CHAPTER 9

Optimization and Nonlinear Least Squares

9.1. Introduction

We will cover the material in sections 9.2 and 9.4. Read sections 9.3 and 9.5 by yourself.

As already outlined in chapter 7, the goal of the present chapter is to find a local minimum of a function $F(x)$ (a single function of several variables). This problem has many connections to the problem of finding a zero of a nonlinear function.

9.1.1. Accuracy. One “feature” of minimum-finding subroutines is that they can only find the solution to lower precision. Consider this in one dimension:

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

If $\bar{x}$ is a local minimum, then $f'(\bar{x}) = 0$, so

$$f(x) - f(\bar{x}) \approx C(x - \bar{x})^2$$

In order to locate the minimum, the relative change between $f(x)$ and $f(\bar{x})$ must be at least the machine $\epsilon$. But then the relative change between $x$ and $\bar{x}$ can only be $\sqrt{\epsilon}$, assuming $c \approx 1$. In single precision arithmetic, $\epsilon \approx 10^{-7}$, so $\sqrt{\epsilon} \approx 10^{-3}$ or $10^{-4}$. This means we can only find a minimum to about 3 or 4 decimal places.

This becomes apparent when you solve the same problem in several ways. The answers you get only agree to about 3 decimals. Watch for this in the examples below.

9.2. One-Dimensional Optimization

We want to find a local minimum of a function $F(x)$.

9.2.1. Golden Section Search. The method outlined in this section is similar to the bisection method in spirit: we start with an interval we think contains a minimum, and keep making the interval smaller. It is slow but reliable, and it requires no derivatives.

This method only works reliably under the assumption that we have an interval where the function is unimodal. This means that there is exactly one minimum $\bar{x}$ in the interval $[a, b]$ we are currently considering, and $F$ is decreasing in $[a, \bar{x}]$ and increasing in $[\bar{x}, b]$. (See figure 9.4 in the book).

We start with a given interval $[a, b]$. Now we pick two points $a < x_1 < x_2 < b$ inside the interval. Assume $F(x_1) > F(x_2)$. Then $F$ is decreasing at least part of the way between $x_1$ and $x_2$. Since $F$ is always increasing to the right of $\bar{x}$, $\bar{x}$ cannot be any smaller than $x_1$. Thus, the minimum $\bar{x}$ is in $[x_1, b]$. Likewise, if $F(x_1) < F(x_2)$, then $\bar{x} < x_2$, and the minimum must be in $[a, x_2]$.

What is the optimal way to choose the subdivision points? If you know how many steps you want to take, the best method is called Fibonacci search. If you don’t know, the best method is the golden section search. Look at the book for more details on the derivation.

In golden section search, we put one subdivision point at $(\sqrt{5} - 1)/2 \approx 0.6180$ of the interval width. This number is called the golden ratio. The other point goes symmetrically, at about 0.3820 of the interval width. With this placement, we can always re-use one of the points at the next step. Remember that the goal is to minimize the number of function evaluations necessary.

At every step, the interval becomes smaller by a factor of 0.618, so this method is linearly convergent.

Example: Find a local minimum of the function

$$F(x) = e^x + \frac{1}{x}.$$
With the initial interval \([0.1, 1]\), we get successive intervals
\[
[0.1000000, 1.0000000] \\
[0.4437694, 1.0000000] \\
[0.4437694, 0.7875388] \\
[0.5750777, 0.7875388] \\
[0.6562306, 0.7875388] \\
... \\
[0.7037065, 0.7037098] \\
[0.7037065, 0.7037086] \\
[0.7037073, 0.7037086] \\
[0.7037078, 0.7037086]
\]
with a best guess of \(x = 0.7037081\), \(F(x) = 3.442277\).

The true answer to 10 decimals is \(x = 0.703467422\), \(F(x) = 3.442277294\). Notice how \(F(x)\) is accurate to 7 decimals, while \(x\) is only accurate to 3 decimals. □

**9.2.2. Newton’s Method.** We convert the minimization problem to a zero-finding problem. Thus, we want to solve
\[
F'(x) = 0.
\]
If we use Newton’s method (for finding zeros of a function) for that, we get the iteration
\[
x_{n+1} = x_n - \frac{F'(x_n)}{F''(x_n)}.
\]
This is also called Newton’s method (for optimization). We now have Newton’s method for finding zeros and Newton’s method for optimization, which is based on the first one. Try not to confuse the two.

**Remark:** Of course, there is also a Newton’s method for polynomial interpolation, which has nothing to do with the present discussion at all.

**Example:** Minimize the function
\[
F(x) = e^x + \frac{1}{x}
\]
We find with initial guess \(x_0 = 1\),
\[
\begin{align*}
x_0 &= 1.0 & F(x_0) &= 3.718282 \\
x_1 &= 0.6358247 & F(x_1) &= 3.461339 \\
x_2 &= 0.6963254 & F(x_2) &= 3.442477 \\
x_3 &= 0.7033929 & F(x_3) &= 3.442277 \\
x_4 &= 0.7034674 & F(x_4) &= 3.442277
\end{align*}
\]
and no further changes. This time, the answer is correct all the way, but that is because this is a zero-finding problem now, not a minimization problem any more. I can only get higher accuracy because I know the exact derivative of \(F\).

Note that \(F(x_3)\) already equals \(F(x_4)\), even though \(x_3\) and \(x_4\) are different. □

Newton’s method for optimization has the same advantages and disadvantages as Newton’s method for finding zeros:

- It converges very fast (quadratically).
- It may run into problems (infinite loops, division by zero, etc.)
- Its main disadvantage is that it requires derivatives. Two derivatives now, even.
9.2.3. Successive Parabolic Interpolation. There is another way to derive Newton’s method which suggests a new method.

Recall that Newton’s method for finding zeros can be interpreted as follows: $x_0$ is an initial guess. We compute the tangent to the graph of $f(x)$ at the point $(x_0, f(x_0))$ and find the zero of the tangent. That is our $x_1$. Then we put a tangent at the point $(x_1, f(x_1))$, and so on. The equation of the tangent at $(x_i, f(x_i))$ is a truncated Taylor series at $x_i$

$$t(x) = f(x_i) + f'(x_i)(x - x_i)$$

If we want to avoid derivatives, we can replace the tangent by a secant through $x_{i-1}$ and $x_i$, and that leads to the secant method. The secant method converges a little more slowly than Newton’s method, but does not require derivatives.

A similar interpretation is possible for Newton’s method for optimization. Here, we need to go one step further in the Taylor series: We replace $F(x_i)$ by the tangent parabola at $(x_i, F(x_i))$

$$p(x) = F(x_i) + F'(x_i)(x - x_i) + \frac{F''(x_i)}{2}(x - x_i)^2$$

and calculate the minimum of the parabola. The result is the Newton’s method for minimization

$$x_{n+1} = x_n - \frac{F'(x_n)}{F''(x_n)}.$$ 

To avoid derivatives, we could put a parabola through the last three points $x_j$ and find the minimum of the parabola. Essentially, we are replacing the derivatives in Newton’s formula by difference quotients. The resulting method is called \textit{successive parabolic interpolation}.

It has properties similar to the secant method: it does not converge with second order, like Newton’s method, but it converges with better than first order.

The formulas are sort of messy, so we will not go into the details.

9.2.4. Summary of One-Dimensional Methods. We have considered three methods: Golden section search, Newton’s method and Successive parabolic interpolation. They correspond more or less to Bisection, Newton’s method and Secant method for the problem of finding zeros.

That means

- Golden Section Search, like Bisection, is slow but reliable and will converge in a certain number of steps which can be calculated in advance.
- Newton’s method is fast and is the method of choice if you have a sufficient number of derivatives available.
- Successive Parabolic Interpolation is a form of Newton’s method with derivatives replaced by difference quotients. It is slower than Newton’s method, but still converges quite fast.

The best general purpose method is a combination of Golden Section and Successive Parabolic Interpolation: Use Parabolic Interpolation most of the time. If it gives crazy answers or tries to divide by zero, use Golden Section for a step or two. This is what routine \textsc{fmin} does.

9.3. Subroutine \textsc{fmin}

Read this section on your own.

9.4. Optimization in Many Dimensions

Most higher-dimensional methods follow the same scheme:

- Pick an initial guess $x_0$ and an initial search direction $u_0$.
- Let $x_1$ be the minimum of $F$ in direction $u_0$, starting at $x_0$. Mathematically speaking, we define \[ h(\lambda) = F(x_0 + \lambda u_0) \]
so $h$ is a function of the single variable $\lambda$. Think of it as a one-dimensional slice through the surface $F$. We find the $\lambda$ for which this function is a minimum, and then

$$x_1 = x_0 + \lambda u_0.$$ 

This step is called the \textit{line search}. Here is where we need one of the one-dimensional minimizers, like the golden section search.
• Pick a new search direction $u_1$.
• Let $x_2$ be the minimum of $F$ in direction $u_1$, starting at $x_1$.

Collectively, these methods are known as direction set methods. Different methods are distinguished mainly by how the search directions $u_i$ are picked. The choice of one-dimensional minimizer makes a minor difference, and there are other tricks you can use (for example, you may overshoot the one-dimensional minimum on purpose at every step). We will ignore all that and just concentrate on the choice of $u_i$. That is where the biggest difference between the methods lies.

First we need to recall a couple more things from calculus.

### 9.4.1. Surfaces and Level Sets

We are given a function $F(x)$, where $x$ is an $n$-vector. We can think of the graph of $F$ as a surface in $(n + 1)$-dimensional space. A level surface of $F$ is the set of all points

$$\{x : F(x) = c\}$$

where $c$ is some constant. For every $c$, we get a different level curve. (The isoclines on a survey map are level curves, for example). Near a local minimum, the level surfaces always look like nested ellipsoids.

The gradient of $F$ is the vector

$$\nabla F = \begin{pmatrix} \frac{\partial F}{\partial x_1} \\ \vdots \\ \frac{\partial F}{\partial x_n} \end{pmatrix}$$

The gradient indicates the direction in which $F$ changes the most. Along the level surfaces, $F$ does not change at all. The gradient is always perpendicular to the level surfaces.

When we are doing our line searches, $F$ is increasing or decreasing as long as we intersect the level surfaces at an angle. $F$ has a stationary point when we are tangent to a level surface. The place where the line minimum occurs is always a stationary point. Thus

$$u_{i-1} \cdot \nabla F(x_i) = 0.$$ 

We will need this formula later.

### 9.4.2. Coordinate Search

I don’t know if this method even has a name, so I made one up. A very simple-minded approach would be to use the coordinate directions as search directions:

$$u_0 = e_1$$
$$u_1 = e_2$$
$$\ldots$$
$$u_{n-1} = e_n$$
$$u_n = e_1$$
$$\ldots,$$

where

$$e_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

and so on. In other words: we first find a minimum in the $e_1$ direction, while keeping all the other coordinates fixed. Then we find a minimum in the $e_2$ direction, and so on.

This will work, but will be slow.

**Example:** The function

$$F(x) = F(x, y) = 10(x - y)^2 + (1 - x)^4$$
has a local minimum at (1, 1). Using $x_0 = (1.25, 1.25)$ and coordinate search, we find

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$F(x_i)$</th>
<th>$u_{i+1} = -\nabla F(x_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.2500000, 1.2500000)</td>
<td>0.0039062500</td>
<td>(-0.0625000, 0.0000000)</td>
</tr>
<tr>
<td>(1.2469850, 1.2500000)</td>
<td>0.0038120960</td>
<td>(0.0000498, -0.0603151)</td>
</tr>
<tr>
<td>(1.2469850, 1.2469890)</td>
<td>0.0037212200</td>
<td>(-0.0602052, -0.0000620)</td>
</tr>
<tr>
<td>(1.2440800, 1.2469890)</td>
<td>0.0036337960</td>
<td>(0.0000185, -0.0581813)</td>
</tr>
<tr>
<td>(1.2440800, 1.2440790)</td>
<td>0.0035491690</td>
<td>(-0.0581253, -0.0000381)</td>
</tr>
<tr>
<td>(1.2412700, 1.2412690)</td>
<td>0.003385690</td>
<td>(0.0001565, -0.0563383)</td>
</tr>
</tbody>
</table>

The gap represents about 400 iterations. As you can see, it works, but the method is awfully slow. □

### 9.4.3. Steepest Descent

A very natural idea is to go in the direction in which $F$ decreases the fastest, which is the negative gradient:

$$u_i = -\nabla F(x_i)$$

**Example:** With the same function

$$F(x) = F(x, y) = 10(x - y)^2 + (1 - x)^4$$

and initial guess $x_0 = (1.25, 1.25)$, we get

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$F(x_i)$</th>
<th>$u_{i+1} = -\nabla F(x_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.2500000, 1.2500000)</td>
<td>0.0039062500</td>
<td>(-0.0625000, 0.0000000)</td>
</tr>
<tr>
<td>(1.2469850, 1.2500000)</td>
<td>0.0038120960</td>
<td>(0.0000498, -0.0603151)</td>
</tr>
<tr>
<td>(1.2469850, 1.2469900)</td>
<td>0.0037212990</td>
<td>(-0.0602052, -0.0000620)</td>
</tr>
<tr>
<td>(1.2440800, 1.2469870)</td>
<td>0.0036336850</td>
<td>(0.0000185, -0.0581813)</td>
</tr>
<tr>
<td>(1.2440790, 1.2440810)</td>
<td>0.0035491140</td>
<td>(-0.0581253, -0.0000381)</td>
</tr>
<tr>
<td>(1.2412620, 1.2440790)</td>
<td>0.0034674370</td>
<td>(0.0001655, -0.0563383)</td>
</tr>
</tbody>
</table>

and so on, for hundreds of iterations. (It will eventually converge).

On the other hand, if we use the initial guess $x_0 = (1.25, 1.5)$, we get

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$F(x_i)$</th>
<th>$u_{i+1} = -\nabla F(x_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.2500000, 1.5000000)</td>
<td>0.6289063000</td>
<td>(4.9375000, -5.0000000)</td>
</tr>
<tr>
<td>(1.3716800, 1.3767800)</td>
<td>0.0193443900</td>
<td>(-0.1033715, -0.1020122)</td>
</tr>
<tr>
<td>(1.0836200, 1.0925090)</td>
<td>0.0008389865</td>
<td>(0.1754356, -0.1777744)</td>
</tr>
<tr>
<td>(1.0880010, 1.0880700)</td>
<td>0.0000600194</td>
<td>(-0.0013527, -0.0013733)</td>
</tr>
<tr>
<td>(1.0375130, 1.0368130)</td>
<td>0.0000068819</td>
<td>(-0.0142135, 0.0140023)</td>
</tr>
<tr>
<td>(1.0371580, 1.0371630)</td>
<td>0.000019065</td>
<td>(-0.0001051, -0.0001001)</td>
</tr>
<tr>
<td>(1.0337760, 1.0339400)</td>
<td>0.000015701</td>
<td>(0.0031241, -0.0032783)</td>
</tr>
<tr>
<td>(1.0338540, 1.0338580)</td>
<td>0.000013136</td>
<td>(-0.0000789, -0.0000763)</td>
</tr>
</tbody>
</table>

and we get much closer to the true solution before things bog down. □

What is happening here is the following: At every step, the new search direction is perpendicular to the previous one. (Remember, the old direction is tangent to the level surface, the new direction is perpendicular to the level surface).
If the level surfaces are almost spherical near the minimum, steepest descent works quite well. If the level curves are elliptical, the method typically does a lot of zig-zagging. The more elongated the level curves are, the worse it gets.

Imagining you are standing in a long and narrow canyon, partly up the side slope. (This is Iowa, but you have probably seen canyons on TV). You want to reach the bottom of the valley, so you should be following the long direction of the valley. Instead, if you choose the method of steepest descent, you have to go sideways first, down to the valley floor, since the slope is steeper in that direction. After that, you have to switch back and forth at 90° angles, so you slowly zigzag down the valley.

If you are lucky, you happen to be on the valley floor at the start, so your initial direction is almost correct. This is what happened with the second set of initial data in the example.

This particular example is constructed to have level curves which are approximately ellipses with the longer diameter ten times as long as the shorter diameter. In general, the method of steepest descent will do very poorly here.

For a general problem, you have no clue what the level surfaces look like. Especially in higher dimensions, it is more likely than not that steepest descent will not work very well. Appropriate scaling may improve things.

**9.4.4. Conjugate Gradient.** Let us review the general approach: We choose an initial \( x_0 \), then find \( x_1 \) by minimizing in the \( u_0 \)-direction, then \( x_2 \) by minimizing in the \( u_1 \)-direction, and so on.

\( x_1 \) is a local minimum in the \( u_0 \)-direction, \( x_2 \) is a local minimum in the \( u_1 \)-direction, by construction. However, we cannot assume that \( x_2 \) is still a minimum in the \( u_0 \) direction. The question arises whether there is a way to choose \( u_1 \) so that \( x_2 \) is still a minimum in the \( u_1 \) direction.

In general, the answer is no. However, in the case when \( F \) is a quadratic function, we can do this. For general \( F \) then we replace \( F \) by its second order Taylor approximation, and achieve this goal approximately. Thus,

\[
F(x) \approx F(x_1) + DF(x_1)(x - x_1) + \frac{1}{2}(x - x_1)^T \left[ D^2F(x_1) \right] (x - x_1) \tag{9.1}
\]

We know

\[
\begin{align*}
    u_0 \cdot \nabla F(x_1) &= 0, \\
    u_1 \cdot \nabla F(x_2) &= 0.
\end{align*}
\]

This just expresses the fact that \( x_1 \) is a local minimum in the \( u_0 \)-direction and \( x_2 \) is a local minimum in the \( u_1 \)-direction. We want \( x_2 \) to be also a local minimum in the \( u_0 \)-direction, so we demand

\[
\begin{align*}
    u_0 \cdot \nabla F(x_2) &= 0. \tag{9.2}
\end{align*}
\]

Putting equations (9.4.4) together, we get

\[
\begin{align*}
    u_0 \cdot [\nabla F(x_2) - \nabla F(x_1)] &= 0. \tag{9.3}
\end{align*}
\]

By differentiating (9.1), we get

\[
\nabla F(x) \approx \nabla F(x_1) + \left[ D^2F(x_1) \right] (x - x_1), \tag{9.4}
\]

so

\[
\nabla F(x_2) - \nabla F(x_1) \approx \left[ D^2F(x_1) \right] (x_2 - x_1) = \lambda_2 \left[ D^2F(x_1) \right] u_1.
\]

(Remember, \( x_2 = x_1 + \lambda u_1 \) for some \( \lambda \)). Substitute that into (9.2) and cancel the \( \lambda_2 \) to get

\[
\begin{align*}
    u_0^T \left[ D^2F(x_1) \right] u_1 &= 0.
\end{align*}
\]

If \( A \) is any symmetric, positive definite matrix (whatever that means), we say that \( u \) and \( v \) are conjugate vectors (with respect to \( A \)) if

\[
\begin{align*}
    u^T A v &= 0.
\end{align*}
\]

Being conjugate is something like being perpendicular. In fact, if \( A \) is the identity matrix, conjugate vectors are just perpendicular vectors.

We just found that \( u_1 \) has the desired property (of making \( x_2 \) a local minimum in the \( u_0 \)-direction) if \( u_0 \) and \( u_1 \) are conjugate with respect to the second derivative matrix of \( F \) (which happens to be symmetric and positive definite).

Likewise, \( x_3 \) will be a minimum in the \( u_0, u_1 \) and \( u_2 \)-directions if \( u_2 \) is conjugate to both \( u_0 \) and \( u_1 \).
Our new algorithm is now this:

- $x_0$ is an initial guess.
- $u_0$ is the first search direction. Usually we use $u_0 = -\nabla F(x_0)$.
- $x_1$ is the local minimum in direction $u_0$.
- $u_1$ is any direction conjugate to $u_0$ with respect to $D^2F(x_1)$ (the second derivative matrix of $F$ at $x_1$).
- $x_2$ is the local minimum in direction $u_1$.
- $u_2$ is any direction conjugate to $u_0$ and $u_1$ with respect to $D^2F$.

and so on.

Think of the analogy with orthogonality again.

You start with some $u_0$. There are many possible directions orthogonal to $u_0$, so $u_1$ is not uniquely determined. At the next step, you need to find a direction orthogonal to both $u_0$ and $u_1$, so the choices are more limited, but there is still no unique answer. Eventually, at the $n$th step, there is a unique answer. After that, you have to throw out some of the older vectors, otherwise there is no solution at all.

In practice, it has proved useful to throw away all the $u_i$ periodically and start over again.

If $F$ was a true quadratic function (that is, if (9.1) was exact), then $x_2$ would be a minimum in both the $u_0$ and $u_1$ directions, $x_3$ would be a minimum in directions $u_0$, $u_1$, and $u_2$, and so on. $x_n$ would be the exact answer, where $n$ is the dimension of the space. In practice, $x_2$ is only approximately a minimum in the $u_0$ direction, and $x_n$ is only an approximate minimum, so we have to keep on going.

A geometric interpretation of the conjugate gradient method is that we are trying to find the main axes of the ellipsoid level surfaces. Once we know those, we can zero in on the minimum much faster. Think of the analogy with the long and steep canyon again. After getting oriented, we want to figure out what the main direction of the valley is.

The algorithm, as described, has a serious flaw: we need to calculate the matrix of second derivatives, and then solve a matrix equation at every step to find the new direction.

There is an ingenious algorithm due to Polak and Ribi`ere which avoids that and still produces conjugate directions. We cannot go over the derivation, but the search directions are produced as follows:

$$u_0 = g_0 = -\nabla F(x_0)$$
$$g_1 = -\nabla F(x_1)$$
$$\gamma_1 = \frac{(g_1 - g_0) \cdot g_1}{g_0 \cdot g_0}$$
$$u_1 = g_1 + \gamma_1 \cdot u_0$$
$$g_2 = -\nabla F(x_2)$$
$$\gamma_2 = \frac{(g_2 - g_1) \cdot g_2}{g_1 \cdot g_1}$$
$$u_2 = g_2 + \gamma_2 \cdot u_1$$

and so on. This will produce conjugate directions. Each new search direction is the negative gradient plus a correction term.

**Example:** With the same function

$$F(x) = F(x, y) = 10(x - y)^2 + (1 - x)^4$$

and initial guess $x_0 = (1.25, 1.25)$, we get

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$F(x_i)$</th>
<th>$u_{i+1} = -\nabla F(x_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.25000000, 1.25000000)</td>
<td>0.0039062500</td>
<td>(-0.00250000, 0.00000000)</td>
</tr>
<tr>
<td>(1.2469400, 1.25000000)</td>
<td>0.0038120960</td>
<td>(-0.0582067, -0.0603151)</td>
</tr>
<tr>
<td>(1.0814370, 1.0784570)</td>
<td>0.0001328376</td>
<td>(-0.5609487, -0.4576368)</td>
</tr>
<tr>
<td>(1.0637270, 1.0640080)</td>
<td>0.0000172822</td>
<td></td>
</tr>
</tbody>
</table>
With initial guess $x_0 = (1.25, 1.5)$, we get

\[
\begin{array}{ccc}
  x_i & F(x_i) & u_{i+1} = -\nabla F(x_i) \\
  (1.2500000, 1.5000000) & 0.6289063000 & (4.9375000, -5.0000000) \\
  (1.3716800, 1.3767800) & 0.0193443900 & (-0.1012289, -0.1041820) \\
  (1.0793160, 1.0758880) & 0.0001571043 & (-0.2005831, -0.0652514) \\
  (1.0740560, 1.0741770) & 0.0000302231 & \\
\end{array}
\]

\[\square\]

9.4.5. Newton and Quasi-Newton Methods. The Newton method for optimization, which we discussed in the one-dimensional case earlier, generalizes to higher dimensions. We can derive the formula as follows.

Look at formula (9.4.4) again, this time around the minimum point $\bar{x}$

\[
\nabla F(x) \approx \nabla F(\bar{x}) + [D^2F(\bar{x})] (x - \bar{x}) = [D^2F(\bar{x})] (x - \bar{x}),
\]

since the gradient vanishes at the minimum. If we are at the point $x$, we need to go in direction $(\bar{x} - x)$ to get to $\bar{x}$. From the last equation we find

\[
\bar{x} - x \approx [D^2F(\bar{x})]^{-1} \nabla F(x) \approx [D^2F(x)]^{-1} \nabla F(x),
\]

if we assume that $D^2F$ does not change very much. The resulting method is an exact analogue of the one-dimensional method

\[
x_{n+1} = x_n - [D^2F(x_n)]^{-1} \nabla F(x_n)
\]

There are actually two different methods here: Either, we can use the formula the way we just wrote it. In this case, we need not do any one-dimensional minimizations. Or, we could use $[D^2F(x_n)]^{-1} \nabla F(x_n)$ as our new search direction $u_{n+1}$ and do a one-dimensional search.

I think that one method is known as Newton-Raphson, the other one is just Newton’s method. Unfortunately, I can never remember which is which, and the books I have handy don’t say. Most people use “Newton” and “Newton-Raphson” interchangeably, anyway.

**Example:** With the same function

\[
F(x) = F(x, y) = 10(x - y)^2 + (1 - x)^4
\]

and initial guess $x_0 = (1.25, 1.25)$, we get

\[
\begin{array}{ccc}
  x_i & F(x_i) & u_{i+1} = -\nabla F(x_i) \\
  (1.2500000, 1.2500000) & 0.0039062500 & (-0.0033333, -0.0033333) \\
  (1.0000190, 1.0000190) & 0.0000000000 & \\
\end{array}
\]

With initial guess $x_0 = (1.25, 1.5)$ we get

\[
\begin{array}{ccc}
  x_i & F(x_i) & u_{i+1} = -\nabla F(x_i) \\
  (1.2500000, 1.5000000) & 0.6289063000 & (-0.0033333, -0.2533333) \\
  (1.2466660, 1.2466660) & 0.0037020270 & (-0.0032189, -0.0031790) \\
  (1.0524790, 1.0548480) & 0.0000637142 & \\
\end{array}
\]

In both cases, I have done line minimizations. $\square$

Instead of trying to find out the main axes of the ellipsoid level surfaces, this method tries to go to the center in a single step. In this example, it pretty much succeeded.

The main problem (as usual) with this method is the need for a full matrix of second derivatives, plus the need to invert the matrix at every step.

A whole class of methods, known as quasi-Newton methods, deals with this problem. The idea is to replace the exact inverse second derivative matrix by some approximation to it which can be found with relatively little work. Usually, we assume that the matrix changes very little from step to step, so we just update the last matrix based on information we gathered at the last step. The details go far beyond the scope of this course.
9.4.6. **Summary of Higher Dimensional Optimization.** You have probably gathered by now that optimization in higher dimensions is fairly complicated and (computer) time consuming. The higher the dimension is, the more work you need to do per step, and simultaneously you need more steps.

We have considered a few direction set methods, namely

- Steepest Descent, where we search in the direction of the negative gradient;
- Conjugate Gradient, where we search in a direction conjugate to the earlier directions;
- Newton’s method;
- Quasi-Newton methods.

Steepest Descent and Conjugate Gradients may or may not work well. Newton’s method, as usual, is the best, except that in higher dimensions the amount of extra work becomes enormous.

A look at some subroutine libraries reveals that quasi-Newton methods are the methods of choice. Every single one I looked at had some form of quasi-Newton method in it. This includes the subroutine UNCMIN provided with NMS.

9.5. **Subroutine UNCMIN**

Read this section on your own.