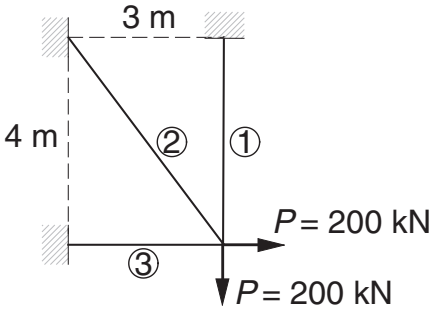


The cylindrical container has a conical bottom and an open top. If the volume V of the container is to be 1.0 m^3, find the dimensions r, h, and b that minimize the surface area S. Note that

$$V = \pi r^2\left(\frac{b}{3} + h\right)$$

$$S = \pi r\left(2h + \sqrt{b^2 + r^2}\right)$$

19. ■



The equilibrium equations of the truss shown are

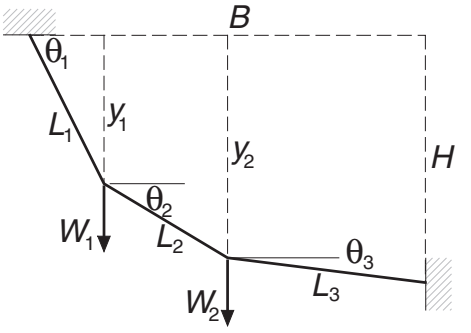
$$\sigma_1 A_1 + \frac{4}{5}\sigma_2 A_2 = P \qquad \frac{3}{5}\sigma_2 A_2 + \sigma_3 A_3 = P$$

where σ_i is the axial stress in member i and A_i are the cross-sectional areas. The third equation is supplied by compatibility (geometrical constraints on the elongations of the members):

$$\frac{16}{5}\sigma_1 - 5\sigma_2 + \frac{9}{5}\sigma_3 = 0$$

Find the cross-sectional areas of the members that minimize the weight of the truss without the stresses exceeding 150 MPa.

20. ■



A cable supported at the ends carries the weights W_1 and W_2. The potential energy of the system is

$$\begin{aligned} V &= -W_1 y_1 - W_2 y_2 \\ &= -W_1 L_1 \sin\theta_1 - W_2(L_1 \sin\theta_1 + L_2 \sin\theta_2) \end{aligned}$$

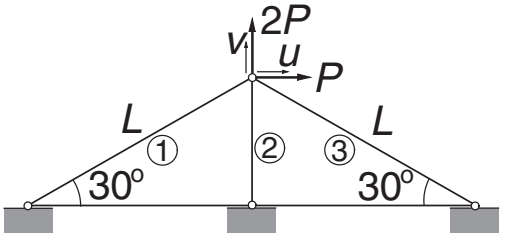
and the geometric constraints are

$$L_1 \cos\theta_1 + L_2 \cos\theta_2 + L_3 \cos\theta_3 = B$$

$$L_1 \sin\theta_1 + L_2 \sin\theta_2 + L_3 \sin\theta_3 = H$$

The principle of minimum potential energy states that the equilibrium configuration of the system is the one that satisfies geometric constraints and minimizes the potential energy. Determine the equilibrium values of θ_1, θ_2, and θ_3 given that $L_1 = 1.2$ m, $L_2 = 1.5$ m, $L_3 = 1.0$ m, $B = 3.5$ m, $H = 0$, $W_1 = 20$ kN, and $W_2 = 30$ kN.

21. ■



The displacement formulation of the truss results in the equations

$$\frac{E}{4L}\begin{bmatrix} 3A_1 + 3A_3 & \sqrt{3}A_1 + \sqrt{3}A_3 \\ \sqrt{3}A_1 + \sqrt{3}A_3 & A_1 + 8A_2 + A_3 \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} P \\ 2P \end{bmatrix}$$

where E is the modulus of elasticity, A_i is the cross-sectional area of member i, and u, v are the displacement components of the loaded joint. Letting $A_1 = A_3$ (a symmetric truss), determine the cross-sectional areas that minimize the structural volume without violating the constraints $u \leq \delta$ and $v \leq \delta$. *Hint*: nondimensionalize the problem as in Example 10.5.

22. ■ Solve Prob. 21 if the three cross-sectional areas are independent.
23. ■ A beam of rectangular cross section is cut from a cylindrical log of diameter d. Calculate the height h and the width b of the cross section that maximizes the cross-sectional moment of inertia $I = bh^3/12$. Check the result by calculus.

10.5 Other Methods

Simulated annealing methods have been successfully employed for complex problems involving many design variables. These methods are based on an analogy with the annealing as a slowly cooled liquid metal solidifies into a crystalline, minimum energy structure. One distinguishing feature of simulated annealing is its ability to pass over local minima in its search for the global minimum.

A topic that we reluctantly omitted is the *simplex method* of linear programming. Linear programming deals with optimization problems where the merit function and the constraints are linear expressions of the independent variables. The general linear programming problem is to minimize the objective function

$$F = \sum_{i=1}^{n} a_i x_i$$

subject to the constraints

$$\sum_{j=1}^{n} B_{ij} x_j \leq b_i, \quad i = 1, 2, \ldots, m_1$$

$$\sum_{j=1}^{n} C_{ij} x_j \geq c_i, \quad i = 1, 2, \ldots, m_2$$

$$\sum_{j=1}^{n} D_{ij} x_j = d_i, \quad i = 1, 2, \ldots, m_3$$

$$x_i \geq 0, \quad i = 1, 2, \ldots n$$

where the constants b_i, c_i, and d_i are non-negative. The roots of linear programming lie in cost analysis, operations research and related fields. We skip this topic because there are very few engineering applications that can be formulated as linear programming problems. In addition, a fail-safe implementation of the simplex method results in a rather complicated algorithm. This is not to say that the simplex method has no place in nonlinear optimization. There are several effective methods that rely in part on the simplex method. For example, problems with nonlinear constraints can often be solved by a piecewise application of linear programming. The simplex method is also used to compute search directions in the *method of feasible directions*.

Appendices

A1 Taylor Series

Function of a Single Variable

The Taylor series expansion of a function $f(x)$ about the point $x = a$ is the infinite series

$$f(x) = f(a) + f'(a)(x-a) + f''(a)\frac{(x-a)^2}{2!} + f'''(a)\frac{(x-a)^3}{3!} + \cdots \tag{A1}$$

In the special case $a = 0$, the series is also known as the *MacLaurin series*. It can be shown that Taylor series expansion is unique in the sense that no two functions have identical Taylor series.

The Taylor series is meaningful only if all the derivatives of $f(x)$ exist at $x = a$ and the series converges. In general, convergence occurs only if x is sufficiently close to a, that is, if $|x-a| \leq \varepsilon$, where ε is called the *radius of convergence*. In many cases, ε is infinite.

Another useful form of Taylor series is the expansion about an arbitrary value of x:

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2!} + f'''(x)\frac{h^3}{3!} + \cdots \tag{A2}$$

Because it is not possible to evaluate all the terms of an infinite series, the effect of truncating the series in Eq. (A2) is of great practical importance. Keeping the first $n+1$ terms, we have

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2!} + \cdots + f^{(n)}(x)\frac{h^n}{n!} + E_n \tag{A3}$$

where E_n is the *truncation error* (sum of the truncated terms). The bounds on the truncation error are given by *Taylor's theorem*:

$$E_n = f^{(n+1)}(\xi)\frac{h^{n+1}}{(n+1)!} \tag{A4}$$

where ξ is some point in the interval $(x, x+h)$. Note that the expression for E_n is identical to the first discarded term of the series, but with x replaced by ξ. Because

the value of ξ is undetermined (only its limits are known), the most we can get out of Eq. (A4) are the upper and lower bounds on the truncation error.

If the expression for $f^{(n+1)}(\xi)$ is not available, the information conveyed by Eq. (A4) is reduced to

$$E_n = \mathcal{O}(h^{n+1}) \tag{A5}$$

which is a concise way of saying that the truncation error is *of the order of* h^{n+1}, or behaves as h^{n+1}. If h is within the radius of convergence, then

$$\mathcal{O}(h^n) > \mathcal{O}(h^{n+1})$$

that is, the error is always reduced if a term is added to the truncated series (this may not be true for the first few terms).

In the special case $n = 1$, Taylor's theorem is known as the *mean value theorem*:

$$f(x+h) = f(x) + f'(\xi)h, \quad x \le \xi \le x+h \tag{A6}$$

Function of Several Variables

If f is a function of the m variables $x_1, x_2, \ldots, x_m$, then its Taylor series expansion about the point $\mathbf{x} = [x_1, x_2, \ldots, x_m]^T$ is

$$f(\mathbf{x}+\mathbf{h}) = f(\mathbf{x}) + \sum_{i=1}^{m} \left.\frac{\partial f}{\partial x_i}\right|_{\mathbf{x}} h_i + \frac{1}{2!}\sum_{i=1}^{m}\sum_{j=1}^{m} \left.\frac{\partial^2 f}{\partial x_i \partial x_j}\right|_{\mathbf{x}} h_i h_j + \cdots \tag{A7}$$

This is sometimes written as

$$f(\mathbf{x}+\mathbf{h}) = f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot \mathbf{h} + \frac{1}{2}\mathbf{h}^T \mathbf{H}(\mathbf{x})\mathbf{h} + \ldots \tag{A8}$$

The vector ∇f is known as the *gradient* of f, and the matrix $\mathbf{H}$ is called the *Hessian matrix* of f.

EXAMPLE A1

Derive the Taylor series expansion of $f(x) = \ln(x)$ about $x = 1$.

Solution The derivatives of f are

$$f'(x) = \frac{1}{x} \qquad f''(x) = -\frac{1}{x^2} \qquad f'''(x) = \frac{2!}{x^3} \qquad f^{(4)} = -\frac{3!}{x^4} \text{ etc.}$$

Evaluating the derivatives at $x = 1$, we get

$$f'(1) = 1 \qquad f''(1) = -1 \qquad f'''(1) = 2! \qquad f^{(4)}(1) = -3! \text{ etc.}$$

which, upon substitution into Eq. (A1) together with $a = 1$, yields

$$\begin{aligned}\ln(x) &= 0 + (x-1) - \frac{(x-1)^2}{2!} + 2!\frac{(x-1)^3}{3!} - 3!\frac{(x-1)^4}{4!} + \cdots \\ &= (x-1) - \frac{1}{2}(x-1)^2 + \frac{1}{3}(x-1)^3 - \frac{1}{4}(x-1)^4 + \cdots\end{aligned}$$

EXAMPLE A2

Use the first five terms of the Taylor series expansion of e^x about $x = 0$:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

together with the error estimate to find the bounds of e.

Solution

$$e = 1 + 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + E_4 = \frac{65}{24} + E_4$$

$$E_4 = f^{(4)}(\xi)\frac{h^5}{5!} = \frac{e^\xi}{5!}, \quad 0 \le \xi \le 1$$

The bounds on the truncation error are

$$(E_4)_{\min} = \frac{e^0}{5!} = \frac{1}{120} \qquad (E_4)_{\max} = \frac{e^1}{5!} = \frac{e}{120}$$

Thus, the lower bound on e is

$$e_{\min} = \frac{65}{24} + \frac{1}{120} = \frac{163}{60}$$

and the upper bound is given by

$$e_{\max} = \frac{65}{24} + \frac{e_{\max}}{120}$$

which yields

$$\frac{119}{120}e_{\max} = \frac{65}{24} \qquad e_{\max} = \frac{325}{119}$$

Therefore,

$$\frac{163}{60} \le e \le \frac{325}{119}$$

EXAMPLE A3

Compute the gradient and the Hessian matrix of

$$f(x, y) = \ln\sqrt{x^2 + y^2}$$

at the point $x = -2$, $y = 1$.

Solution

$$\frac{\partial f}{\partial x} = \frac{1}{\sqrt{x^2 + y^2}}\left(\frac{1}{2}\frac{2x}{\sqrt{x^2 + y^2}}\right) = \frac{x}{x^2 + y^2} \qquad \frac{\partial f}{\partial y} = \frac{y}{x^2 + y^2}$$

$$\nabla f(x, y) = \begin{bmatrix} x/(x^2 + y^2) & y/(x^2 + y^2) \end{bmatrix}^T$$

$$\nabla f(-2, 1) = \begin{bmatrix} -0.4 & 0.2 \end{bmatrix}^T$$

$$\frac{\partial^2 f}{\partial x^2} = \frac{(x^2+y^2) - x(2x)}{(x^2+y^2)^2} = \frac{-x^2+y^2}{(x^2+y^2)^2}$$
$$\frac{\partial^2 f}{\partial y^2} = \frac{x^2-y^2}{(x^2+y^2)^2}$$
$$\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial^2 f}{\partial y \partial x} = \frac{-2xy}{(x^2+y^2)^2}$$

$$\mathbf{H}(x, y) = \begin{bmatrix} -x^2+y^2 & -2xy \\ -2xy & x^2-y^2 \end{bmatrix} \frac{1}{(x^2+y^2)^2}$$
$$\mathbf{H}(-2, 1) = \begin{bmatrix} -0.12 & 0.16 \\ 0.16 & 0.12 \end{bmatrix}$$

A2 Matrix Algebra

A matrix is a rectangular array of numbers. The *size* of a matrix is determined by the number of rows and columns, also called the *dimensions* of the matrix. Thus, a matrix of m rows and n columns is said to have the size $m \times n$ (the number of rows is always listed first). A particularly important matrix is the square matrix, which has the same number of rows and columns.

An array of numbers arranged in a single column is called a *column vector*, or simply a vector. If the numbers are set out in a row, the term *row vector* is used. Thus, a column vector is a matrix of dimensions $n \times 1$, and a row vector can be viewed as a matrix of dimensions $1 \times n$.

We denote matrices by boldface, uppercase letters. For vectors we use boldface, lowercase letters. Here are examples of the notation:

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{A9}$$

Indices of the elements of a matrix are displayed in the same order as its dimensions: The row number comes first, followed by the column number. Only one index is needed for the elements of a vector.

Transpose

The transpose of a matrix $\mathbf{A}$ is denoted by $\mathbf{A}^T$ and defined as

$$A_{ij}^T = A_{ji}$$

The transpose operation thus interchanges the rows and columns of the matrix. If applied to vectors, it turns a column vector into a row vector and *vice versa*. For example,

transposing **A** and **b** in Eq. (A9), we get

$$\mathbf{A}^T = \begin{bmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \\ A_{13} & A_{23} & A_{33} \end{bmatrix} \qquad \mathbf{b}^T = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

An $n \times n$ matrix is said to be *symmetric* if $\mathbf{A}^T = \mathbf{A}$. This means that the elements in the upper triangular portion (above the diagonal connecting A_{11} and A_{nn}) of a symmetric matrix are mirrored in the lower triangular portion.

Addition

The sum $\mathbf{C} = \mathbf{A} + \mathbf{B}$ of two $m \times n$ matrices **A** and **B** is defined as

$$C_{ij} = A_{ij} + B_{ij}, \quad i = 1, 2, \ldots, m; \quad j = 1, 2, \ldots, n \tag{A10}$$

Thus, the elements of **C** are obtained by adding elements of **A** to the elements of **B**. Note that addition is defined only for matrices that have the same dimensions.

Vector Products

The *dot* or *inner product* $c = \mathbf{a} \cdot \mathbf{b}$ of the vectors **a** and **b**, each of size m, is defined as the scalar

$$c = \sum_{k=1}^{m} a_k b_k \tag{A11}$$

It can also be written in the form $c = \mathbf{a}^T\mathbf{b}$. In NumPy, the function for the dot product is `dot(a,b)` or `inner(a,b)`.

The *outer product* $\mathbf{C} = \mathbf{a} \otimes \mathbf{b}$ is defined as the matrix

$$C_{ij} = a_i b_j \tag{A12}$$

An alternative notation is $\mathbf{C} = \mathbf{a}\mathbf{b}^T$. The NumPy function for the outer product is `outer(a,b)`.

Array Products

The *matrix product* $\mathbf{C} = \mathbf{AB}$ of an $l \times m$ matrix **A** and an $m \times n$ matrix **B** is defined by

$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}, \quad i = 1, 2, \ldots, l; \quad j = 1, 2, \ldots, n \tag{A12}$$

The definition requires the number of columns in **A** (the dimension m) to be equal to the number of rows in **B**. The matrix product can also be defined in terms of the dot product. Representing the ith row of **A** as the vector $\mathbf{a}_i$ and the jth column of **B** as the

vector $\mathbf{b}_j$, we have

$$\mathbf{AB} = \begin{bmatrix} \mathbf{a}_1 \cdot \mathbf{b}_1 & \mathbf{a}_1 \cdot \mathbf{b}_2 & \cdots & \mathbf{a}_1 \cdot \mathbf{b}_n \\ \mathbf{a}_2 \cdot \mathbf{b}_1 & \mathbf{a}_2 \cdot \mathbf{b}_2 & \cdots & \mathbf{a}_2 \cdot \mathbf{b}_n \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_\ell \cdot \mathbf{b}_1 & \mathbf{a}_\ell \cdot \mathbf{b}_2 & \cdots & \mathbf{a}_\ell \cdot \mathbf{b}_n \end{bmatrix} \tag{A13}$$

NumPy treats the matrix product as the dot product for arrays, so that the function `dot(A,B)` returns the matrix product of **A** and **B**.

NumPy defines the *inner product* of matrices **A** and **B** to be $\mathbf{C} = \mathbf{AB}^T$. Equation (A13) still applies, but now **b** represents the jth *row* of B.

NumPy's definition of the *outer product* of matrices **A** (size $k \times \ell$) and **B** (size $m \times n$) is as follows. Let $\mathbf{a}_i$ be the ith row of **A**, and let $\mathbf{b}_j$ represent the jth row of **B**. Then the outer product is of **A** and **B** is

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b}_1 & \mathbf{a}_1 \otimes \mathbf{b}_2 & \cdots & \mathbf{a}_1 \otimes \mathbf{b}_m \\ \mathbf{a}_2 \otimes \mathbf{b}_1 & \mathbf{a}_2 \otimes \mathbf{b}_2 & \cdots & \mathbf{a}_2 \otimes \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_k \otimes \mathbf{b}_1 & \mathbf{a}_k \otimes \mathbf{b}_2 & \cdots & \mathbf{a}_k \otimes \mathbf{b}_m \end{bmatrix} \tag{A14}$$

The submatrices $\mathbf{a}_i \otimes \mathbf{b}_j$ are of dimensions $\ell \times n$. As you can see, the size of the outer product is much larger than either **A** or **B**.

Identity Matrix

A square matrix of special importance is the identity or *unit matrix*

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{A15}$$

It has the property $\mathbf{AI} = \mathbf{IA} = \mathbf{A}$.

Inverse

The inverse of an $n \times n$ matrix **A**, denoted by $\mathbf{A}^{-1}$, is defined to be an $n \times n$ matrix that has the property

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I} \tag{A16}$$

Determinant

The determinant of a square matrix **A** is a scalar denoted by $|\mathbf{A}|$ or $\det(\mathbf{A})$. There is no concise definition of the determinant for a matrix of arbitrary size. We start with the

determinant of a 2×2 matrix, which is defined as

$$\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} = A_{11}A_{22} - A_{12}A_{21} \tag{A17}$$

The determinant of a 3×3 matrix is then defined as

$$\begin{vmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{vmatrix} = A_{11} \begin{vmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{vmatrix} - A_{12} \begin{vmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{vmatrix} + A_{13} \begin{vmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{vmatrix}$$

Having established the pattern, we can now define the determinant of an $n \times n$ matrix in terms of the determinant of an $(n-1) \times (n-1)$ matrix:

$$|\mathbf{A}| = \sum_{k=1}^{n} (-1)^{k+1} A_{1k} M_{1k} \tag{A18}$$

where M_{ik} is the determinant of the $(n-1) \times (n-1)$ matrix obtained by deleting the ith row and kth column of $\mathbf{A}$. The term $(-1)^{k+i} M_{ik}$ is called a *cofactor* of A_{ik}.

Equation (A18) is known as *Laplace's development* of the determinant on the first row of $\mathbf{A}$. Actually, Laplace's development can take place on any convenient row. Choosing the ith row, we have

$$|\mathbf{A}| = \sum_{k=1}^{n} (-1)^{k+i} A_{ik} M_{ik} \tag{A19}$$

The matrix $\mathbf{A}$ is said to be *singular* if $|\mathbf{A}| = 0$.

Positive Definiteness

An $n \times n$ matrix A is said to be positive definite if

$$\mathbf{x}^T \mathbf{A} \mathbf{x} > 0 \tag{A20}$$

for all nonvanishing vectors $\mathbf{x}$. It can be shown that a matrix is positive definite if the determinants of all its leading minors are positive. The leading minors of $\mathbf{A}$ are the n square matrices

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1k} \\ A_{12} & A_{22} & \cdots & A_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k1} & A_{k2} & \cdots & A_{kk} \end{bmatrix}, \quad k = 1, 2, \ldots, n$$

Therefore, positive definiteness requires that

$$A_{11} > 0, \quad \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} > 0, \quad \begin{vmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{vmatrix} > 0, \ldots, |A| > 0 \tag{A21}$$

Useful Theorems

We list without proof a few theorems that are utilized in the main body of the text. Most proofs are easy and could be attempted as exercises in matrix algebra.

$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T \tag{A22a}$$

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1} \tag{A22b}$$

$$\left|\mathbf{A}^T\right| = |\mathbf{A}| \tag{A22c}$$

$$|\mathbf{AB}| = |\mathbf{A}|\,|\mathbf{B}| \tag{A22d}$$

$$\text{if } \mathbf{C} = \mathbf{A}^T\mathbf{BA} \text{ where } \mathbf{B} = \mathbf{B}^T, \text{ then } \mathbf{C} = \mathbf{C}^T \tag{A22e}$$

EXAMPLE A4

Letting

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix} \qquad \mathbf{u} = \begin{bmatrix} 1 \\ 6 \\ -2 \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} 8 \\ 0 \\ -3 \end{bmatrix}$$

compute $\mathbf{u} + \mathbf{v}$, $\mathbf{u} \cdot \mathbf{v}$, $\mathbf{Av}$, and $\mathbf{u}^T\mathbf{Av}$.

Solution

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} 1+8 \\ 6+0 \\ -2-3 \end{bmatrix} = \begin{bmatrix} 9 \\ 6 \\ -5 \end{bmatrix}$$

$$\mathbf{u} \cdot \mathbf{v} = 1(8)) + 6(0) + (-2)(-3) = 14$$

$$\mathbf{Av} = \begin{bmatrix} \mathbf{a}_1 \cdot \mathbf{v} \\ \mathbf{a}_2 \cdot \mathbf{v} \\ \mathbf{a}_3 \cdot \mathbf{v} \end{bmatrix} = \begin{bmatrix} 1(8) + 2(0) + 3(-3) \\ 1(8) + 2(0) + 1(-3) \\ 0(8) + 1(0) + 2(-3) \end{bmatrix} = \begin{bmatrix} -1 \\ 5 \\ -6 \end{bmatrix}$$

$$\mathbf{u}^T\mathbf{Av} = \mathbf{u} \cdot (\mathbf{Av}) = 1(-1) + 6(5) + (-2)(-6) = 41$$

EXAMPLE A5

Compute $|\mathbf{A}|$, where $\mathbf{A}$ is given in Example A4. Is $\mathbf{A}$ positive definite?

Solution Laplace's development of the determinant on the first row yields

$$\begin{aligned} |\mathbf{A}| &= 1\begin{vmatrix} 2 & 1 \\ 1 & 2 \end{vmatrix} - 2\begin{vmatrix} 1 & 1 \\ 0 & 2 \end{vmatrix} + 3\begin{vmatrix} 1 & 2 \\ 0 & 1 \end{vmatrix} \\ &= 1(3) - 2(2) + 3(1) = 2 \end{aligned}$$

Development on the third row is somewhat easier because of the presence of the zero element:

$$\begin{aligned} |\mathbf{A}| &= 0\begin{vmatrix} 2 & 3 \\ 2 & 1 \end{vmatrix} - 1\begin{vmatrix} 1 & 3 \\ 1 & 1 \end{vmatrix} + 2\begin{vmatrix} 1 & 2 \\ 1 & 2 \end{vmatrix} \\ &= 0(-4) - 1(-2) + 2(0) = 2 \end{aligned}$$

To verify positive definiteness, we evaluate the determinants of the leading minors:

$$A_{11} = 1 > 0 \quad \text{O.K.}$$

$$\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} = \begin{vmatrix} 1 & 2 \\ 1 & 2 \end{vmatrix} = 0 \quad \text{Not O.K.}$$

A is not positive definite.

EXAMPLE A6

Evaluate the matrix product **AB**, where **A** is given in Example A4 and

$$\mathbf{B} = \begin{bmatrix} -4 & 1 \\ 1 & -4 \\ 2 & -2 \end{bmatrix}$$

Solution

$$\mathbf{AB} = \begin{bmatrix} \mathbf{a}_1 \cdot \mathbf{b}_1 & \mathbf{a}_1 \cdot \mathbf{b}_2 \\ \mathbf{a}_2 \cdot \mathbf{b}_1 & \mathbf{a}_2 \cdot \mathbf{b}_2 \\ \mathbf{a}_3 \cdot \mathbf{b}_1 & \mathbf{a}_3 \cdot \mathbf{b}_2 \end{bmatrix}$$

$$= \begin{bmatrix} 1(-4)+2(1)+3(2) & 1(1)+2(-4)+3(-2) \\ 1(-4)+2(1)+1(2) & 1(1)+2(-4)+1(-2) \\ 0(-4)+1(1)+2(2) & 0(1)+1(-4)+2(-2) \end{bmatrix} = \begin{bmatrix} 4 & -13 \\ 0 & -9 \\ 5 & -8 \end{bmatrix}$$

EXAMPLE A7

Compute $\mathbf{A} \otimes \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 5 & -2 \\ -3 & 4 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

Solution

$$\mathbf{A} \otimes \mathbf{b} = \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b} \\ \mathbf{a}_2 \otimes \mathbf{b} \end{bmatrix}$$

$$\mathbf{a}_1 \otimes \mathbf{b} = \begin{bmatrix} 5 \\ -2 \end{bmatrix} \begin{bmatrix} 1 & 3 \end{bmatrix} = \begin{bmatrix} 5 & 15 \\ -2 & -6 \end{bmatrix}$$

$$\mathbf{a}_2 \otimes \mathbf{b} = \begin{bmatrix} -3 \\ 4 \end{bmatrix} \begin{bmatrix} 1 & 3 \end{bmatrix} = \begin{bmatrix} -3 & -9 \\ 4 & 12 \end{bmatrix}$$

$$\therefore \mathbf{A} \otimes \mathbf{b} = \begin{bmatrix} 5 & 15 \\ -2 & -6 \\ -3 & -9 \\ 4 & 12 \end{bmatrix}$$