Lecture 18

Optimization in n dimensions

1 Introduction

We now consider the problem of minimizing a single scalar function of n variables, $f(\mathbf{x})$, where $\mathbf{x}=[x_1, x_2, \ldots, x_n]^T$. The 2D case can be visualized as "finding the lowest point" of a surface $z=f(x, y)$ (Fig. 1).

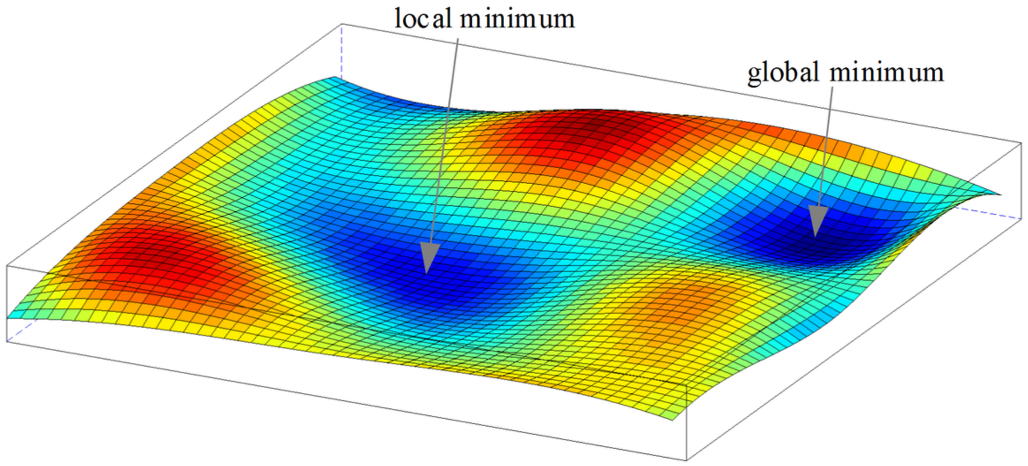

Fig. 1 The 2D minimization problem is equivalent to finding the "lowest point" on a surface.

A necessary condition for a minimum is that $\partial f / \partial x_i = 0$ for all $1 \le i \le n$. The partial derivative $\partial f / \partial x_i$ is the ith component of the gradient of *f*, denoted ∇f, so at a minimum we must have

$$\nabla f = 0 \tag{1}$$

In the 2D case this implies we've "bottomed out" at the lowest point of a "valley." The gradient also vanishes at a maximum, so this is a necessary but not sufficient condition for a minimum.

2 Quadratic functions

Quadratic functions of several variables come up in many applications. A quadratic function of n variables x_i has the form

$$f(x_1, x_2, \ldots, x_n) = c + \sum_{i=1}^{n} b_i x_i + \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} x_i x_j \tag{2}$$

Since

$$\frac{1}{2} a_{ij} x_i x_j + \frac{1}{2} a_{ji} x_j x_i = \frac{1}{2}(a_{ij} + a_{ji}) x_i x_j \tag{3}$$

the coefficients a_{ij}, a_{ji} only appear as the sum $a_{ij} + a_{ji}$. Without loss of generality, therefore, we can take $a_{ij} = a_{ji}$.

By differentiation we have

$$f(0)=c \ , \ \frac{\partial f}{\partial x_i}=b_i \ , \ \frac{\partial f^2}{\partial x_i \partial x_j}=a_{ij} \tag{4}$$

which allows us to interpret the form (2) as a multivariable Taylor series of an arbitrary function. The conditions for a minimum (or maximum) are (for $k=1,2,\ldots,n$)

$$\frac{\partial f}{\partial x_k}=b_k+\frac{1}{2}\sum_{i=1}^{n} a_{ik}x_i+\frac{1}{2}\sum_{j=1}^{n} a_{kj}x_j=b_k+\sum_{j=1}^{n} a_{kj}x_j=0 \tag{5}$$

In matrix notation this reads

$$\nabla f=\mathbf{b}+\mathbf{A}\,\mathbf{x}=0 \tag{6}$$

where b_i are the components of **b**, and a_{ij} are the components of the symmetric matrix **A**. The solution is

$$\mathbf{x}=-\mathbf{A}^{-1}\mathbf{b} \tag{7}$$

The minimization of a quadratic function is equivalent to the solution of a linear system. However, for an arbitrary function $f(\mathbf{x})$ we can't make any general statements about a minimum. This motivates us to seek a method to systematically search for the minimum of a function of *n* variables.

3 Line minimization

We know how to go about minimizing a function of one variable. If we start at a point $\mathbf{x}_0$ and move only in the direction of a vector **u** (Fig. 2) then the $f(\mathbf{x})$ values we can sample form a function of a single variable

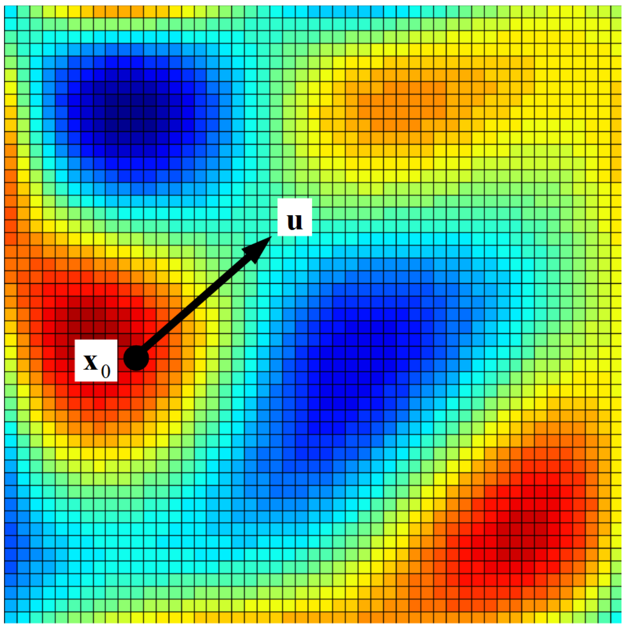

Fig. 2 $\mathbf{x}_0$ *is the starting point and* **u** *the direction in which we search for a minimum.*

$$g(t) = f(\mathbf{x}_0 + t\,\mathbf{u}) \tag{8}$$

Here the variable t is the distance we move in the direction $\mathbf{u}$. We can use any of our 1D minimization methods on this function. Of course this is not likely to find the minimum of $f(\mathbf{x})$. However, suppose we start at $\mathbf{x}_0$ and move along the direction $\mathbf{u}_1$ to a minimum. Call this new point $\mathbf{x}_1 = \mathbf{x}_0 + t_1\mathbf{u}_1$. Then move along another direction $\mathbf{u}_2$ to find a minimum at $\mathbf{x}_2 = \mathbf{x}_1 + t_2\mathbf{u}_2$ and so on. This process should eventually find a local minimum (if one exists). The process of minimizing the function $f(\mathbf{x})$ along a single line defined by some vector $\mathbf{u}$ is called *line minimization*. The algorithm is quite simple

```
Successive line minimization algorithm
start with initial guess $\mathbf{x}_0$ and search directions $\mathbf{u}_i$
iterate until converged
  for $i = 1,2,\ldots,n$
    find $t$ that minimizes $g(t) = f(\mathbf{x}_0 + t\,\mathbf{u}_i)$
    set $\mathbf{x}_0 \leftarrow \mathbf{x}_0 + t\,\mathbf{u}_i$
```

An obvious set of directions is the coordinate axes

$$\mathbf{u}_1 = \begin{pmatrix}1\\0\\0\\\vdots\\0\end{pmatrix}, \mathbf{u}_2 = \begin{pmatrix}0\\1\\0\\\vdots\\0\end{pmatrix}, \cdots, \mathbf{u}_n = \begin{pmatrix}0\\0\\0\\\vdots\\1\end{pmatrix} \tag{9}$$

In this case the algorithm simply minimizes $f(\mathbf{x})$ with respect to x_1, then with respect to x_2 and so on. The algorithm will find a minimum (if one exists), but in many cases it can be very slow to do so. An example is shown in Fig. 3 where we minimize the quadratic function

$$f(x,y) = \left(\frac{x-y}{4}\right)^2 + (x+y-2)^2 \tag{10}$$

starting at $x=y=0$. The minimum is at $x=y=1$. From the contour plot we see that the "valley" of this function is narrow and oriented 45° to the coordinate axes. Since we are limited to moving in only the *x* or *y* direction at any one time, the algorithm ends up taking many, progressively smaller, zig-zag steps down the valley. The net movement is in a diagonal direction along the "valley floor." If that direction was one of our $\mathbf{u}_i$ directions then we might be able to take one big step directly to the minimum. This motivates the development of "direction set" methods which attempt to adapt the $\mathbf{u}_i$ directions to the geometry of the function being minimized.

Consider the quadratic function (2). This can be written as

$$f = c + \mathbf{x}^T\mathbf{b} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\,\mathbf{x} \tag{11}$$

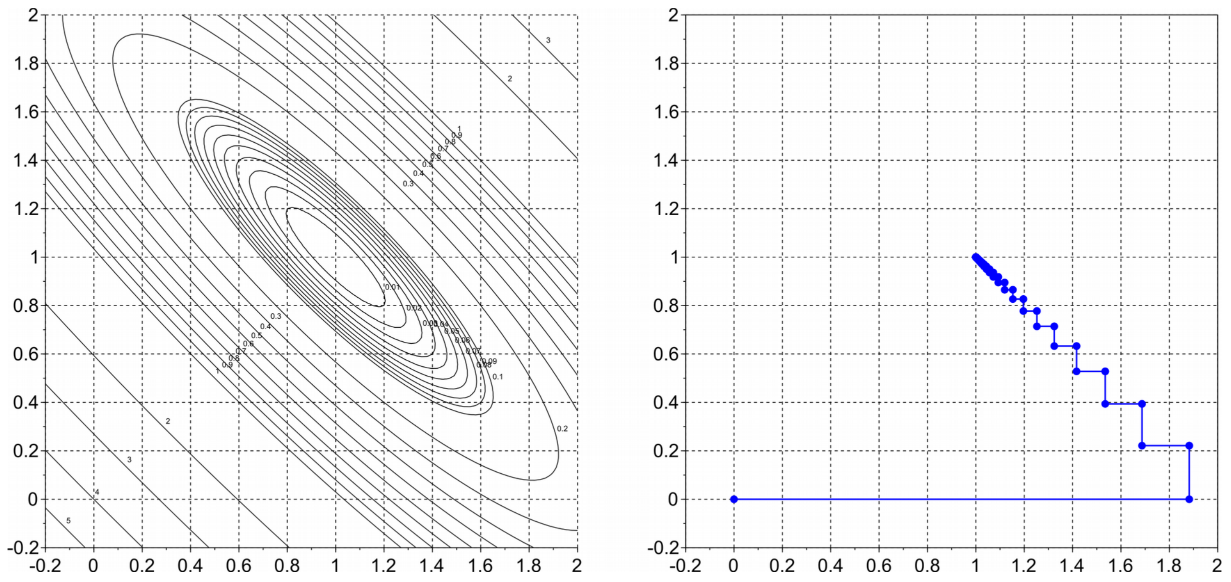

Fig. 3: Successive line minimization along the coordinate axes. Left: contours of quadratic function. Right: Progressive results of line minimization.

The gradient is

$$\nabla f = \mathbf{b} + \mathbf{A}\,\mathbf{x} \tag{12}$$

Suppose we have found the minimum of $g_1(t) = f(\mathbf{x}_0 + t\,\mathbf{u}_1)$ at $\mathbf{x}_1$. That this point the gradient of *f* must be orthogonal to $\mathbf{u}_1$, otherwise we could move along the $\mathbf{u}_1$ to lower values of *f*. Therefore

$$\mathbf{u}_1^T\,\mathbf{b} + \mathbf{u}_1^T\,\mathbf{A}\,\mathbf{x}_1 = 0 \tag{13}$$

Now we find the minimum of $g_2(t) = f(\mathbf{x}_1 + t\,\mathbf{u}_2)$ at $\mathbf{x}_2 = \mathbf{x}_1 + t_m\,\mathbf{u}_2$. At this new $\mathbf{x}$ value we want both $\mathbf{u}_1$ and $\mathbf{u}_2$ to be orthogonal to the gradient. This ensures that the new point remains a minimum along the $\mathbf{u}_1$ direction as well as the $\mathbf{u}_2$ direction. This requires

$$\mathbf{u}_1^T\,\mathbf{b} + \mathbf{u}_1^T\,\mathbf{A}(\mathbf{x}_1 + t_m\,\mathbf{u}_2) = 0 \tag{14}$$

Because of (13) this reduces to

$$\mathbf{u}_1^T\,\mathbf{A}\,\mathbf{u}_2 = 0 \tag{15}$$

Two vectors $\mathbf{u}_1, \mathbf{u}_2$ satisfying this condition are said to be *conjugate*. A set of *n* vectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n$ in which all pairs are conjugate is a *conjugate set*. One of the first methods presented (1964) to generate a set of conjugate directions was *Powell's method* to which we now turn.

4 Powell's method

Powell showed that a simple addition to the successive line minimization algorithm enables it to find conjugate directions and minimize an arbitrary quadratic function of *n* variables in *n* iterations. After completing the line minimizations of the for loop, we form a new direction $\mathbf{v}$ which is the net direction $\mathbf{x}_0$ moved due to the *n* line minimizations. We then perform a single line minimization along the direction $\mathbf{v}$. Finally, we discard the first search direction, $\mathbf{u}_1$, "left shift" the other $n-1$ directions ($\mathbf{u}_i \leftarrow \mathbf{u}_{i+1}$) and make $\mathbf{v}$ the new $\mathbf{u}_n$ direction. It turns out any quadratic function will be minimized by *n* iterations of this procedure. The algorithm is

```
Powell's method
start with initial guess $\mathbf{x}_0$ and search directions $\mathbf{u}_i$
iterate until converged
    save current estimate $\mathbf{x}_0^{\text{old}} \leftarrow \mathbf{x}_0$
        for $i=1,2,\ldots,n$
            find $t$ that minimizes $f(\mathbf{x}_0+t\,\mathbf{u}_i)$
            set $\mathbf{x}_0 \leftarrow \mathbf{x}_0+t\,\mathbf{u}_i$

    $\mathbf{v} \leftarrow [\mathbf{x}_0-\mathbf{x}_0^{\text{old}}]/\|\mathbf{x}_0-\mathbf{x}_0^{\text{old}}\|$
    find $t$ to minimize $f(\mathbf{x}_0+t\,\mathbf{v})$
    set $\mathbf{x}_0 \leftarrow \mathbf{x}_0+t\,\mathbf{v}$
    for $i=1,2,\ldots,n-1$
        $\mathbf{u}_i \leftarrow \mathbf{u}_{i+1}$
    $\mathbf{u}_n \leftarrow \mathbf{v}$
```

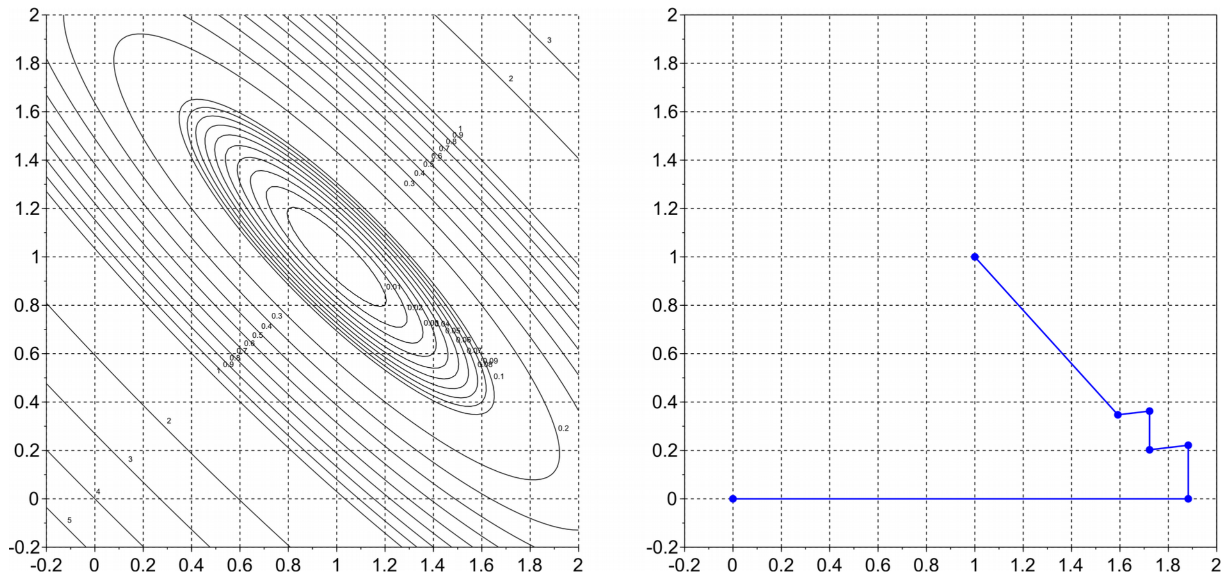

Fig. 4: Powell's method. Left: contours of quadratic function; Right: progressive results of each line minimization. The minimum is found in $n=2$ iterations (6 line minimizations total).

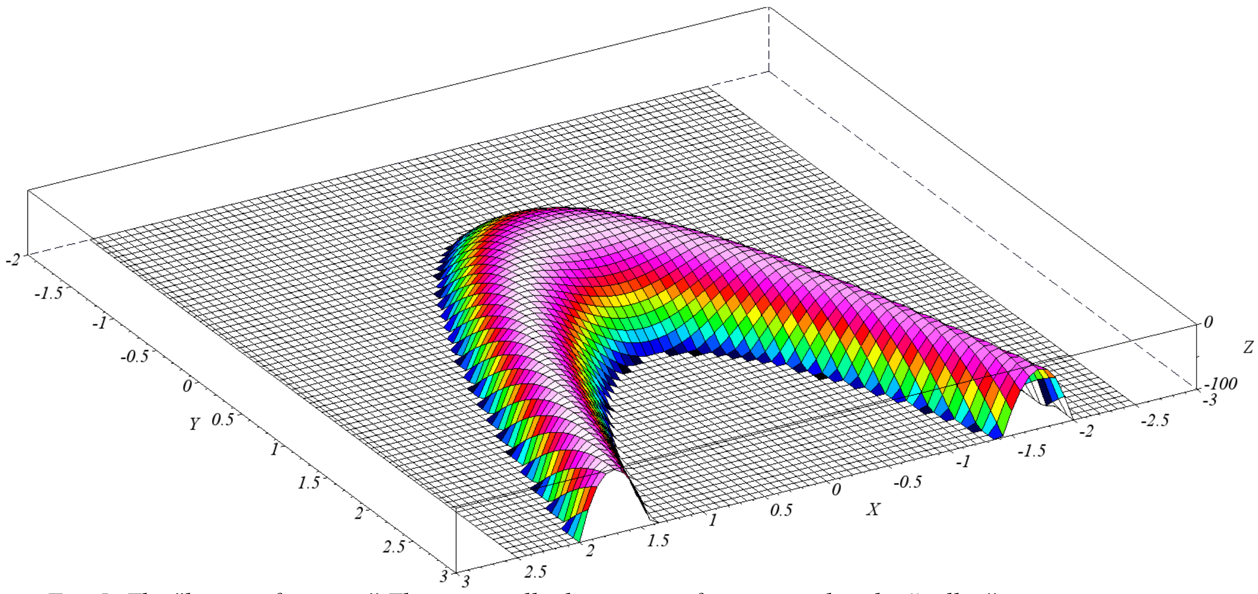

Fig. 5: The "banana function." This is actually the negative function so that the "valley" appears as a "hill." Values less than -100 have been "chopped off" to show greater detail.

A Scilab version of Powell's method is given in the Appendix. Applying this to function (10), starting at $x=y=0$, we obtain the results shown in Fig. 4. In two iterations of three line minimizations each (Powell's method adds one line minimization after the for loop) we arrive at the minimum. Powell's method has "figured out" the necessary diagonal direction.

A more challenging test is given by the *Rosenbrock function*

$$f(x,y)=(1-x)^2+100(y-x^2)^2 \tag{16}$$

shown in Fig. 5. Because of its shape it is sometimes called the "banana function." The minimum

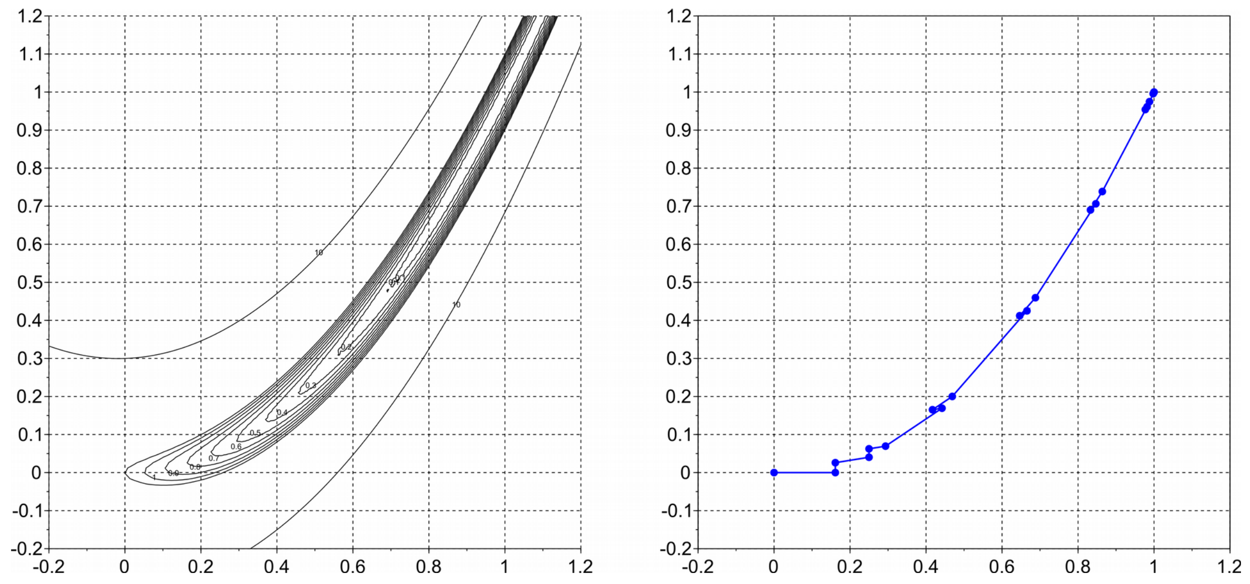

Fig. 6 Powell's method following the "valley" of the "banana" function.

is $f(1,1)=0$. Unlike the function of Fig. 4, the “valley” of this function twists and the algorithm must following this changing direction. Starting at $x=y=0$ Powell's method gives the results shown in Fig. 6. We can see how the algorithm tracks the twisting valley and arrives at the minimum after only a few iterations.

5 Newton's method

Earlier we saw that the gradient of the quadratic function

$$f(x_1,x_2,\ldots,x_n)=c+\sum_{i=1}^{n} b_i x_i+\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_i x_j \tag{17}$$

vanishes at

$$\mathbf{x}=-\mathbf{A}^{-1}\mathbf{b} \tag{18}$$

Newton's method approximates an arbitrary function by a quadratic Taylor series with

$$b_i=\frac{\partial f}{\partial x_i} \ , \ a_{ij}=a_{ji}=\frac{\partial f^2}{\partial x_i \partial x_j} \tag{19}$$

The vector **b** is the gradient of *f*. The matrix of second derivatives **A** is called the *Hessian* of *f*. We solve for the minimum of this quadratic Taylor series and take that to be our new **x**. We continue until the method has converged.

```
Newton's method
iterate until converged
  at $\mathbf{x}$ evaluate the gradient $\mathbf{b}$ and the Hessian $\mathbf{A}$
  set $\mathbf{x}\leftarrow\mathbf{x}-\mathbf{A}^{-1}\mathbf{b}$
```

Let's apply Newton's method to the banana function

$$f(x,y)=(1-x)^2+100(y-x^2)^2 \tag{20}$$

The gradient is

$$\mathbf{b}=\begin{pmatrix}\dfrac{\partial f}{\partial x}\\ \dfrac{\partial f}{\partial y}\end{pmatrix}=\begin{pmatrix}-2(1-x)-400x(y-x^2)\\ 200(y-x^2)\end{pmatrix} \tag{21}$$

The Hessian is

$$\mathbf{A}=\begin{pmatrix}\dfrac{\partial^2 f}{\partial x^2} & \dfrac{\partial^2 f}{\partial x\partial y}\\ \dfrac{\partial^2 f}{\partial y\partial x} & \dfrac{\partial^2 f}{\partial y^2}\end{pmatrix}=\begin{pmatrix}2+1200x^2-400y & -400x\\ -400x & 200\end{pmatrix} \tag{22}$$

Starting at $x=y=0$, we have

$$\mathbf{b}=\begin{pmatrix}-2\\0\end{pmatrix}\ ,\ \mathbf{A}=\begin{pmatrix}2&0\\0&200\end{pmatrix}\ ,\ \mathbf{A}^{-1}=\frac{1}{400}\begin{pmatrix}200&0\\0&2\end{pmatrix}\ ,\ \mathbf{A}^{-1}\mathbf{b}=\begin{pmatrix}-1\\0\end{pmatrix} \tag{23}$$

and the new estimate is

$$\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}0\\0\end{pmatrix}-\begin{pmatrix}-1\\0\end{pmatrix}=\begin{pmatrix}1\\0\end{pmatrix} \tag{24}$$

At $x=1, y=0$ we have

$$\mathbf{b}=\begin{pmatrix}400\\-200\end{pmatrix}\ ,\ \mathbf{A}=\begin{pmatrix}1202&-400\\-400&200\end{pmatrix}\ ,\ \mathbf{A}^{-1}=\frac{1}{80400}\begin{pmatrix}200&400\\400&1202\end{pmatrix}\ ,\ \mathbf{A}^{-1}\mathbf{b}=\begin{pmatrix}0\\-1\end{pmatrix} \tag{25}$$

and the new estimate is

$$\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}1\\0\end{pmatrix}-\begin{pmatrix}0\\-1\end{pmatrix}=\begin{pmatrix}1\\1\end{pmatrix} \tag{26}$$

which is the solution.

Newton's method is conceptually simple and quite powerful. However, it requires us to compute the gradient and the Hessian of the function. This may be difficult or impossible in many cases. To overcome this challenge, *quasi-Newton* methods have been developed which attempt to form an approximation of the Hessian matrix, and possibly the gradient also, as the algorithm progresses. One such method is the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm, presented in 1970.

Another challenge is that for a complicated function, Newton's method needs to be started sufficiently close to a minimum to be stable. As in the 1D case this motivates the development of hybrid algorithms that attempt to use a quasi-Newton method when it works, but revert to a slow-but-sure backup when it does not. The Scilab `optim` function is of this type. The calling syntax is the same as for the 1D case

```
[fopt,xopt] = optim(list(NDcost,f),x0);
```

Here `f(x)` is the function to be minimized, `x0` is an initial guess at the minimum, `xopt` is the computed minimum and `fopt` is the minimum function value. By default optim assumes the function *f* provides both function and gradient values. If *f* returns only a function value, the `list(NDcost,f)` statement takes care of providing numerical derivatives. Here's this function applied to the banana function.

```
x0 = [0;0];
function z = f(x)
  z = (1-x(1))^2+100*(x(2)-x(1)^2)^2;
endfunction

[fopt,xopt] = optim(list(NDcost,f),x0);
disp(xopt);
```

This produces the output

```
    1.0000000
    1.0000000
```

which is the minimum.

6 Appendix – Scilab code

6.1 Powell's method

```
//////////////////////////////////////////////////////////////////////
// optimPowell.sci
// 2014-10-31, Scott Hudson, for pedagogic purposes.
// Implements Powell's method for minimizing a function of
// n variables.
//////////////////////////////////////////////////////////////////////
function [xmin, fmin]=optimPowell(fun, x0, h, tol)
  n = length(x0); //# of variables
  searchDir = eye(n,1); //direction for current search
  searchDirs = eye(n,n); //set of n search directions

  function s=gfun(t)         //local scalar function to pass to 1D
    s = fun(x0+t*searchDir) //optimization routines
  endfunction

  done = 0;
  while(~done)
    x0old = x0; //best solution so far
    for i=1:n //minimize along each of n directions
      searchDir = searchDirs(:,i);
      [a,b,c,fa,fb,fc] = optimBracket(-h,h,gfun);
      [tmin,gmin] = optimGolden(a,b,c,fa,fb,fc,gfun,tol/10);
      x0 = x0+tmin*searchDir; //minimum along this direction
    end
    for i=1:n-1 //update search directions
      searchDirs(:,i) = searchDirs(:,i+1);
    end
    v = x0-x0old; //new search direction
    searchDirs(:,n) = v/sqrt(v'*v); //add new search dir unit vector
    searchDir = searchDirs(:,n); //minimize along new direction
    [a,b,c,fa,fb,fc] = optimBracket(-h,h,gfun);
    [tmin,gmin] = optimGolden(a,b,c,fa,fb,fc,gfun,tol/10);
    x0 = x0+tmin*searchDir;
    xChange = sqrt(sum((x0-x0old).^2));
    if (xChange<tol)
      done = 1;
    end
  end //while
  xmin = x0;
  fmin = fun(xmin);
endfunction
```