UNIVERSITY OF MORATUWA
M.Sc/PG Diploma in Operational Research

Dr . M. Indralingam

LECTURE NOTE 6

▪ UNCONSTRAINED OPTIMIZATION

6.1 MATHEMATICAL BASIS

Given a function $f : \mathcal{R}^n \to \mathcal{R}$, and $x^* \in \mathcal{R}^n$ such that $f(x^*) < f(x)$ for all $x \in \mathcal{R}^n$ then x^* is called a minimizer of f and $f(x^*)$ is the minimum(value) of f . We wish to develop algorithms that find x^*, given $f(x)$ and some starting vector x_1. Since $\max f = -\min(-f)$ we consider minimization problems only.

6.1.1 Calculus in n dimensions.

We will be dealing with scalar functions of n variables $f(x_1, x_2, \ldots, x_n)$, and so we need the basic facts from the calculus of n dimensions. For simplicity we may view these as generalizations of the 1-dimensional case.

First Derivative or Gradient. If the function $f(x_1, x_2, \ldots, x_n)$ has first partial derivatives with respect to each element x_i then the n-dimensional equivalent of the first derivative $f'(x)$ is the *gradient* vector

$$g(x) = \left[\frac{\partial f(x)}{\partial x_i}\right] = \begin{bmatrix} \dfrac{\partial f(x)}{\partial x_1} \\ \dfrac{\partial f(x)}{\partial x_2} \\ \vdots \\ \dfrac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

Second Derivative or Hessian. The n-dimensional equivalent of the second derivative $f''(x)$ is the *Hessian* matrix

$$H(x) = \left[\frac{\partial^2 f(x)}{\partial x_i \partial x_j}\right] = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n \partial x_n} \end{bmatrix}.$$

Taylor Series in n Dimensions. The Taylor series expansion of $f(x)$ about some $x_k \in \mathcal{R}^n$ is :

$$f(x) \approx f(x_k) + g^T(x_k)(x - x_k) + \frac{1}{2}(x - x_k)^T H(x_k)(x - x_k) + \cdots ,$$

where $f(x) \in \mathcal{R}^1$, $g(x_k)$ and $x_k \in \mathcal{R}^n$, and $H(x_k) \in \mathcal{R}^{n \times n}$.

6.1.2 Optimality Conditions.

The condition for the solution x^* of the zero-finding problem is easy to state : x^* is a solution if $f(x^*) = 0$.
The conditions for the solution of the minimum-finding problem are not so simple.

Conditions in 1-Dimension. From elementary calculus we know that for a 1-dimensional function the following conditions must hold at an optimum $x^* \in \mathcal{R}^1$:

$$\begin{aligned} f'(x^*) &= 0, \text{ and} \\ f''(x^*) &< 0, \text{ at a maximum,} \\ f''(x^*) &> 0, \text{ at a minimum,} \\ f''(x^*) &= 0, \text{ at a point of inflection.} \end{aligned}$$

These conditions are shown in Figure 6.1.

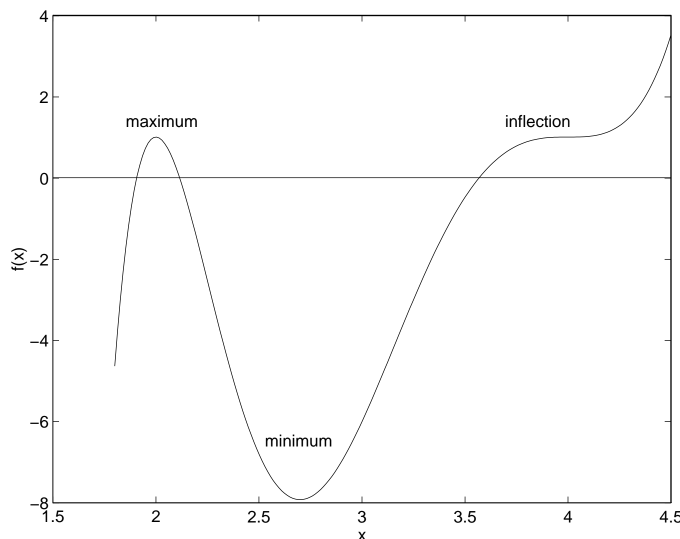

Figure 6.1. Three points where $f'(x) = 0$.

Conditions in n-Dimensions. The equivalent conditions in n dimensions are :

$$\begin{aligned}
&g(x^*) = 0, \text{ and} \\
&H(x^*) \text{ is negative definite at a maximum,} \\
&H(x^*) \text{ is positive definite at a minimum,} \\
&H(x^*) \text{ is indefinite at a saddle point,}
\end{aligned}$$

where $H(x^*)$ is symmetric and $x^T H(x^*)x$ is negative, positive, or 0 for all $\|x\| > 0$. Figure 6.2 shows three such points.

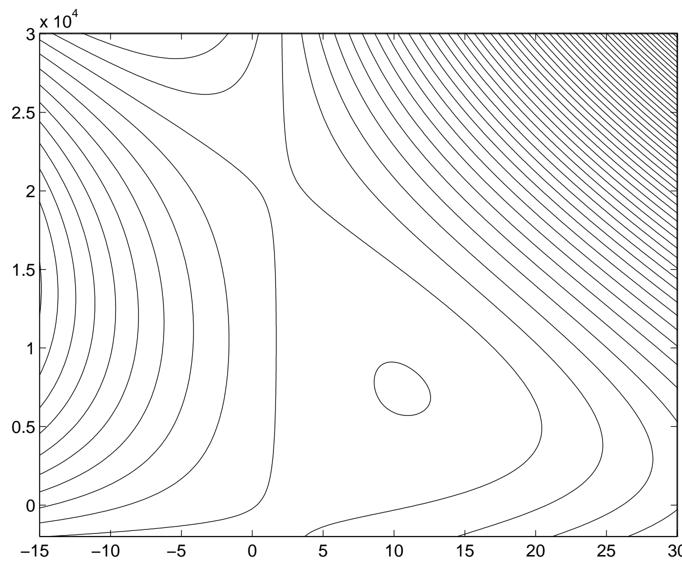

Figure 6.2. A Minimum and Two Saddle Points.

6.2 ONE-DIMENSIONAL MINIMIZATION

Given a function $f : \mathcal{R}^1 \to \mathcal{R}^1$, find $x \in \mathcal{R}^1$ such that $f(x^*) < f(x)$ for all $x \in \mathcal{R}^1$. Minimization in $\mathcal{R}^1$ is very similar to zero-finding in $\mathcal{R}^1$ in that we are looking for a point x^* such that $f(x^*)$ is a minimum, and in zero-finding we are looking for a point x^* such that $f(x^*) = 0$. We will see that one-dimensional minimization is at the heart of most minimum-finding algorithms in $\mathcal{R}^n$. Indeed most of the algorithms that search for a minimum in n dimensions perform a sequence of 1-dimensional searches in a chosen direction. The following prototype algorithm is called the Basic Descent Algorithm :

```
algorithm Basic-Descent (f, x_0)
---------------------------------------
    while not converged do
      1. Choose a descent direction d_k
      2. Find s_k that minimizes f(x_k + s_k d_k)
      3. x_{k+1} := x_k + s_k d_k
    endwhile
    return x_k

endalg {Basic-Descent}
```

In this algorithm x_k and d_k are n-vectors while s_k is a scalar. The algorithm, after choosing (somehow) a descent direction, searches for the minimum of f along the line through x_k in the direction d_k. This is called the *Line Search* step and, as such, is a 1-dimensional search.

6.2.1 Minimum-Finding versus Zero-Finding.

We will see that, conceptually, the search for the minimizer x^* is the same in each case : generate a sequence $\{x_{k+1} = T(x_k)\}$, and stop when the sequence converges. The only differences are

1. The manner in which the sequence $\{x_{k+1} = T(x_k)\}$ is generated and,
2. How convergence is checked.

We consider convergence-checking first because the convergence criteria for minimum-finding are fundamentally different from those for zero-finding. Mathematically the criteria for both are similar :

1. *Zero-Finding* : Stop when either $f_k = 0$ or $\|x_k - x_{k-1}\| \le \epsilon_x$
2. *Min-Finding* : Stop when either $|f_k - f_{k-1}| \le \epsilon_f = 0$ or $|x_k - x_{k-1}\| \le \epsilon_x$.

The most obvious difference is f-sequence convergence : for zero-finding we know that the value of the function at x^* is $f(x^*) = 0$, whereas for minimum finding we do not know the value of the minimum $f(x^*)$. We do know, however, that at a minimum the derivative $f'(x^*) = 0$ so that, in principle, we may replace a minimum-finding algorithm for $f(x)$ with a zero-finding algorithm for $f'(x)$. The fact that at a minimum the derivative $f'(x^*) = 0$ raises the second difference : x-sequence convergence. To see this let us expand $f(x_k)$ in a Taylor series about the minimizer x^*. We get

$$f(x_k) \approx f(x^*) + f'(x^*)(x_k - x^*) + \frac{1}{2}f''(x^*)(x_k - x^*)^2.$$

Now $f'(x^*) = 0$ and so

$$f(x_k) \approx f(x^*) + \frac{1}{2}f''(x^*)(x_k - x^*)^2.$$

From this we get the relationship between changes in f and changes in x. That is

$$\begin{aligned} f(x_k) - f(x^*) = \Delta f(x_k) &\approx \frac{1}{2}f''(x^*)\Delta x_k^2 \quad \text{or,} \\ e_f &= c\, e_x^2, \end{aligned}$$

where e_f and e_x are the relative errors in f and x, respectively.
If we wish to find the minimum $f(x^*)$ to full precision then the stopping rule will be

$$e_f \le e_{mach} \Rightarrow c\, e_x^2 \le e_{mach} \Rightarrow e_x \le \sqrt{e_{mach}}\ .$$

This means that in IEEE single precision with $e_{mach} = 2^{-24}$, the best precision we can get for x-sequence convergence is $\sqrt{e_{mach}} = 2^{-12}$, that is, half machine precision.

6.2.2 Unimodal Functions.

In one-dimensional minimization, the equivalent of a function that changes sign in an interval $[a, b]$ is a function that is *unimodal* on $[a, b]$. Roughly speaking, a unimodal function has one minimum on $[a, b]$.

Definition *(Unimodal Function).* A function $f(x)$ is unimodal on $[a, b]$ if, for all $x_1 < x_2 \in [a, b]$ there is a point $x^* \in [a, b]$ such that

$$\begin{aligned} x_2 < x^* &\Rightarrow f(x_1) > f(x_2), \quad \text{or} \\ x_1 > x^* &\Rightarrow f(x_1) < f(x_2). \end{aligned}$$

■

Theorem *(Reduction of Interval).* If $f(x)$ is a unimodal function on $[a, b]$ and $a \le x_1 < x_2 \le b$ then

$$\begin{aligned} f(x_1) \le f(x_2) &\Rightarrow x^* \le x_2, \quad \text{or} \\ f(x_1) \ge f(x_2) &\Rightarrow x^* \ge x_1. \end{aligned}$$

■

This theorem allows us to reduce the interval $[a, b]$ to either $[a, x_2]$ or $[x_1, b]$, as shown in Figures 6.3 and 6.4.

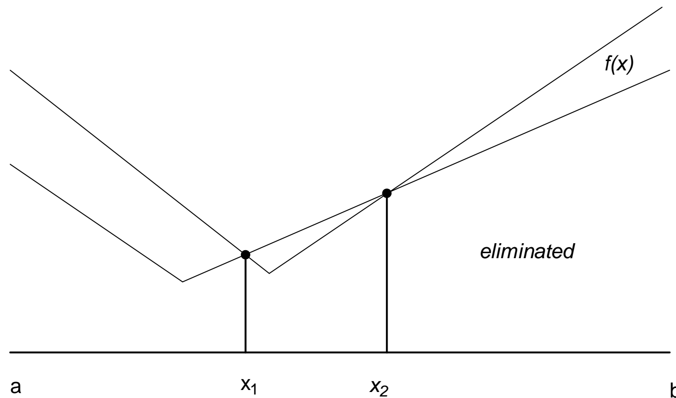

Figure 6.3. Reduction of the Interval.

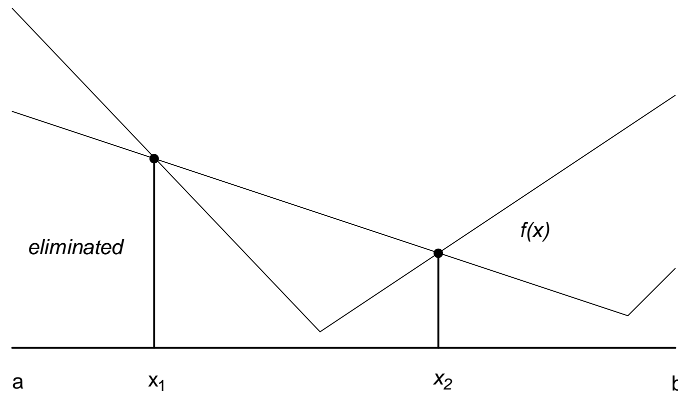

Figure 6.4. Reduction of the Interval.

6.2.3 Bisection Algorithm.

The Bisection algorithm (also called Binary Search) is similar to that for zero-finding. At each iteration it halves the interval of search by checking the value of the function at the mid-point of the interval. The starting condition is : given an interval $[a, b]$ on which the function is unimodal, i.e., it has a minimum in the interval. This is equivalent to the condition that the function changes sign in the interval when zero-finding.

Before we state the algorithm formally, let us see if the zero-finding Bisection method works.

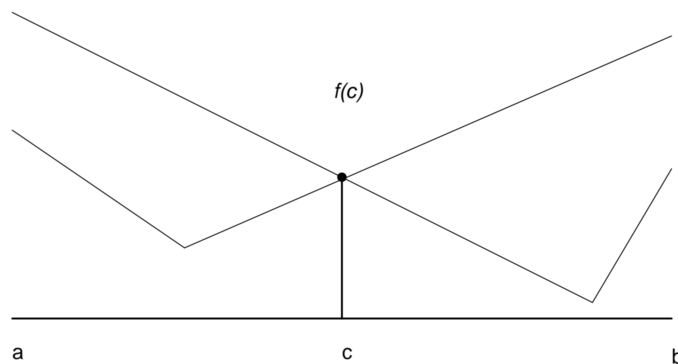

Figure 6.5. One f-evaluation gives no information.

It is obvious from Figure 6.5that one function evaluation gives no useful information. If we also had $f(a)$ and $f(b)$ would that help? No. It is easy to construct two function with $f_1(a) = f_2(a)$, $f_1(c) = f_2(c)$ and $f_1(b) = f_2(b)$ that have minima on different sides of the mid-point c.

To find out which side of c the minimum lies we need two function evaluations, as shown in Figure 6.6 This shows very clearly that minimum-finding is more difficult than zero-finding, at least for the Bisection algorithm.

We now formally state the algorithm.

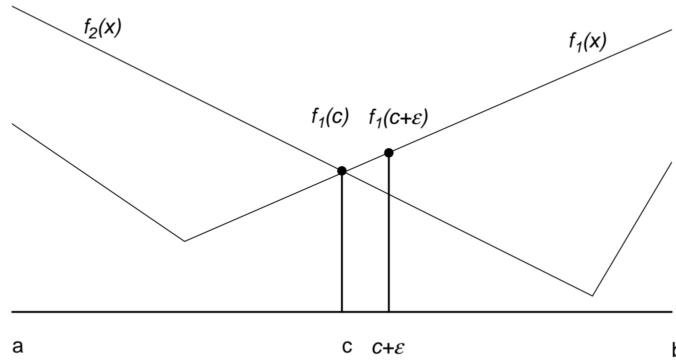

Figure 6.6. Two f-evaluations give information.

algorithm *MinBisect* $(f, a, b, tol, maxits)$

---

for $k := 1$ **to** $maxits$ **do**
  $m := a + (b - a)/2$
  if $f(m - \epsilon) < f(m + \epsilon)$ **then**
    $b := m + \epsilon$
  else
    $a := m - \epsilon$
  endif
  if $|a - b| \leq tol$ **then return** m
endfor
endalg {MinBisect}

Notice that the function values are not saved between iterations. Each is either outside the new interval or at either end of it and so provide no useful information for the next iteration of this algorithm.

Analysis of MinBisect. The algorithm performs two function evaluations per iteration. It is obvious that the interval is halved (contracted) each iteration. Hence

$$e_k = \frac{1}{2}\, e_{k-1}.$$

Thus the MinBisect Algorithm has order 1 or linear convergence and it gains 1 bit of precision for each iteration. The error after k iterations is $e_k = 2^{-k}|b-a|$. (See the analysis of Bisect in Chapter 4 for further details).
We note that if $f'(x)$ is available then the minimization problem may be replaced with finding a zero of $f'(x)$, at a cost of one derivative evaluation per iteration. Alternatively we could replace the **if**-test in the algorithm above with the test

if $f'(m) > 0$ **then** etc.

Is MinBisect Optimal?. This question is not well-posed and we need to be more specific. If we are given an interval $[a, b]$ and are allowed only two evaluations then is *MinBisect* optimal? Placing

the evaluations slightly apart in the middle of the interval reduces the interval by approximately 1/2. If we place them at $a + \theta(b - a),\ 0 \le \theta \le 1$ then if $\theta < 1/2$ or $\theta > 1/2$ then this is not as good as *MinBisect*. However, let us assume we have 4 function evaluations at our disposal. Obviously *MinBisect* reduces the interval to 1/4 but placing the evaluations in the positions and order shown in Figure 6.7 reduces the interval to $[3, 4]$, which is 1/5 of the original. Clearly *MinBisect* is not optimal if we have more than two evaluations at our disposal.

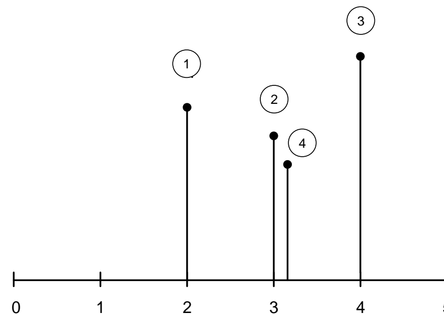

Figure 6.7. Four f-evaluations give a reduction of 1/5.

6.2.4 Fibonacci Algorithm.

The fact that we can do better than *MinBisect* with four evaluations poses the natural question : what is the best we can do with n evaluations? Keifer (1953) studied this and other optimum search strategies. The answer involves the *Fibonacci Numbers* which we have seen before.
The Fibonacci numbers are defined as : $F_0 = F_1 = 1$, and $F_{k+2} = F_{k+1} + F_k$. The Fibonacci numbers for $k = 0, 1, \ldots, 10$ are

k	0	1	2	3	4	5	6	7	8	9	10
F_k	1	1	2	3	5	8	13	21	34	55	89

The example in Figure 6.8 shows the working of the Fibonacci Search Algorithm.
It should be noted that this algorithm includes, as a special case, the *MinBisect* algorithm, i.e., when $n = 2$. The algorithm divides the interval into $F_2 = 2$ equal subintervals and places evaluations at $F_0 - \epsilon$ and $F_1 + \epsilon$.

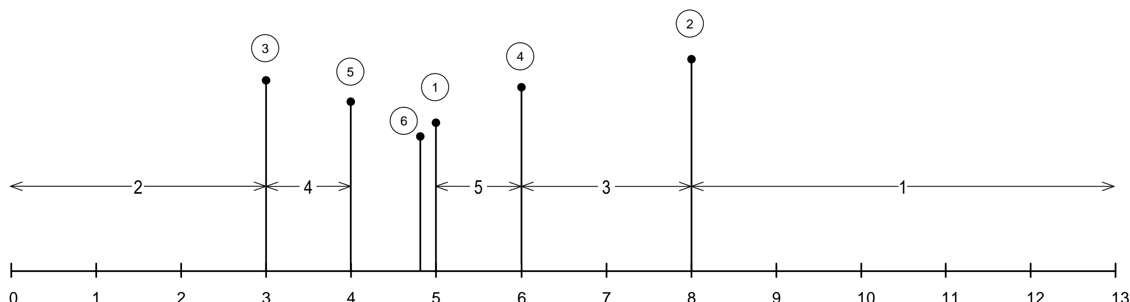

Figure 6.8. Six f-evaluations give a reduction of 1/13.

Analysis of Fibonacci Search. Given n evaluations the Fibonacci algorithm reduces the initial interval to $(b-a)/F_n$. It can be shown that a closed-form expression for F_n is

$$\begin{aligned} F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) &= \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] \\ &= \frac{1}{\sqrt{5}}\,(1.618^n - 0.618^n) \\ &\approx \frac{1}{\sqrt{5}}\,1.618^n,\ n \to \infty. \end{aligned}$$

This is a good approximation to F_n for $n > 5$ and so we can calculate the reduction using this simple formula. The ratio of the lengths of successive intervals is

$$\frac{F_{n-k}}{F_{n-k+1}},\ k = 1, 2, \ldots, n.$$

As $n \to \infty$ this ratio is $1/1.618 = 0.618$, the *Golden Ratio*, or section.

6.2.5 Golden Section Algorithm.

Although the Fibonacci Search algorithm is optimal for n evaluations, we usually want an iterative algorithm to continue until the initial interval has been reduced to some pre-determined tolerance, and not a pre-determined number of evaluations.
We get such an algorithm from Fibonacci Search by imagining we have an infinite number of evaluations. This gives a reduction to $0.618 = \lambda$ at each iteration, as we saw above. The Golden Section λ has the property

$$\frac{1}{1+\lambda} = \lambda, \text{ or } \lambda^2 + \lambda = 1.$$

We start the algorithm by placing two evaluations at

$$x_L = a + (1-\lambda)(b-a) \quad \text{and} \quad x_U = a + \lambda(b-a)$$

.
For simplicity let us assume that $a = 0$ and $b = 1$. Then the first step is as shown in Figure 6.7 (a)

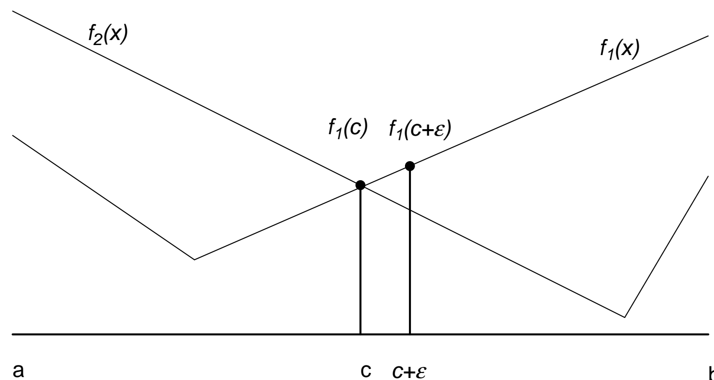

Figure 6.9. Golden Section Search .

From the figure it can be seen that the reduced interval is $[0, \lambda]$.
We place two more evaluations in this interval at the points

$$x_L = 0 + (1-\lambda)(\lambda - 0) = \lambda(1-\lambda) \quad \text{and} \quad x_U = \lambda(\lambda - 0) = \lambda^2 = 1 - \lambda.$$

Thus we see that one of the two evaluations is already in the reduced interval. This is shown in Figure 6.7(b).

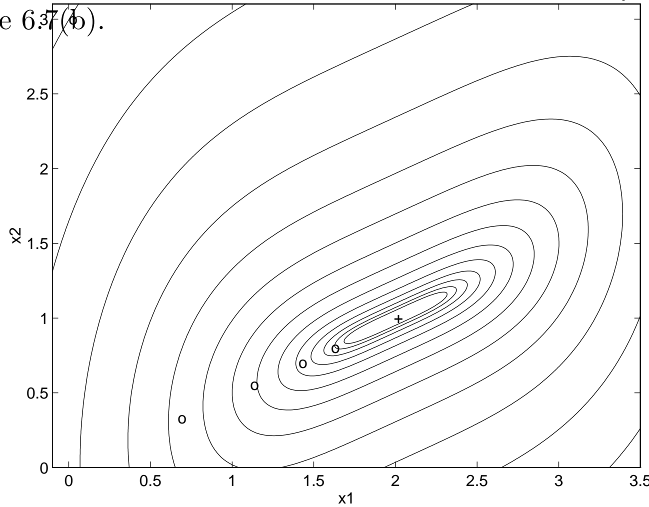

Figure 6.10. Relationship between Sub-intervals.

The two possibilities at each step of the algorithm is shown in Figure 6.8. We see that in either case the new interval already has an evaluation in it which is used at the next step. Thus only one evaluation is needed per iteration.

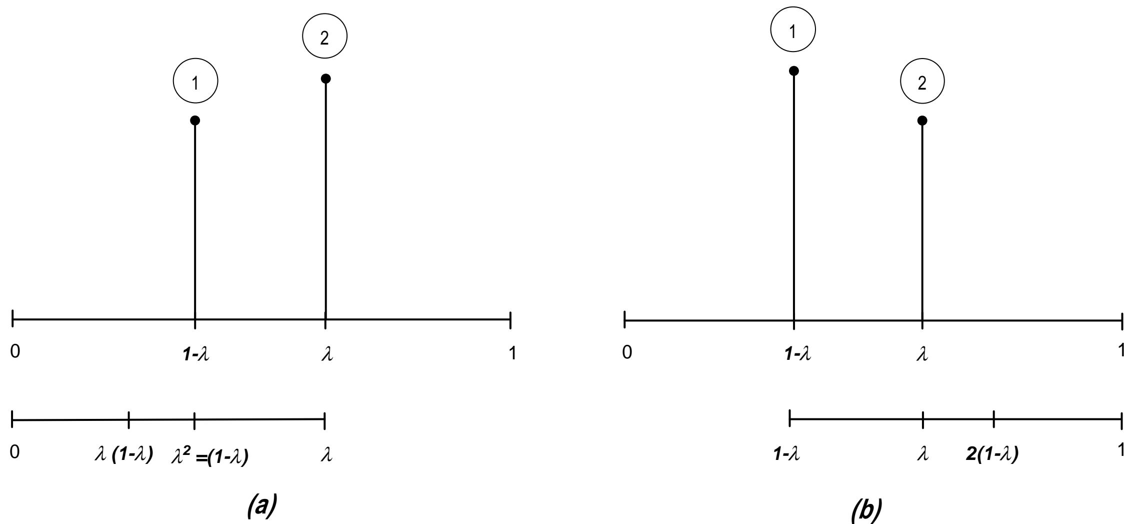

Figure 6.11. General Step.

The formal statement of the algorithm is as follows :

```
algorithm   GoldenSection (f, a, b, tol, maxits)
-----------------------------------------------------------
        λ := (1 + √5)/2
        x_L := a + (1 − λ)(b − a)
        x_U := a + (λ(b − a)
        f_L := f(x_L)
        f_U := f(x_U)
        for  k := 1  to  maxits  do
            if (f_L < f_U) then
                b := x_U
                x_U := x_L
                f_U := f_L
                x_L := a + (1 − λ)(b − a)
                f_L := f(x_L)
            else
                a := x_L
                x_L := x_U
                f_L := f_U
                x_U := a + λ(b − a)
                f_U := f(x_U)
            endif
            if |a − b| ≤ tol  then  return  (a, b)
        endfor { k }
endalg {GoldenSection}
```

Analysis of Golden Section. It can be seen that only one function evaluation is used at each iteration and

$$e_k = 0.618 e_{k-1}.$$

Note that this is better than *MinBisect* which reduces the interval by 0.5 for two evaluations whereas *Golden Section* gives a reduction of $0.618^2 = 0.382$ for the same work.

Example : Golden Section. This is the *Golden Section Search* on the function

$$f(x) = \sqrt{(x-2)^2+1}, \quad \text{with } [a,b] = [-3,10].$$

k	a	b	x_L	x_U	f_L	f_U
1	−3.00000	10.00000	1.96600	5.03400	1.00058	3.19455
2	−3.00000	5.03400	0.06899	1.96600	2.17458	1.00058
3	0.06899	5.03400	1.96600	3.13737	1.00058	1.51446
4	0.06899	3.13737	1.24111	1.96600	1.25536	1.00058
5	1.24111	3.13737	1.96600	2.41300	1.00058	1.08193
6	1.24111	2.41300	1.68877	1.96600	1.04731	1.00058
7	1.68877	2.41300	1.96600	2.13634	1.00058	1.00925
8	1.68877	2.13634	1.85974	1.96600	1.00979	1.00058
9	1.85974	2.13634	1.96600	2.03068	1.00058	1.00047

Compare this with the *Fibonacci Search*

k	a	b	x_L	x_U	f_L	f_U
1	−3	10	2	5	1.000	3.162
2	−3	5	0	2	2.236	1.000
3	0	5	2	3	1.000	1.414
4	0	3	1	2	1.414	1.000
5	1	3	1.966	2.004	1.001	1.082

This shows that the *Golden Section Search* algorithm is close to optimal.

6.2.6 Quadratic Algorithms.

These algorithms replace the function $f(x)$ with the polynomial

$$f(x) \approx p_2(x) = a_0 + a_1 x + a_2 x^2.$$

The first derivative of $p_2(x)$ is $a_1 + 2a_2 x = 0$ at a minimum and so we get the approximate minimizer $\hat{x}^* = -a_1/2a_2$.
Thus at each iteration of a quadratic algorithm we have 3 points, (x_0, f_0) , (x_1, f_1) , (x_2, f_2), and we solve the following equations for a_0 , a_1 , a_2 :

$$\begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix}.$$

The next approximation is then $x_{k+1} = -a_1/2a_2$.

6.2.7 Poly-Algorithms.

Brent (1973) has written an algorithm similar to *Zeroin* called *Fmin* which uses a combination of

- Golden Section Search
- Quadratic Search

This is probably the most robust 1-dimensional minimum-finder available.

6.3 N-DIMENSIONAL MINIMIZATION WITHOUT DERIVATIVES

These algorithms use function evaluations only, without any recourse to derivative information. These algorithms are necessary when derivatives are not available, or difficult to calculate, or if the function has discontinuities in its derivatives.

Example : *The 1-Median in the Plane Problem.* Find the location (x, y) of a central office to serve k clusters of customers at locations (x_i, y_i) with w_i customers at each location so that the total distance travelled by all customers is minimized, i.e.,

$$\min_{x,y} D(x,y) = \sum_{i=1}^{k} w_i \sqrt{(x - x_i)^2 + (y - y_i)^2}.$$

This 2-dimensional function $D(x, y)$ has no derivatives at the points where $x = x_i$ or $y = y_i$.

6.3.1 Basic Descent Algorithm.

This is a prototype for all minimization algorithms. It reduces an n-dimensional search for a minimum to a sequence of 1-directional searches along chosen directions. In the algorithm below d_k is a direction vector with $\|d_k\| = 1$ and s_k is a scalar. Thus, for a given d_k and x_k, $f(x_k + s_k d_k)$ is a function of the scalar variable s_k, called the *step length.*

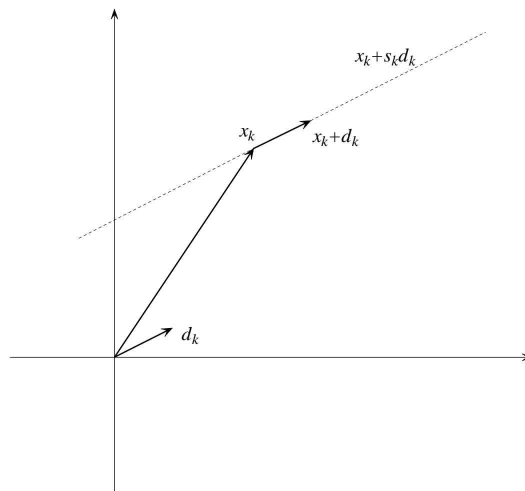

Figure 6.12. Searching in 1 direction.

algorithm *Basic-Descent* (f, x_0)

---

while not converged **do**
1. Choose a descent direction d_k
2. Find s_k that minimizes $f(x_k + s_k d_k)$
3. $x_{k+1} := x_k + s_k d_k$

endwhile
return x_k

endalg {Basic-Descent}

6.3.2 Cyclic Coordinate Descent Algorithm.

This is a very simple, but not very effective, algorithm. It chooses each coordinate direction in turn and minimizes the function in that direction. In effect it minimizes $f(x_1, x_2, \ldots, x_n)$ with respect to x_k at each iteration k. Thus it chooses $d_k = e_k$ where e_k is the kth standard basis vector and the algorithm 'cycles' through the coordinate directions $e_1, e_2, \ldots, e_n, e_1, e_2, \ldots$.

algorithm *Cyclic-Coordinate-Descent* $(f, x_1, n, tol, maxits)$

$i := 1$
for $k := 1$ **to** maxits **do**
$d_k := e_i$
Find s_k that minimizes $f(x_k + s_k d_k)$
$x_{k+1} := x_k + s_k d_k$
if converged **then return** x_{k+1}
$i := (i \bmod n) + 1$
endfor

endalg {Cyclic-Coordinate-Descent}

Example : Two dimensions. Let us minimize the function

$$f(x) = x_1^2 + x_2^2 - 4, \quad \text{starting at} \quad x_0 = (4, 4).$$

Iteration 1 :

$$d_1 = (1, 0), \quad \text{and} \quad x_1 + s_1 d_1 = (4, 4) + s_1(1, 0) = (4 + s_1, 4).$$

$$f(x_1 + s_1 d_1) = (4 + s_1)^2 + (4)^2 - 4 = s_1^2 + 8s_1 + 28$$

This has a minimum at $s_1 = -4$ and we have $x_2 = (4, 4) - 4(1, 0) = (0, 4)$.

Iteration 2 :

$$d_2 = (0,1), \quad \text{and} \quad x_2 + s_2 d_2 = (0,4) + s_1(0,1) = (0, 4 + s_2).$$

$$f(x_2 + s_2 d_2) = 0^2 + (4 + s_2)^2 - 4 = s_2^2 + 8s_2 + 12$$

This has a minimum at $s_1 = -4$ and we have $x_2 = (0,4) - 4(0,1) = (0,0)$.
This is in fact the minimum, obtained after two iterations. This is because the function is very simple and regular. It can be seen that the function with elliptical contours causes the method to take many steps and it is unlikely to converge for more complicated functions. The function $f(x) = x_1^2 + x_2^2 - 4$ is *separable* in that there are no cross-product terms such as $x_1 x_2$ and the algorithm works well on such functions. Where there is interaction between variables the algorithm does badly. This type of behavior can be seen in adjusting a television to get the optimum picture. The picture is a function of the vector $x = (colour, brightness, contrast)$. Most people adjust these one-at-a-time but need to cycle through these adjustments because they affect each other. This method is exactly analogous to cyclic coordinate descent.

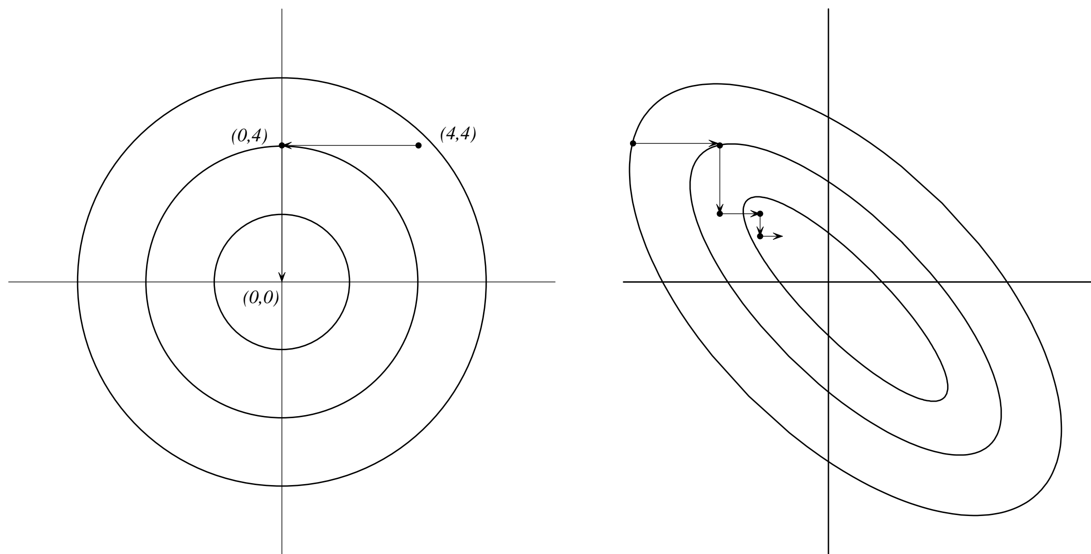

Figure 6.13. Cyclic Coordinate Descent.

6.3.3 The Polytope Algorithm.

This is a simple method that has been found to be quite useful for minimizing functions of low dimension ($n < 6$). It has been used extensively in the chemical industry to optimize plant performance.

The Regular Polytope Algorithm. This was proposed by Spendley, Hext and Himsworth (1962) and they called it the *Simplex Method*. To avoid confusion with its much more famous relative, we will call it the Regular Polytope Algorithm. This was one of many sequential algorithms applied in the chemical industry under the general heading of *Evolutionary Operation*. This phrase was coined by G.E.P. Box in 1957 to denote sequential methods 'for increasing industrial productivity'. At this time he or (his relative?) M.J. Box, or both were working for Imperial Chemicals Industries, U.K.

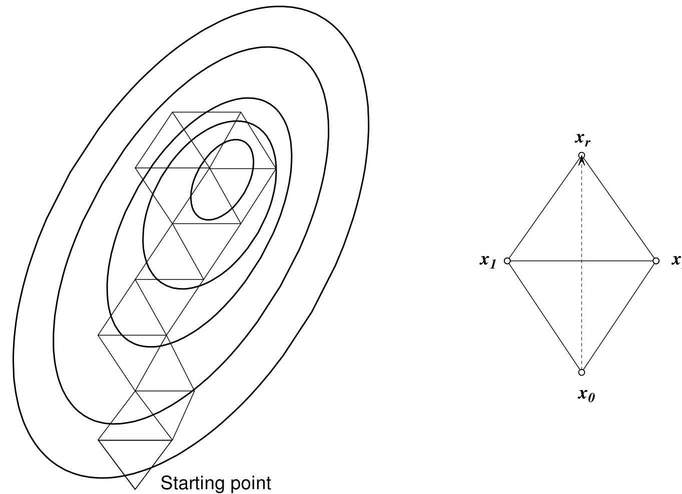

Figure 6.14. The Regular Polytope Method.

6.4 N-DIMENSIONAL MINIMIZATION WITH DERIVATIVES

These algorithms use first and second derivatives to guide the search for a minimum. The following conditions are *sufficient* for x^* to be a local minimizer of $f(x)$:

$$\|g(x^*)\| = 0, \quad \text{and} \quad H(x^*) \text{ is positive definite, i.e.,} \quad x^T H(x^*) x > 0.$$

The derivative information is used to choose a 'good' direction d_k in step 1 of the Basic Descent Algorithm.

6.4.1 The Method of Steepest Descent.

This is probably the oldest (formal) optimization algorithm and was invented by Cauchy in 1847. The essential idea of the algorithm is this : from the point x_k go in the direction in which the function decreases the fastest. That is, go in the direction that is steepest.
We can derive this direction formally as follows : using the first two terms of the Taylor Series expansion about x_k we get

$$f(x) \approx f(x_k) + g^T(x_k)(x - x_k), \quad \text{or} \quad \Delta f_k = g^T(x_k)\Delta x_k.$$

We wish to make Δf_k to be as big a decrease as possible, i.e., large and negative. We get this by choosing $\Delta x_k = -g(x_k)$ and so we choose

$$d_k = -g_n(x_k) = -g(x_k)/\|g(x_k)\|.$$

The steepest descent algorithm is as follows :

algorithm *Steepest-Descent* $(f, g, x_1, n, tol, maxits)$

---

for $k := 1$ **to** maxits **do**
  Calculate $g(x_k)$
  $d_k := -g(x_k)/\|g(x_k)\|$
  Find s_k that minimizes $f(x_k + s_k d_k)$
  $x_{k+1} := x_k + s_k d_k$
  if converged **then return** x_{k+1}
endfor

endalg {Steepest-Descent}

Example : *Steepest Descent.* Minimize Rosenbrock's function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \quad \text{starting at} \quad x_0 = (2, 2).$$

First we find

$$g(x) = \begin{bmatrix} \partial f(x)/\partial x_1 \\ \partial f(x)/\partial x_2 \end{bmatrix} = \begin{bmatrix} -400(x_1 x_2 - x_1^3) - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{bmatrix}.$$

Iteration 1 :

$$x_1 = (2, 2),\ f(x_1) = 401,\ g(x_1) = (1602, -400)^T,$$
$$d_1 = -g(x_1)/\|g(x_1)\| = -(0.9702, -0.2422)^T.$$

$$x_1 + s_1 d_1 = (2, 2) - s_1(0.9702, -0.2422) = (2 - 0.9702 s_1, 2 + 0.2422 s_1).$$
$$f(x_1 + s_1 d_1) = 100(-2 - 1.61982 s_1 - 0.94135 s_1^2)^2 + (-1 + 0.9702 s_1)^2.$$

This has a minimum at $s_1 = 0.555$ and we have $x_2 = (1.4615, 2.1345)$.

Iteration 2 :

$$x_2 = (1.4615, 2.1345),\ f(x_2) = 0.213,\ g(x_2) = (0.922, 0.0)^T,$$
$$d_2 = -g(x_2)/\|g(x_2)\| = -(1.0, 0.0)^T.$$

$$x_2 + s_2 d_2 = (1.4615,\ 2.1345) - s_1(1, 0) = (1.4615 - s_1,\ 2.1345).$$
$$f(x_2 + s_2 d_1) = 100(2.1345 - (1.4615 - s_2)^2)^2 + (0.4614 - s_2)^2.$$

This has a minimum at $s_2 = 0.001$ and we have $x_3 = (1.4605, 2.1345)$ and $f(x_3) = 0.212$.
At this point the method gets stuck unless we resort to higher precision.

This is typical of the Steepest Descent algorithm. Indeed, it can be shown (see GMW 82, page 103) that even for simple quadratic functions the method gets stuck. For example, if f is the quadratic

$$f(x) = c^T x + \frac{1}{2} x^T G x,$$

where G is a symmetric positive-definite matrix, then

$$f(x_{k+1}) - f(x^*) = \Delta f_{k+1} \approx \frac{(\text{cond}^*(G) - 1)^2}{(\text{cond}^*(G) + 1)^2} \Delta f_k.$$

If $\text{cond}^*(G) = 50$ then the reduction in the function error at each step is just $(49/51)^2 = 0.923$, i.e., if $\Delta f_k = 1$ then $\Delta f_{k+1} = 0.923$, $\Delta f_{k+2} = 0.852$, $\Delta f_{k+3} = 0.786$. This shows that for a very mildly ill-conditioned G the convergence of $\{f_k\}$ to a minimum is linear and very slow. Furthermore, if this algorithm performs badly on a simple quadratic function it is unlikely to do well on more complicated functions.

6.4.2 Newton's Algorithm.

Newton's method for minimizing a function of one variable $f(x)$ is obtained by using the first three terms of the Taylor series expansion about x_k :

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2} f''(x_k)(x - x_k)^2.$$

This is a quadratic in x that has a minimum x^* at

$$x^* = x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}.$$

This is the same as Newton's zero-finding algorithm except that f and f' have been replaced by f' and f'', respectively.
In an analogous manner we get Newton's method in n dimensions by replacing $f(x)$ by its 2nd-order Taylor approximation :

$$f(x) \approx f(x_k) + g^T(x_k)(x - x_k) + \frac{1}{2}(x - x_k)^T H(x_k)(x - x_k).$$

This is a quadratic in $x \in \mathcal{R}^n$ that has a minimum x^* at

$$x^* = x_{k+1} = x_k - H^{-1}(x_k)g(x_k) \quad \text{or} \quad \Delta x_k = -H^{-1}(x_k)g(x_k).$$

The last expression shows that for the Newton method the descent direction is $d_k = -H^{-1}(x_k)g(x_k)$. In the 'pure' form of Newton's algorithm we use a fixed step length $s_k = 1$. Hence, no line search (one-directional minimization) is performed so that step 2 of the basic descent algorithm is omitted. The main work is the calculation of d_k. It may appear that this requires the calculation of $H^{-1}(x_k)$. However, we re-arrange the expression and solve the following equation for d_k :

$$H(x_k)d_k = g(x_k).$$

We can now write the pure form of Newton's algorithm.

algorithm *Pure-Newton*$(f, g, H, x_1, n, tol, maxits)$

```
for k := 1 to maxits do
   Calculate g(x_k)
   Calculate H(x_k)
   Solve H(x_k)d_k = -g(x_k) for d_k
   x_{k+1} := x_k + d_k
   if converged then return x_{k+1}
endfor

endalg {Pure-Newton}
```

Analysis of Newton's Algorithm. The convergence analysis is difficult and so we merely assert that if $f(x)$ is sufficiently 'nice' then Newton's algorithm has 2nd-order convergence.
We are more interested in the computational burden because it is quite substantial for each iteration. There are three main computations per iteration :

1. Calculate $g(x_k)$. This requires n function evaluations.
2. Calculate $H(x_k)$. This requires $n^2/2$ function evaluations (H is symmetric).
3. Solve $H(x_k)d_k = g(x_k)$ for d_k. This requires $O(n^3)$ flops.

Example : Newton's Algorithm. Minimize

$$(x_1 - 2)^4 + (x_1 - 2x_2)^2.$$

$$f(x_{old}) = (-2)^4 + (-2 \times 3)^2 = 16 + 36 = 52.$$

$$g(x) = \begin{bmatrix} 4(x_1 - 2)^3 + 2(x_1 - 2x_2) \\ -4(x_1 - 2x_2) \end{bmatrix} = \begin{bmatrix} -44 \\ 24 \end{bmatrix}$$

and

$$H(x) = \begin{bmatrix} 12(x_1 - 2)^2 + 2 & -4 \\ -4 & +8 \end{bmatrix} = \begin{bmatrix} 50 & -4 \\ -4 & +8 \end{bmatrix}$$

We must now solve $H(x_{old})\delta x = -g(x_{old})$.
That is

$$\begin{bmatrix} 50 & -4 \\ -4 & +8 \end{bmatrix} \begin{bmatrix} \delta x_1 \\ \delta x_2 \end{bmatrix} = \begin{bmatrix} 44 \\ -24 \end{bmatrix}$$

This gives

$$\delta x = [0.667 \ \ -2.667]^T, \quad x_{new} = x_{old} + \delta x = [0.667 \ 0.333]^T, \quad f(x_{new}) = 3.161.$$

Here is a table of the first three iterations

Pure Newton Method

k	x_k	$f(x_k)$	$g(x_k)$	$H(x_k)$	$\delta x_k = -H^{-1}g$	x_{k+1}
0	$\begin{bmatrix}0\\3\end{bmatrix}$	52	$\begin{bmatrix}-44\\24\end{bmatrix}$	$\begin{bmatrix}50 & -4\\-4 & +8\end{bmatrix}$	$\begin{bmatrix}-0.667\\-2.667\end{bmatrix}$	$\begin{bmatrix}0.667\\0.333\end{bmatrix}$
1	$\begin{bmatrix}0.667\\0.333\end{bmatrix}$	3.161	$\begin{bmatrix}-9.42\\0\end{bmatrix}$	$\begin{bmatrix}23.3 & -4\\-4 & +8\end{bmatrix}$	$\begin{bmatrix}-0.444\\-0.222\end{bmatrix}$	$\begin{bmatrix}1.111\\0.555\end{bmatrix}$
2	$\begin{bmatrix}1.111\\0.555\end{bmatrix}$	0.624	$\begin{bmatrix}-2.81\\0\end{bmatrix}$	$\begin{bmatrix}11.5 & -4\\-4 & +8\end{bmatrix}$	$\begin{bmatrix}-0.296\\-0.148\end{bmatrix}$	$\begin{bmatrix}1.41\\0.707\end{bmatrix}$

The algorithm converges, after about 10 iterations to

$$x = \begin{bmatrix}2\\1\end{bmatrix}, \quad f(x) = 0, \quad g(x) = \begin{bmatrix}0\\0\end{bmatrix} \quad \text{and} \quad H(x) = \begin{bmatrix}2 & -4\\-4 & +8\end{bmatrix}$$

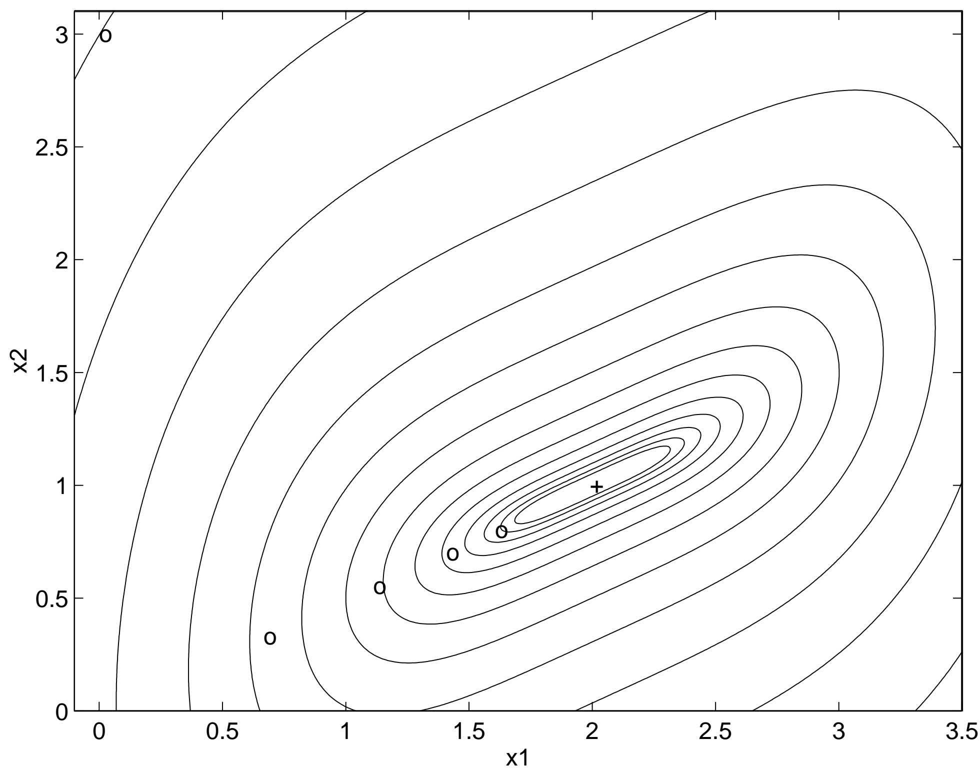

Figure 6.15. Newton's Method on $f(x) = (x_1 - 2)^4 + (x_1 - 2x_2)^2$.