STAT 513/413: Lecture 20
Walking hills and ravines

(A refresment of calculus)

Hills and ravines

One-dimensional case too trivial to bother with

Multi-dimensional cases too difficult to imagine/picture

The only one that somewhat can be: the two-dimensional case

hills = maximization   ravines = minimization

Some map reading experience may be handy – in particular, when I cannot wave my hands in class to enact 3D

Multidimensional calculus

That is, two-dimensional here: $f(x_1, x_2) = \frac{1}{64}x_1^4 + \frac{1}{4}x_2^2$, say

Well, this is nice, as it is readily generalized into p dimensions...

...but will be notational pain when doing iterative methods

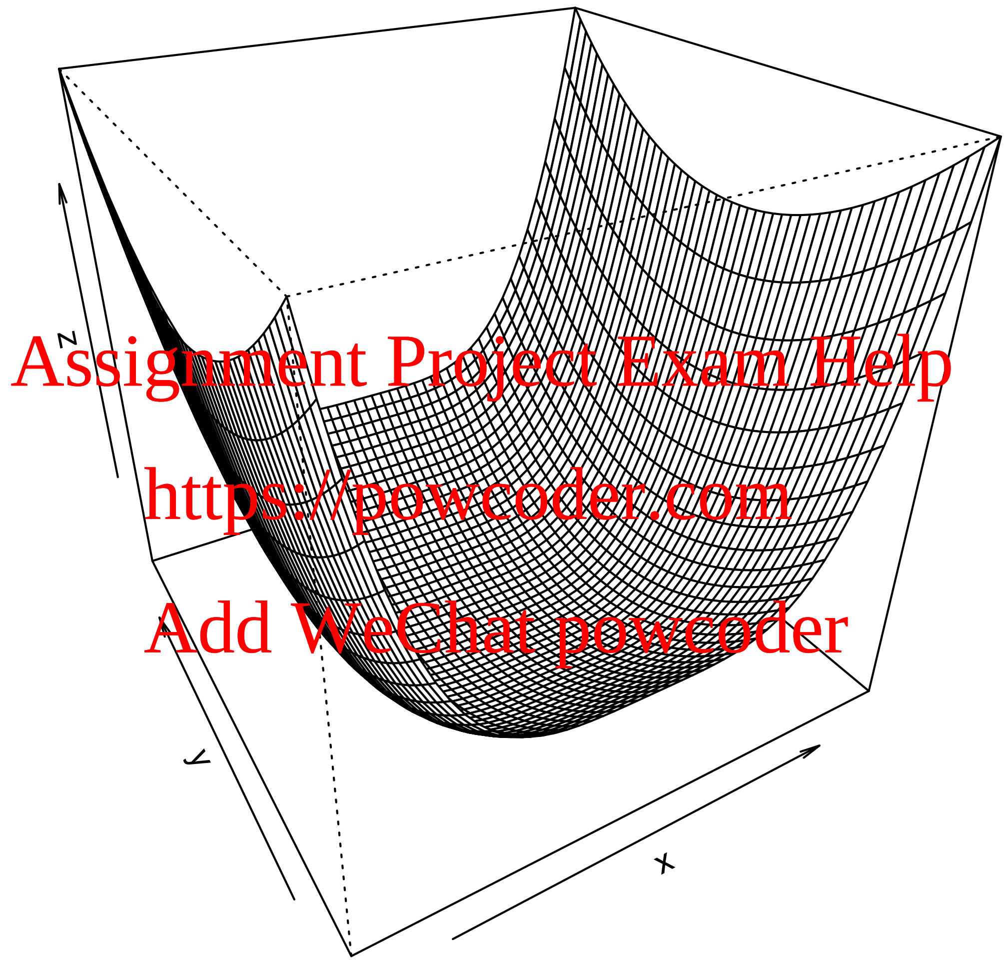
(although could be handled: (x_1^1, x_2^1), (x_1^2, x_2^2), (x_1^3, x_2^3), ...)

So, let us make it rather $f(x, y) = \frac{1}{64}x^4 + \frac{1}{4}y^2$,

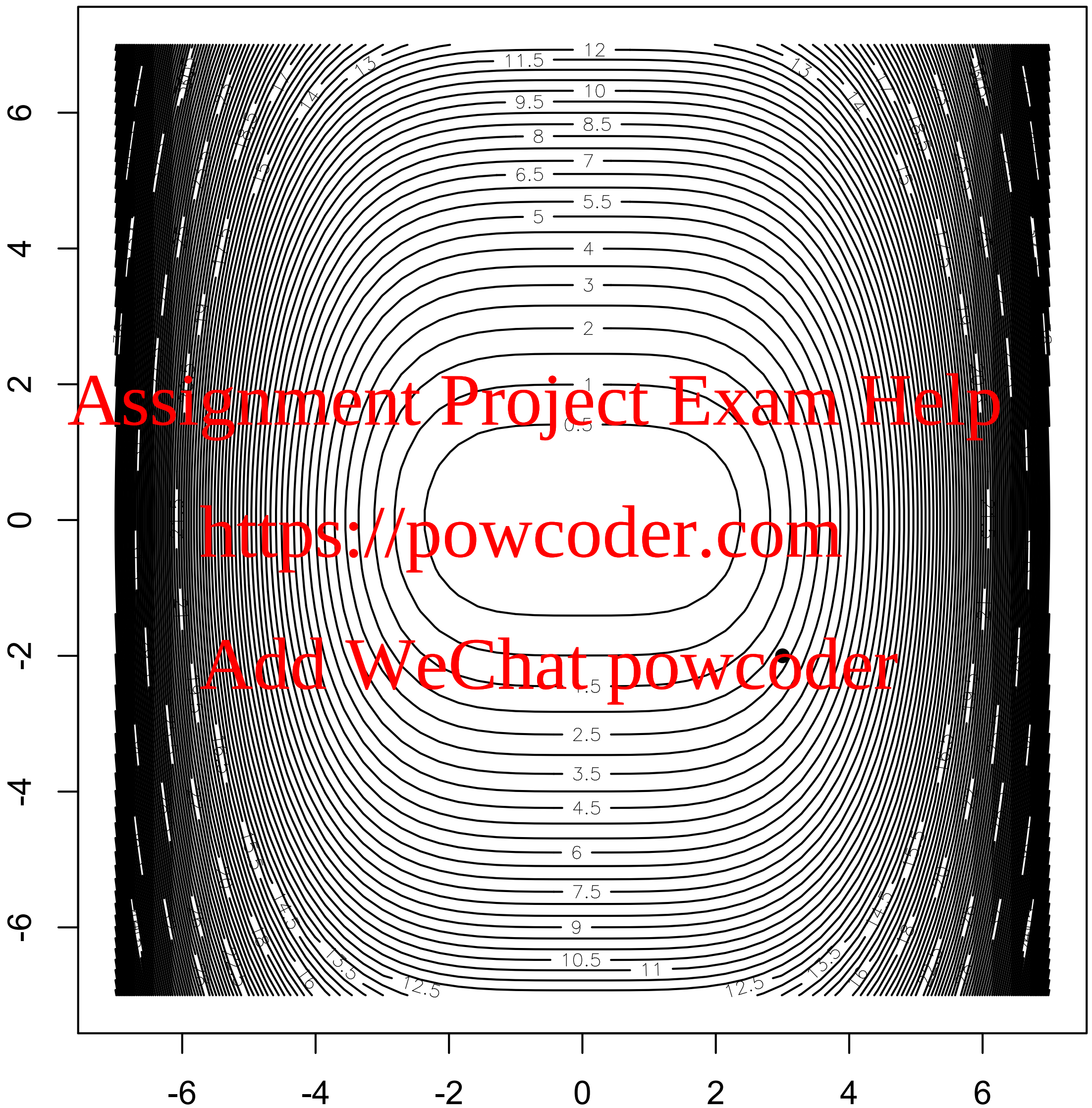
and have a look


```
> x=seq(-7,7,len=50)
> y=x
> xy=expand.grid(x,y)
> z=matrix((1/64)*xy[,1]^4 + (1/4)*xy[,2]^2,
+ nrow=length(x),ncol=length(y))
> persp(x,y,z,theta=-30,phi=30)

> contour(x,y,z,nlevels=100)
> points(3,-2)

> arrows(3,-2,3+3^3/16,-2+(-2)/2)
```

Many looks possible, of course



So, now the map view



Question: where does the gradient point?

Partial derivatives first: good notation is here hard to come by...

$\dfrac{\partial f}{\partial x}$ ...good, but where is this taken?

$$\left.\frac{\partial f}{\partial x}\right|_{(x_1,y_1)} \quad \text{or} \quad \frac{\partial f(x,y)}{\partial x} \quad \text{or} \quad ???$$

I like more this: $D_x f(x,y)$, $D_y f(x,y)$

(In general, x_1, x_2 notation could be $D_1 f(x_1,x_2)$, $D_2 f(x_1,x_2)$, ...)

We can handle also higher-order partial derivatives this way

$$D_{xx} f(x,y) = D_x(D_x f(x,y)) \qquad D_{yy} f(x,y) = D_y(D_y f(x,y))$$

$$D_{yx} f(x,y) = D_y(D_x f(x,y))$$

which can be easily confused with $D_{xy} f(x,y) = D_x(D_y f(x,y))$

because in general it is not the same - but very often it is, and thus will be also in all what follows

$$D_{yx} f(x,y) = D_{xy} f(x,y)$$

So, where does the gradient point?

(was just kidding: no bonus, too late)

Gradient: partial derivatives put in a (column) vector

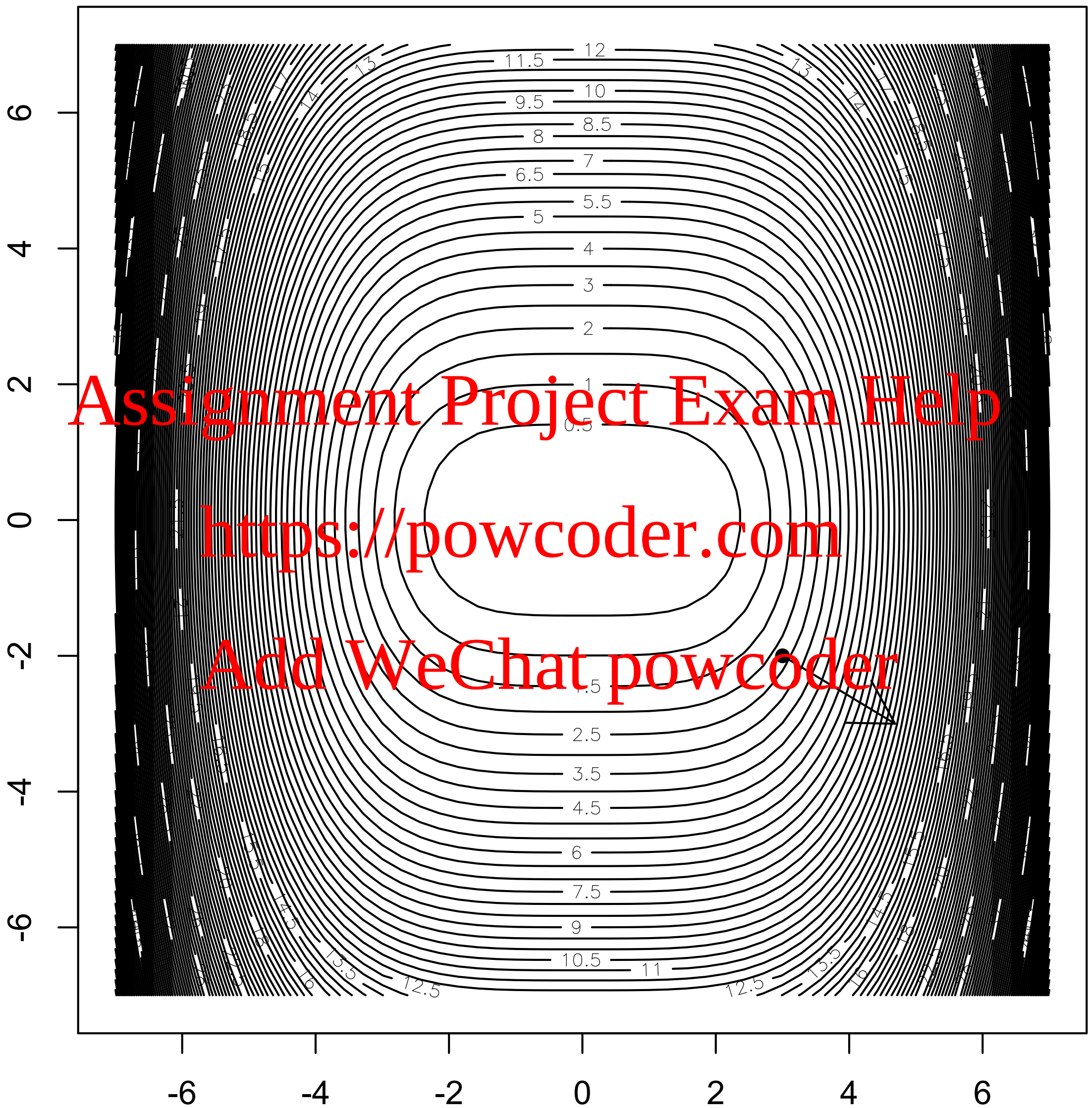
$$\nabla f(x, y) = Gf(x, y) = G(x, y) = \begin{pmatrix} D_x f(x, y) \\ D_y f(x, y) \end{pmatrix}$$

For our particular $f(x, y) = \frac{1}{64}x^4 + \frac{1}{4}y^2$,

$$G(x, y) = \begin{pmatrix} \frac{1}{16}x^3 \\ \frac{1}{2}y \end{pmatrix} \quad \text{which at } \begin{pmatrix} 1 \\ -1 \end{pmatrix} \text{ is } \begin{pmatrix} 5 \\ -3 \end{pmatrix}$$

So, where does the gradient point?

In the direction of maximal slope up!



Gradient descent algorithm

Also: method of steepest descent

Steepest ascent: in the direction of $G(x, y)$

Steepest descent: in the direction of $-G(x, y)$

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - G(x_k, y_k)$$

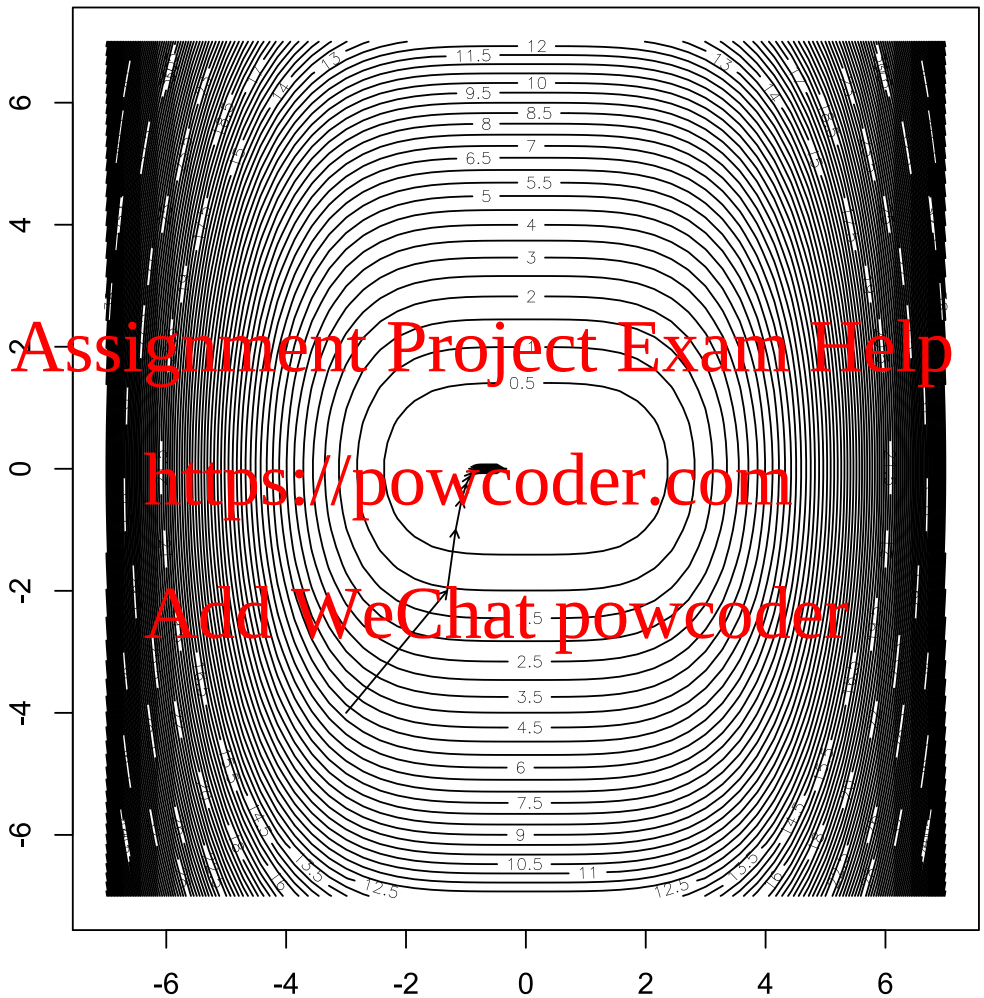
As this may bomb quite quickly, we may consider a modification

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - sG(x_k, y_k)$$

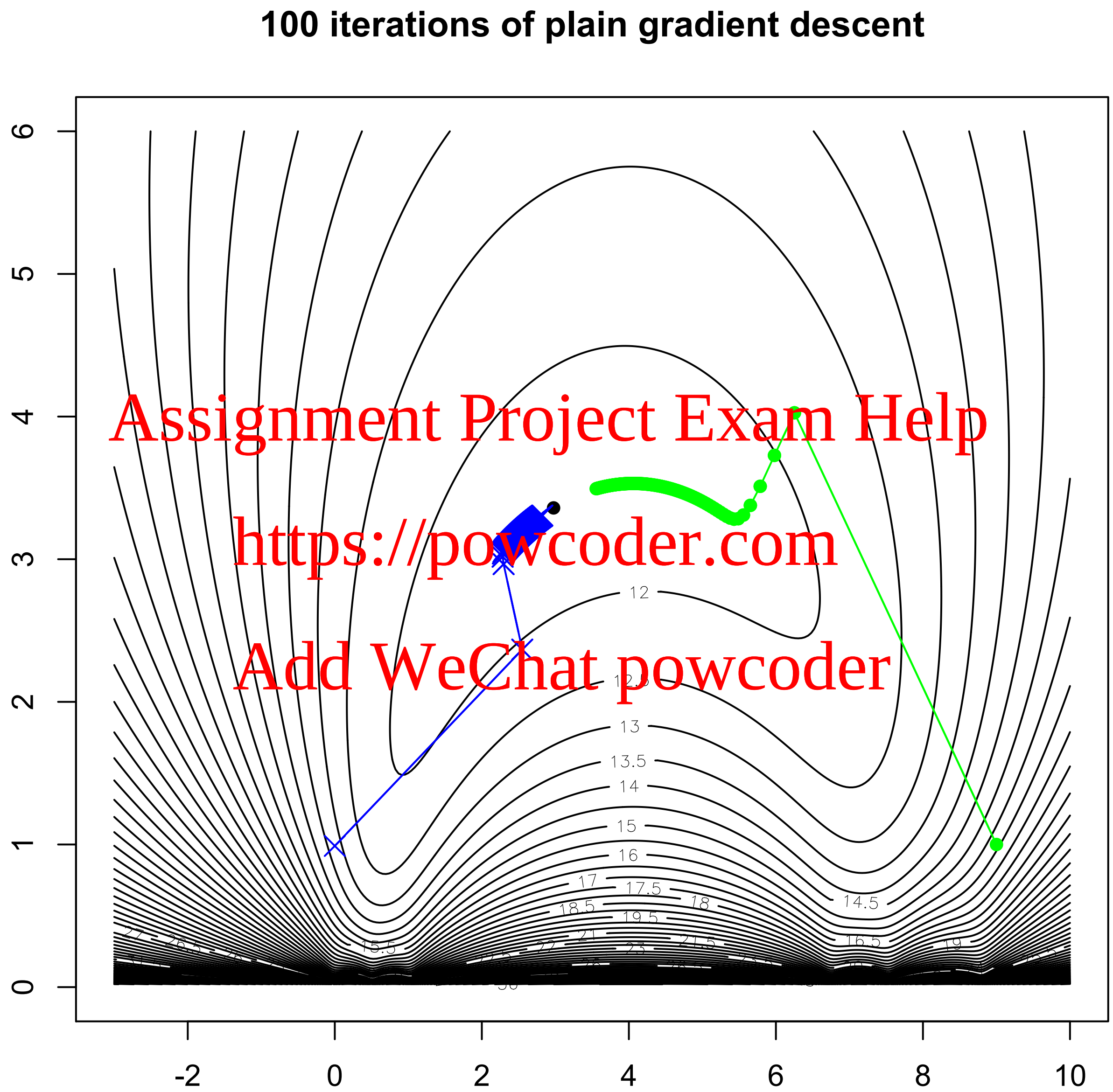
where $s > 0$ is some suitable number

As simple as that...

... but as lousy


12
11.5
10
9.5
8.5
8
7
6.5
5.5
5
4
3
2
0.5
2.5
3.5
4.5
6
7.5
9
10.5
11
12.5
13
13.5
-6
-4
-2
0
2
4
6
-6
-4
-2
0
2
4
6

Another demo



Gradient descent works - if the step is right

Slow, unreliable - but a workhorse in neural networks, for instance

And it works - if the step is set right at each iteration

Instead of simply doing

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - G(x_k, y_k)$$

we do

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - s_k G(x_k, y_k)$$

where s_k is found using the one-dimensional search along the halfline (so it is not constant, like γ, over all iterations)

$$\begin{pmatrix} x_k \\ y_k \end{pmatrix} - sG(x_k, y_k)$$

The determination of s_k is often done by backtracking: we select first $s = 1$, and then halve it, until we arrive to a suitable value

Another improvement of the gradient descent method: method of conjugate gradients (to be discussed later)

And now, the Newton method

First, we are in the optimization here now:

minimizing f means looking for $f'(x) = 0$

and the Newton method then does $\quad x_{k+1} = x_k - \dfrac{f'(x_k)}{f''(x_k)}$

Now, f' is likely the gradient. What is f''?

The Hessian matrix ("the derivative of gradient"): the element in the i-th row and j-th column is the (second) partial derivative, in i-th and j-th variables (recall: we consider only functions where the order of differentiation is inessential)

In two-dimensional notation started above, the Hessian is

$$Hf(x, y) = H(x, y) = \begin{pmatrix} D_{xx}f(x, y) & D_{xy}f(x, y) \\ D_{yx}f(x, y) & D_{yy}f(x, y) \end{pmatrix}$$

For our function $f(x, y) = \frac{1}{64}x^4 + \frac{1}{4}y^2, \quad H = \begin{pmatrix} \frac{3}{16}x^2 & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$

Convexity via Hessian: if Hf is positive definite everywhere, then f is convex

The Newton method

Simply, instead of $\quad x_{k+1} = x_k - \dfrac{f'(x_k)}{f''(x_k)}$

we do

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - H^{-1}(x_k, y_k)G(x_k, y_k)$$

Well... we do not invert any matrix, of course. In fact, the Newton method linearizes the function at a given point; the linearized function is the gradient now, and the linearization is

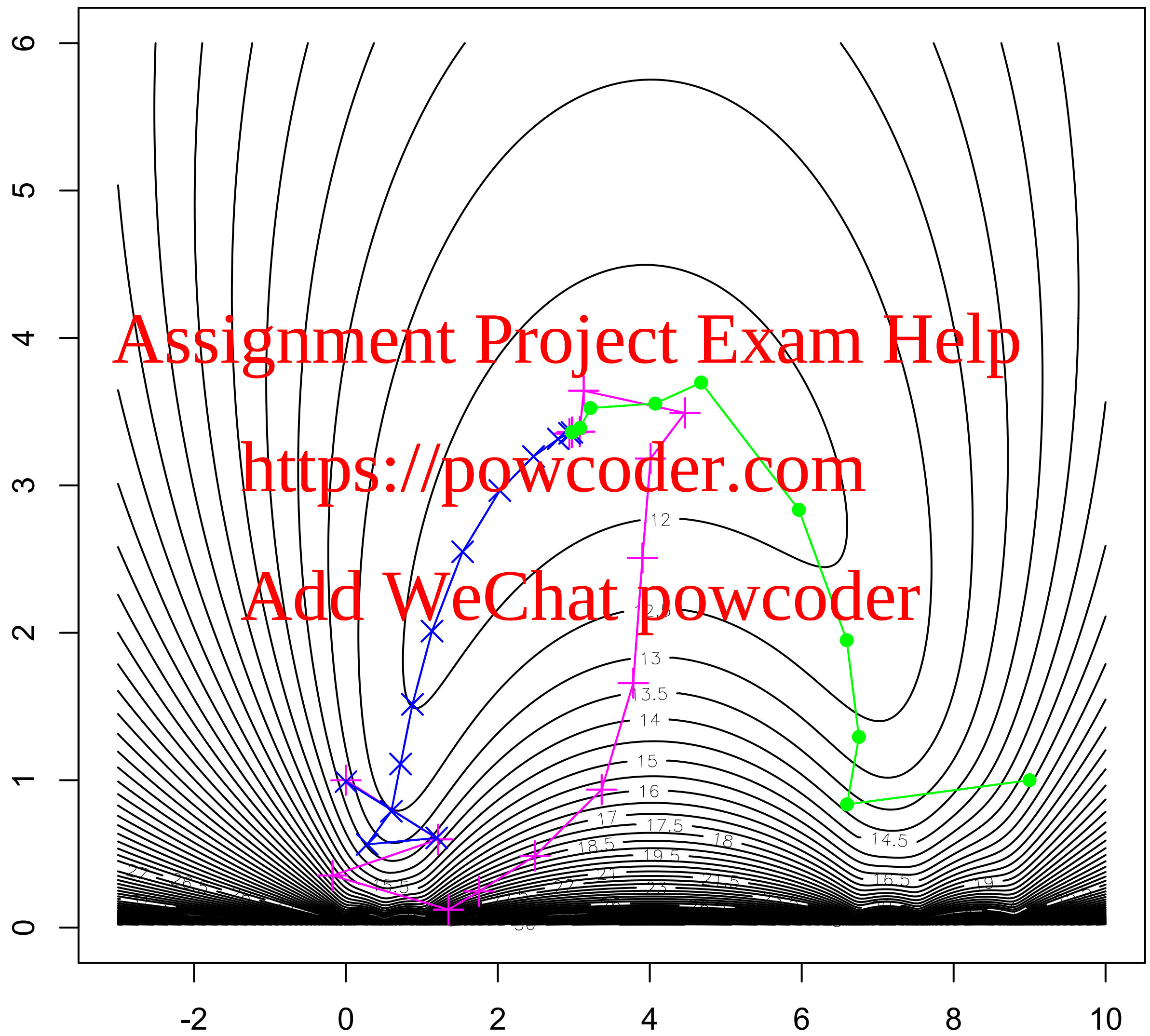
$$H(x_k, y_k)\begin{pmatrix} x - x_k \\ y - y_k \end{pmatrix} + G(x_k, y_k)$$

And now we obtain $(x_{k+1}, y_{k+1})^{\mathsf{T}}$ via solving the linear system

$$H(x_k, y_k)\begin{pmatrix} x - x_k \\ y - y_k \end{pmatrix} + G(x_k, y_k) = O$$ which is the same as

$$H(x_k, y_k)\begin{pmatrix} x \\ y \end{pmatrix} = H(x_k, y_k)\begin{pmatrix} x_k \\ y_k \end{pmatrix} - G(x_k, y_k)$$

Newton demo



Newton method: pros and cons

Newton method, when works, then it works very well: the convergence is fast, the precision good

Newton methods derives from the quadratic approximation of the minimized function at (x_k, y_k)

$$f(x,y) = f(x_k,y_k) + (x,y)G(x_k,y_k) + \frac{1}{2}(x,y)H(x_k,y_k)\begin{pmatrix} x \\ y \end{pmatrix}$$

When the approximation is good, it works: for a quadratic function, Newton method gives the optimum right away

When the approximation is bad, Newton method can fail completely. To this end, it is sometimes improved in several ways: for instance

- adjusting the length of the step (similarly as for gradient descent)

instead of $-H^{-1}(x_k,y_k)G(x_k,y_k)$ taking $-sH^{-1}(x_k,y_k)G(x_k,y_k)$

- forcing $H(x_k,y_k)$ to be positive definite

This may be done by the eigenvalue decomposition - negative eigenvalues are changed to their absolute values. However, the eigenvalue decomposition at each step can considerably slow down the algorithm; and there may be also eigenvalues equal to zero

Compromise methods

Levenberg-Marquardt compromise: originated for the Gauss-Newton method in nonlinear regression

Note:

gradient method: the step is $-s\,G(x_k, y_k)$

Newton method: the step is $-s\,H^{-1}(x_k, y_k)G(x_k, y_k)$

the step is $-s\,(H(x_k, y_k) + \lambda I)^{-1}G(x_k, y_k)$ (where $\lambda \geqslant 0$)

For some λ, the matrix $H(x_k, y_k) + \lambda I$ is positive definite

In such case we have a guarantee: at every iteration, the objective function, the minimized function, *decreases*

Methods with this guarantee are called *descent methods*; gradient descent is just one of them

Other problems

Another problem with the Newton method: we have to provide those derivatives. In p dimensions it means: we have to provide p partial derivatives for the gradient, and then we need another p^2 second partial derivatives... well, not really: only $p(p+1)/2$, but that is still quite a lot for large p

We would like to avoid calculating the Hessian - perhaps via approximating it somehow (recall the secant method in dim 1)

For very large problems, the simple fact of having to have all the Hessian in the computer memory can be an issue. If there are 1000 variables, for instance, then the Hessian has 1000000 elements...

And other methods then

The first issue (having to provide the Hessian) is mitigated by *quasi-Newton methods*: like the secant method in dim 1, they can calculate some approximation - or, better, substitute - of Hessian from past iterations

The *method of conjugate gradients* eliminates even the need of keeping this matrix inside the computer

Both methods provide guarantees when the minimized function is quadratic: in such a case, they converge to the optimum

(Note, however, that in such case the Newton methods converged right away, in one step. But, on the other hand, the methods often behave better in cases when the Newton method fails.)

Quasi-Newton methods

Newton method step (with adjustable length)

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} - \begin{pmatrix} x_k \\ y_k \end{pmatrix} = -s_k H^{-1}(x_k, y_k) G(x_k, y_k)$$

can be generalized to a general scheme

$$d_k = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} - \begin{pmatrix} x_k \\ y_k \end{pmatrix} = -s_k Q_k G(x_k, y_k)$$

Firstly, s_k is calculated so that it gives minimum, in s, of

$$f\left(\begin{pmatrix} x_k \\ y_k \end{pmatrix} - s Q_k G(x_k, y_k)\right)$$

Secondly, Q_k is defined in a way so that the previous step

$$d_{k-1} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - \begin{pmatrix} x_{k-1} \\ y_{k-1} \end{pmatrix} = Q_k \left(G(x_k, y_k) - G(x_{k-1}, y_{k-1})\right)$$

Under these requirements, the method guarantees to find the optimum of a quadratic function

Quasi-Newton methods: the choice

Updates: idea by Davidon, elaborated by Fletcher and Powell (DFP)

$$Q_k = Q_{k-1} + \frac{d_{k-1} \otimes d_{k-1}}{d_{k-1}^T c_k} - \frac{(Q_{k-1}c_k) \otimes (Q_{k-1}c_k)}{c_k Q_{k-1} c_k}$$

where

$$c_k = G(x_k, y_k) - G(x_{k-1}, y_{k-1})$$

and $\otimes$ is the "tensor product": for vectors a and b,

the element of $a \otimes b$ in the i-th row and j-th columns is $a_i b_j$

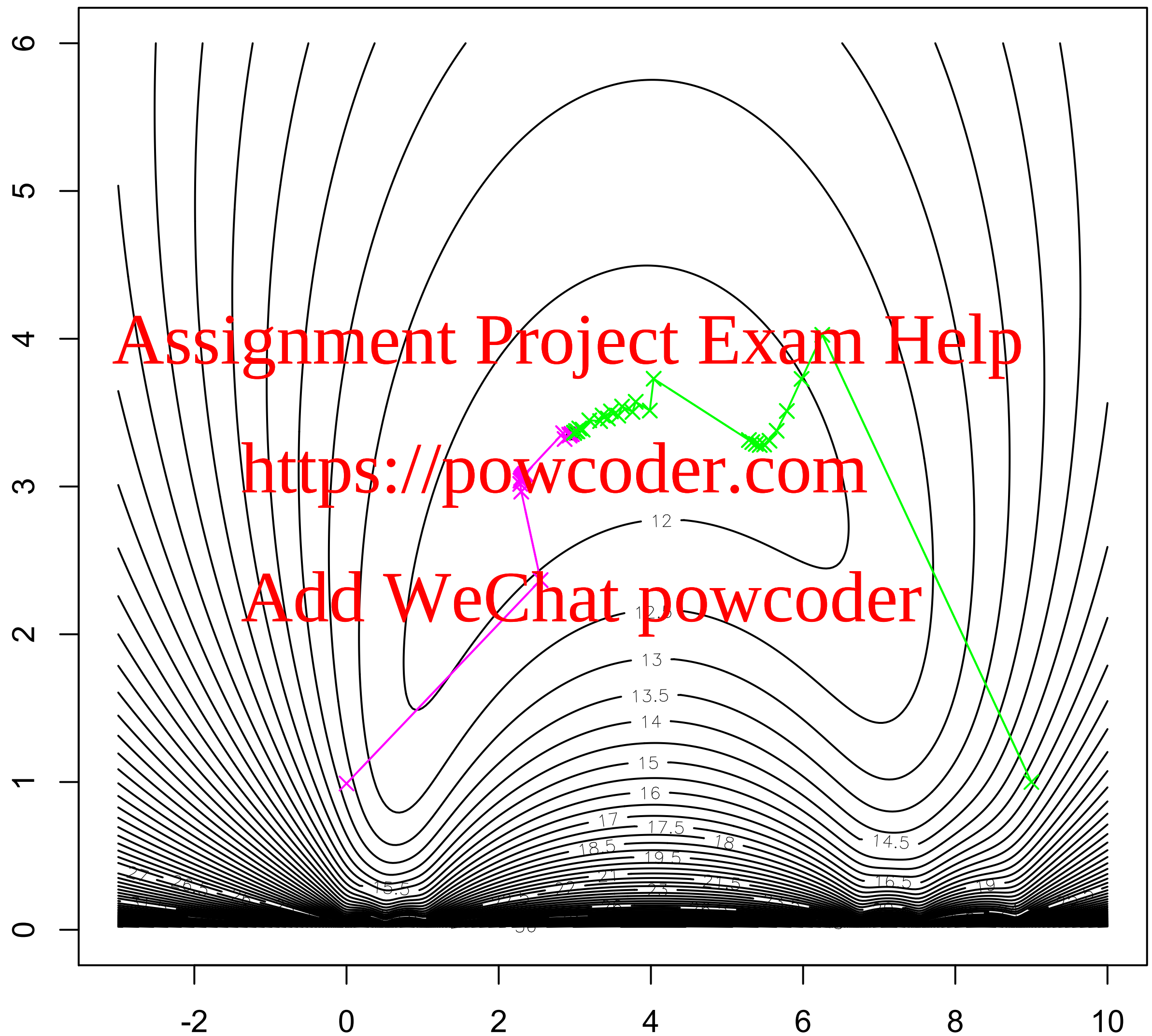
More popular is BFGS methods, named after Broyden, Fletcher, Goldfarb, and Shanno, and possessing also superior engineering details (which we do not discuss here); its updating scheme is

$$Q_k = Q_{k-1} + \frac{d_{k-1} \otimes d_{k-1}}{d_{k-1}^T c_k} - \frac{(Q_{k-1}c_k) \otimes (Q_{k-1}c_k)}{c_k Q_{k-1} c_k} + (c_k Q_k c_k)\, u_k \otimes u_k$$

where

$$u_k = \frac{d_{k-1}}{d_{k-1}^T c_k} - \frac{Q c_k}{c_k Q_k c_k}$$

Yet another demo



Conjugate gradients: principle

The method of gradient descent has

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} + s_k C_k \qquad \text{where } C_k = -G(x_k, y_k)$$

The method of conjugate gradients selects the next C_{k+1} so that

$$C_{k+1}^T Q C_k = 0$$

where Q pertains to some quadratic approximation - which is, however, not calculated; instead, C_k is calculated as a linear combination of C_{k-1} and $G(x_k, y_k)$

Conjugate gradients: details

Given (x_k, y_k) and C_k, we calculate s_k such that

$$\begin{pmatrix} x_k \\ y_k \end{pmatrix} + s\,C_k \qquad \text{is minimal in } s\text{, and then set}$$

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} + s_k C_k \quad \text{and} \quad C_{k+1} = -G(x_{k+1}, y_{k+1}) + t_k C_k$$

Common choices for t_k are

$$t_k = \frac{G(x_{k+1}, y_{k+1})^{\top} G(x_{k+1}, y_{k+1})}{G(x_k, y_k)^{\top} G(x_k, y_k)}$$

$$\text{or} \quad t_k = \frac{(G(x_k, y_k) - G(x_{k+1}, y_{k+1})^{\top} G(x_{k+1}, y_{k+1})}{G(x_k, y_k)^{\top} G(x_k, y_k)}$$

The method starts with $C_1 = -G(x_1, y_1)$, and is reset - returns to it $C_k = -G(x_k, y_k)$ periodically, always after certain fixed number of steps

And yet another demo