Machine Learning

Prof. Jose L. Mendoza-Cortes

Scientific Computing Department, Dirac Science Building
Materials Science and Engineering, High Performance Materials Institute
Florida State University
jmendozacortes@fsu.edu

Condensed Matter Theory, National High Magnetic Field Laboratory
Florida State University
mendoza@magnet.fsu.edu

Chemical and Biomedical Engineering
Florida State University — Florida A&M University — College of Engineering
mendoza@eng.famu.fsu.edu

Web: http://mendoza.eng.fsu.edu/

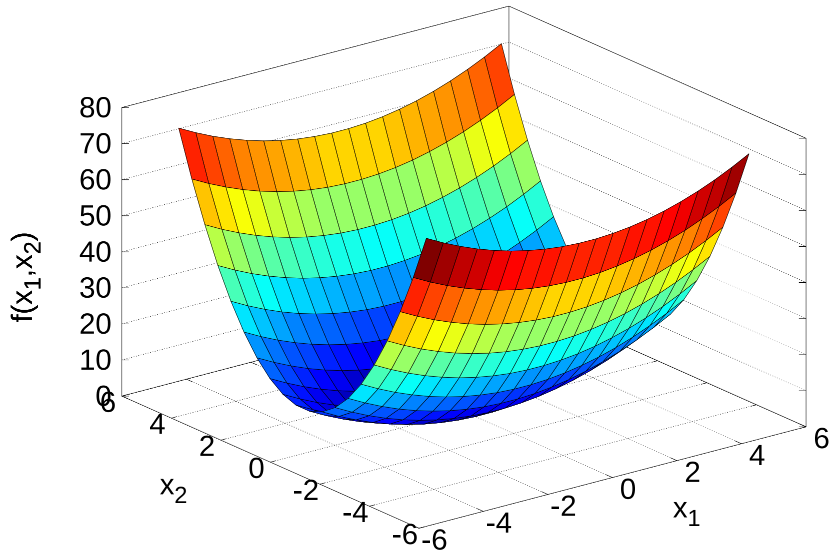
Contents

[1]In this class "multi" = 2, although we will sometimes use more general language

Multidimensional functions

- Instead of just $f(x)$, we will now consider finding the *minima* of functions $f(\mathbf{x}) = f(x_1, x_2, ..., x_n)$
- Example: Consider the 2D function

$$f(x_1, x_2) = 0.5x_1^2 + 2.5x_2^2$$



Multidimensional functions

Code 1: MATLAB's functions for optimization

```
    %Using MATLAB's fminbnd
%clc; clear; close;
figure(4);
f =@(x) -(2*sin(x)-x^2/10); %1D = One Dimension
[x, functionAtOptimum]=fminbnd(f,0,4);
Answer1D = sprintf('The optimal is %f and the function is %f', x, functionAtOptimum)

%Using MATLAB's fminsearch
f=@(x) 2+x(1)-x(2)+2*x(1)^2+2*x(1)*x(2)+x(2)^2; %2D = Two Dimensions
[x,fval]=fminsearch(f,[-0.5,0.5]);
Answer2D = sprintf('The optimal is x1= %f and x2= %f the function is %f \n', x, fval)

%GRAPHICAL Solution
x=linspace(-2,0,40);y=linspace(0,3,40);
[X,Y] = meshgrid(x,y); %Rectangular grid in 2-D and 3-D space
Z=2+X-Y+2*X.^2+2*X.*Y+Y.^2; %The function to plot

subplot(1,2,1);
cs=contour(X,Y,Z);clabel(cs);
xlabel('x_1');ylabel('x_2');title('(a) Contour plot');grid;

subplot(1,2,2);
cs=surfc(X,Y,Z);
zmin=floor(min(Z));zmax=ceil(max(Z));
xlabel('x_1');ylabel('x_2');zlabel('f(x_1,x_2)');title('(b) Mesh plot');
```

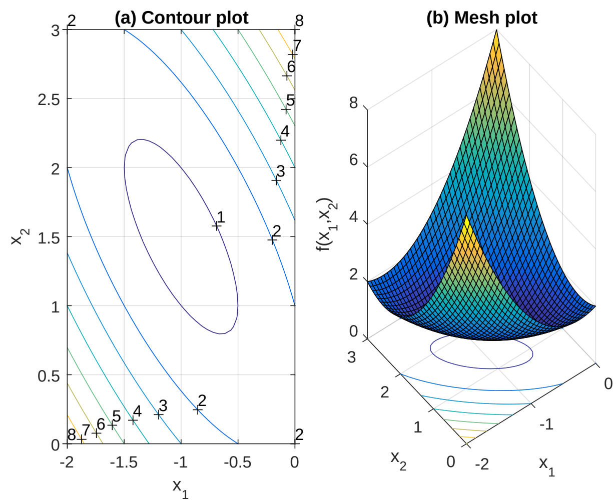
Remember that is always recommended to start with a graphical representation of the function to optimize.

Multidimensional functions

- We considered the 2D function

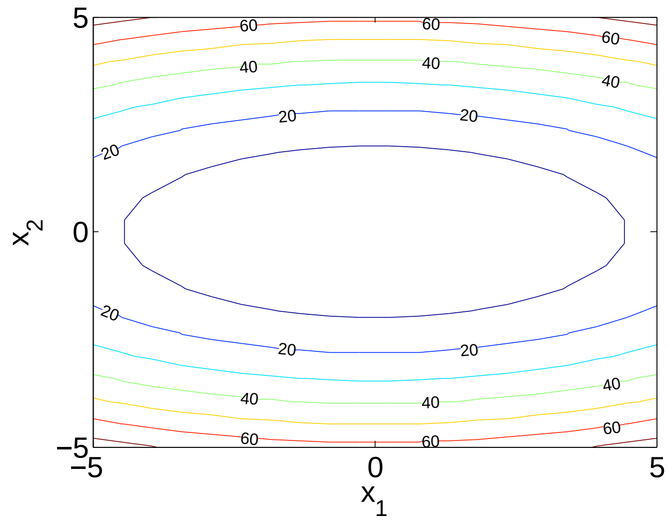
$$Z = 2 + x_1 - x_2 + 2 * x_1^2 + 2 * x_1 * x_2 + x_2^2;$$

$$Z = 2 + X - Y + 2 * X^2 + 2 * X * Y + Y^2$$



2D Function

- We can also construct a contour plot



- An important concept in multidimensional optimization is the gradient of $f(\mathbf{x})$. For a 2D function such as the one here:

$$\nabla f(\mathbf{x}) = \frac{\partial f}{\partial x_1}\mathbf{e}_1 + \frac{\partial f}{\partial x_2}\mathbf{e}_2$$

Gradient

- The gradient generalizes the concept of derivative to multiple dimensions
- Note that it is a vector, and has a "direction" in addition to a magnitude. This direction is important in optimization.
- We can also write it as a vector

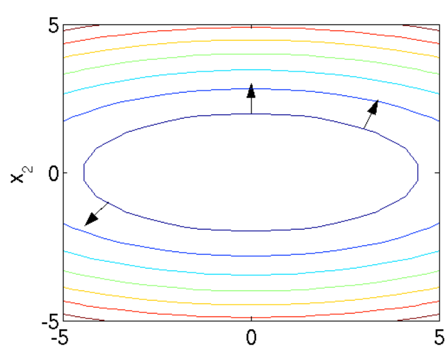
$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}, \quad \text{assuming } \mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

- We can evaluate the gradient of this particular function:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} x_1 \\ 5x_2 \end{bmatrix} = x_1\mathbf{e}_1 + 5x_2\mathbf{e}_2$$

Gradient

- I plotted the direction of the gradient vector at a few different points on the contour map



- The three points and the corresponding normalized gradients are $(\mathbf{x}, \nabla f(\mathbf{x})/||\nabla f(\mathbf{x})||)$

$$\left(\begin{bmatrix} 0.0 \\ 2.0 \end{bmatrix}, \begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix}\right), \quad \left(\begin{bmatrix} 3.0 \\ 1.5 \end{bmatrix}, \begin{bmatrix} 0.37 \\ 0.93 \end{bmatrix}\right), \quad \left(\begin{bmatrix} -3.8 \\ 1.0 \end{bmatrix}, \begin{bmatrix} -0.61 \\ -0.80 \end{bmatrix}\right)$$

Plotting Gradients: Matlab Code

```
clear; clc; close;
x = linspace(-5,5,20);y = linspace(-5,5,20);

[X, Y] = meshgrid(x,y);Z = 0.5*X.^2 + 2.5*Y.^2;
[DX,DY] = gradient(Z,.5,.5);

figure(1);
surf(X,Y,Z); hold on; quiver(X,Y,DX,DY); axis tight; hold off

figure(2);
contour(X,Y,Z); hold on; quiver(X,Y,DX,DY); axis tight; hold off

figure(3)
[C,h] = contour(X,Y,Z);
set(h,'ShowText','on','TextStep',get(h,'LevelStep')*2)
xlabel('x_1'); ylabel('x_2'); zlabel('f(x_1,x_2)')
axis tight; grid minor; hold on;

x1 =[0, 2];   dx1 = [x1(1),5*x1(2)]; dx1 = dx1/norm(dx1);
quiver( x1(1,1), x1(1,2), dx1(1,1), dx1(1,2),'m','MaxHeadSize',1)

x2 =[3, 1.5];  dx2 = [x2(1),5*x2(2)]; dx2 = dx2/norm(dx2);
quiver( x2(1,1), x2(1,2), dx2(1,1), dx2(1,2),'m','MaxHeadSize',1)

x3 =[-3.8,-1];dx3 = [x3(1),5*x3(2)]; dx3 = dx3/norm(dx3);
quiver( x3(1,1), x3(1,2), dx3(1,1), dx3(1,2),'m','MaxHeadSize',1)

set(gca,'FontSize',18);hold off
%print -dpdf -FHelvetica:18 2dplot.pdf
%print(h,'-dpng','2dcontour.png')
```


Gradient

- Note that the gradient is perpendicular to contour lines
- The direction of ∇f tells us which way to travel in to gain elevation as quickly as possible
- The magnitude of ∇f tell us how much we gain by travelling in that direction
- This is similar to derivative of a single variable $f(x)$ where df/dx measures the “rate” at which $f(x)$ changes with x.
- Next we are going to consider a method called “steepest descent” to find the *minima* of a function $f(\mathbf{x})$ by travelling in the direction of $-\nabla f(\mathbf{x})$
- There is completely analogous method called “steepest ascent” to find the maxima by travelling in the direction of $\nabla f(\mathbf{x})$

Steepest Descent: Algorithm (Pseudocode)

1. $k = 0$; $\mathbf{x}_0 =$ initial guess
2. Compute the -ve gradient $\mathbf{s}_k = -\nabla f(\mathbf{x}_k)$
3. Choose α_k to minimize $f(\mathbf{x}_k + \alpha \mathbf{s}_k)$. Note that this is a 1D problem, for which we have devised methods before. This is called a *line* search.
4. Update the solution: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{s}_k$
5. Set $k = k + 1$, and go back to step 2, and repeat until convergence.

Example

Problem: Use steepest descent to minimize the function[2]

$$f(\mathbf{x}) = 0.5x_1^2 + 2.5x_2^2,$$

where $\mathbf{x} = [x_1,\ x_2]^T$, and whose gradient is:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} x_1 \\ 5x_2 \end{bmatrix}$$

Start with an initial guess of $\mathbf{x}_0 = [5,\ 1]^T$.

Solution:

$$\mathbf{x}_0 = [5,\ 1]^T, \quad \mathbf{s}_0 = -\nabla f(\mathbf{x}_0) = -[5,\ 5]^T$$

Therefore,

$$f(\mathbf{x}_0 + \alpha \mathbf{s}_0) = f\left(\begin{bmatrix} 5 \\ 1 \end{bmatrix} - \alpha \begin{bmatrix} 5 \\ 5 \end{bmatrix}\right) = f\left(\begin{bmatrix} 5 - 5\alpha \\ 1 - 5\alpha \end{bmatrix}\right)$$

[2] Problem from Heath, chapter 6.

Example

Thus,

$$f(\mathbf{x}_0 + \alpha \mathbf{s}_0) = 0.5(5 - 5\alpha)^2 + 2.5(1 - 5\alpha)^2$$
$$= 75\alpha^2 - 50\alpha + 15$$

One can easily find the minima of this function by taking the derivative with respect to α which gives us $150\alpha_0 - 50 = 0$, or $\alpha_0 = 1/3$.

Thus,

$$\mathbf{x}_1 = \mathbf{x}_0 + (1/3)\mathbf{s}_0 = \begin{bmatrix} 3.333 \\ -0.667 \end{bmatrix}$$

We can now repeat the process until we are happy!

Or we can write the following matlab code.

Steepest Descent: Matlab Code

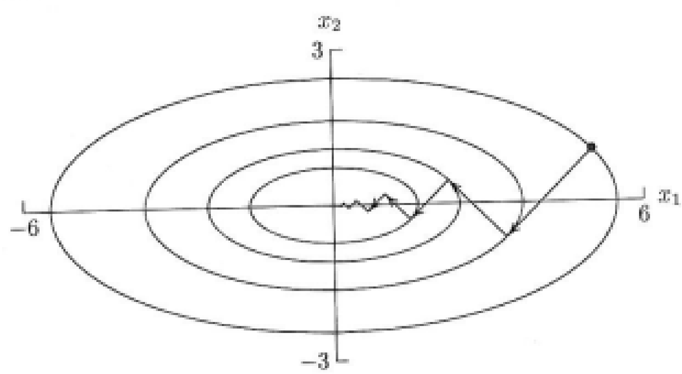
Code 2: SteepDescent.m

```
% Steepest Descent Demo: use [x f n] = SteepDescent([5;1], 1e-3)

function [xopt fopt nopt] = SteepDescent(x0, tol)

  x = x0; %Step 1: Here we setup the intial guess
  k = 0;

  while(norm(gradf(x)) > tol)      % Need to make gradf ~ 0

    s      = -gradf(x); %Step 2: calcluate the gradient of f
    falpha = @(alpha) f(x + alpha*s); %Step 3 define function for calulating best value of alpha
    alpha  = fminsearch(falpha,0.1); %find best value of alpha
    x      = x + alpha * s; %Step 4 %update new value of x
    k      = k + 1; %Step 5 : increase iteration count

  end

  xopt = x;   fopt = f(x); nopt = k; %Set up output for optimal vector, minimal value and number of iterations

end

function Z = f(x)
  Z = 0.5*x(1)^2 + 2.5*x(2)^2;
end

function Z = gradf(x)
  Z = [x(1);5*x(2)];
end
```

Example

From Heath pg 278. What is the geometrical interpretation?

k	x_k^T		$f(x_k)$	$\nabla f(x_k)^T$	
0	5.000	1.000	15.000	5.000	5.000
1	3.333	−0.667	6.667	3.333	−3.333
2	2.222	0.444	2.963	2.222	2.222
3	1.481	−0.296	1.317	1.481	−1.481
4	0.988	0.198	0.585	0.988	0.988
5	0.658	−0.132	0.260	0.658	−0.658
6	0.439	0.088	0.116	0.439	0.439
7	0.293	−0.059	0.051	0.293	−0.293
8	0.195	0.039	0.023	0.195	0.195
9	0.130	−0.026	0.010	0.130	−0.130



Hessian

- The Hessian of a multidimensional scalar function $f(\mathbf{x})$ is given by the symmetric square matrix

$$\mathbf{H}_f(\mathbf{x}) = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

- Just as the gradient generalizes df/dx, the Hessian generalizes d^2f/dx^2.

Hessian

- In 2D, the Hessian is:

$$\mathbf{H}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \, \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \, \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix}$$

- Recall how $d^2 f(x^*)/dx^2$ told us whether x^* (obtained by solving $df/dx = 0$) was a maxima, minima or a saddle point
- The Hessian does the same job. If $\mathbf{x}^*$ is a solution to $\nabla f(\mathbf{x}) = \mathbf{0}$, then if $\mathbf{H}_f(\mathbf{x}^*)$ is

$$\begin{array}{lll} \text{+ve definite} & \Longrightarrow & x^* \text{ is a minima} \\ \text{-ve definite} & \Longrightarrow & x^* \text{ is a maxima} \\ \text{indefinite} & \Longrightarrow & x^* \text{ is a saddle point} \end{array}$$

Newton's Method

- Recall 1D Newton's Method for optimization:

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)}$$

- We could rewrite this expression as:

$$(x_{i+1} - x_i) f''(x_i) = -f'(x_i)$$

- We are going to generalize this method for multiple dimensions
- It is useful to visualize Newton's method as a quadratic approximation to a Taylor's series

Newton's Method

- That is consider

$$f(x+s) \approx f(x) + \frac{df}{dx}s + \frac{1}{2}\frac{d^2f}{dx^2}s^2.$$

- We can think of the RHS as a quadratic function in s which can be minimized

$$\frac{df(x+s)}{ds} = 0 \implies 0 + \frac{df}{dx} + \frac{d^2f}{dx^2}s = 0$$

- Leading to

$$s\frac{d^2f}{dx^2} = -\frac{df}{dx}$$

which is the same as Newton's method for optimization

Newton: Multidimensional case

- We can repeat the Taylor series expansion for $f(\mathbf{x})$.

$$f(\mathbf{x}+\mathbf{s}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^T\mathbf{s} + \frac{1}{2}\mathbf{s}^T\mathbf{H}_f(\mathbf{x})\mathbf{s}$$

Note that for 1D this collapses to the previous expression.

- Taking the derivative, leads us to:

$$\mathbf{H}_f(\mathbf{x})\mathbf{s} = -\nabla f(\mathbf{x})$$

Compare with the 1D case:

$$s\frac{d^2f}{dx^2} = -\frac{df}{dx}$$

- This allows us to write an algorithm for Newton's method

Newton's Method: Algorithm

1. $k = 0$; $\mathbf{x}_0 =$ initial guess
2. Compute the gradient $\nabla f(\mathbf{x}_k)$ and the Hessian $\mathbf{H}_f(\mathbf{x}_k)$
3. Solve $\mathbf{H}_f(\mathbf{x}_k)\mathbf{s}_k = -\nabla f(\mathbf{x}_k)$ for $\mathbf{s}_k$
4. Update the solution: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
5. Set $k = k + 1$, and go back to step 2, and repeat until convergence.

Example

Problem: Solve the previous example again, this time using Newton's method

$$f(\mathbf{x}) = 0.5x_1^2 + 2.5x_2^2,$$

Solution:
The gradient and Hessian are given by:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} x_1 \\ 5x_2 \end{bmatrix}, \quad \mathbf{H}_f(\mathbf{x}_k) = \begin{bmatrix} 1 & 0 \\ 0 & 5 \end{bmatrix}$$

We solve the system (with $\mathbf{x}_0 = [5,\ 1]^T$)

$$\begin{bmatrix} 1 & 0 \\ 0 & 5 \end{bmatrix} \mathbf{s}_0 = -\begin{bmatrix} 5 \\ 5 \end{bmatrix} \implies \mathbf{s}_0 = \begin{bmatrix} -5 \\ -1 \end{bmatrix}$$

Example

- This implies
$$\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{s}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
which is the true solution
- It is not surprising that Newton's method converged in 1 iteration since the $f(\mathbf{x})$ was quadratic
- In general the convergence rate is quadratic, but the method can veer off unless we start close enough to the solution
- Note: No line search required, but we had to determine a Hessian matrix and solve a linear system at each iteration
- In damped Newton methods, a line search is added to make the method more robust.

Quasi-Newton Methods

- Newton's method converges rapidly once you are close to the solution. But it doesn't come cheap.
- For a n-dimensional problem, each iteration requires $\mathcal{O}(n^2)$ function evaluations to form the gradient and the Hessian, and $\mathcal{O}(n^3)$ operations to solve the linear system.
- To reduce overhead, quasi-Newton methods have been developed which seek to replace the step:
$$\mathbf{H}_f(\mathbf{x}_k)(\mathbf{x}_{k+1} - \mathbf{x}_k) = -\nabla f(\mathbf{x}_k)$$
with
$$\mathbf{B}_k(\mathbf{x}_{k+1} - \mathbf{x}_k) = -\alpha_k \nabla f(\mathbf{x}_i)$$
where $\mathbf{B}$ is an approximation to the Hessian matrix, and may be obtained by secant updating and α_k is a line-search parameter.

BFGS Method

- A popular secant updating method named after its co-inventors: Broyden, Fletcher, Goldfarb and Shanno.
- Initially set $\mathbf{B}_0 = \mathbf{I}$, which means the first step is in the negative gradient direction (like steepest descent).
- Unlike Newton's method, the second derivatives (Hessian) do not have to be pre-computed.
- It is built up over time.
- Convergence is superlinear.
- We consider a simple algorithm (better implementations update a factorization of the matrix $\mathbf{B}$ rather than the matrix itself)

BFGS Algorithm

1. $k = 0$; $\mathbf{x}_0$ = initial guess
2. Set $\mathbf{B}_0 = \mathbf{I}$ as the initial Hessian approximation
3. Compute the gradient $\nabla f(\mathbf{x}_k)$
4. Solve $\mathbf{B}_k \mathbf{s}_k = -\nabla f(\mathbf{x}_k)$ for $\mathbf{s}_k$
5. Update the solution: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$
6. Set $\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$
7. Update the Hessian

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} - \frac{\mathbf{B}_k \mathbf{s}_k \mathbf{s}_k^T \mathbf{B}_k}{\mathbf{s}_k^T \mathbf{B}_k \mathbf{s}_k}$$

8. Set $k = k + 1$, and go back to step 3, and repeat until convergence.

Example

Problem: Solve the previous example again, this time using BFGS method

$$f(\mathbf{x}) = 0.5x_1^2 + 2.5x_2^2,$$

Solution:
The gradient and approximate Hessian are given by:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} x_1 \\ 5x_2 \end{bmatrix}, \quad \mathbf{B}_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

We solve the system (with $\mathbf{x}_0 = [5,\ 1]^T$).
The first step is simply: $\mathbf{I}\mathbf{s}_0 = -\nabla f(\mathbf{x}_0) = -[5,\ 5]^T$
$\implies \mathbf{x}_1 = [5,\ 1]^T + [-5,\ -5]^T = [0,\ -4]^T$
We can update the approximate Hessian to

$$\mathbf{B}_1 = \begin{bmatrix} 0.667 & 0.333 \\ 0.333 & 0.667 \end{bmatrix}$$

Example

We can continue to get the sequence:

k	x_1	x_2	$f(\mathbf{x})$
0	5.0000	1.0000	15.0000
1	0.0000	-4.0000	40.0000
2	-2.2222	0.4444	2.9630
3	0.8163	0.0816	0.3499
4	-0.0092	-0.0153	0.0006
5	-0.0005	0.0009	0.0000

using the following Matlab code:

BFGS method: Matlab Code

Code 3: Master Script that calls the function BFGS

```
clc; clear;close

f = @(x) (1 - x(1)).^2 + 100.*((x(2) - x(1).^2).^2);
[ min_x, fx ] = BFGS( [0; 1], 1e-3 )

%If the given initial guess of [-1; 1] is used, the output is said to be
%not a number. However, using an initial guess of [0; 1] outputs the
%correct values. The output value using the BFGS method is given as well to
%prove the function evaluated at that point is close to zero.

[ xopt,fopt,nopt ] = BFGS_Alt( [-1;1],1e-8,1000 );
```


BFGS method: Matlab Code

Code 4: Function BFGS to find critical points

```
function [ x, fx, iter ] = BFGS( x0, tol )
k = 0; x = x0; %STEP 1 - set k = 0 and x as initial guess
n = length(x); B = eye(n); %STEP 2 - set B as the initial Hessian approximation, given as an identity matrix with the same length as the guess

while( norm(gradf(x)) > tol ) %while the greatest value of the gradient is greater than the tolerance value
    df = gradf(x); %STEP 3 - as defined beginning in line 22, the initial gradient is saved as variable df to use later on in conjunction with the gradient
    s = B\(-df) ; %STEP 4 - solve B(k)*s(k) = -df(x_k) for s which results in this equation
    x = x + s ; %STEP 5 - updates the solution periodically with x(k+1) = x(k) + s(k)
    y = gradf(x) - df; %STEP 6 - sets y(k) = gradf(x_k+1) - df(x_k)
    B = B + ( y*y')/(y'*s) - (B*s*s'*B)/(s'*B*s); %STEP 7 - updates the Hessian with the appropriate formula
    k = k + 1 ; %STEP 8 - sets k = k + 1 as the last step before the loop repeats
end
fx = f(x);
iter = k;
end

function Z = f(x)
Z = (1 - x(1)).^2 + 100.*((x(2) - x(1).^2).^2);
end

function Z = gradf(x)
Z = [10*(x(2) - x(1)^2); (1 - x(1))];
end
```

BFGS method: Matlab Code

Code 5: Function BFGS (alternative) to find critical points

```
function [ xopt,fopt,nopt ] = BFGS_Alt( x0,eApprox,itermax )
x=x0; n=length(x); B=eye(n); iter=0;

for i=1:itermax
df=gradf(x);
s=B\(-df); %\ is dividing two matrices
x=x+s;
y=gradf(x)-df; %step 6 in algorithim
B=B+(y*y')/(y'*s)-(B*s*s'*B)/(s'*B*s); %step 7, complicated formula
iter=iter+1;
if norm(gradf(x)) < eApprox
    break, end
end
xopt=x; fopt=f(x); nopt=iter;
end

function Z=f(x)
Z=(1-x(1))^2+100*(x(2)-x(1)^2)^2; %this is the function to evaluate from the example
     problem.
end

function Z=gradf(x)
Z=[2*(-1+x(1)+200*x(1)^3-200*x(1)*x(2)); 200*(-x(1)^2+x(2))];
end
```


Summary

- Methods for optimization in 1D have "counterparts" in methods for the solution of nonlinear equations:

$$\begin{aligned} \text{Golden Search} &\rightarrow \text{Bisection} \\ \text{Parabolic Interpolation} &\rightarrow \text{Regula Falsi} \\ \text{Newton } (f(x)=0) &\rightarrow \text{Newton } (f'(x)=0) \end{aligned}$$

  and resemble many of their properties (linear/quadratic convergence etc.).
- Multidimensional optimization requires knowledge of gradients and sometimes Hessians, which generalize first and second order derivatives.
- Steepest descent moves in the direction of negative gradient - results in zig-zag moves, which are "fixed" in the conjugate-gradient method.

Summary

- Multidimensional Newton's method generalizes Newton's method for optimization in 1D. It is fast, but requires significant work (deriving the Hessian, and solving a linear system).
- BFGS is an extremely popular secant-updating method which works with an approximate Hessian.

Extra Material: Not evaluated

Not evaluated in Exams
Not evaluated in Quizzes
Not evaluated in Homeworks

This is just extra material in case you want to know more about related topics.

Appendix (not evaluated in exams or quizzes)

Code 6: Random Search Method for Optimization

```
clc; clear;
maxf = -1e9; %A very negative value
itermax=[1000]; %iterations
for j = 1:itermax
x = -2+4*rand; %Notice the random generator 'rand'
y = 1+2*rand;  %Notice the random generator 'rand'
fn = y - x - 2.*x.^2-2.*x.*y - y.^2;
 if fn > maxf %We save the largest value among the trials
    maxf = fn;
    maxx = x;
    maxy = y;
 end
end %Next j

format shortG
disp(['Iterations          x             y             f(x,y)'])
disp([itermax,          maxx,      maxy, maxf])
```

This will not be evaluated in exams or quizzes but I thought it is a very interesting thing to know. This is the essence of Monte Carlo Algorithms used in the financial and other industries.

Conjugate Gradient

- Another alternative to Newton's method, that doesn't require explicitly require the calculation of the Hessian
- In fact, unlike secant-updating methods like BFGS, it doesn't even require the storage of an approximate Hessian. This makes it suitable for very large problems.
- It resembles the method of steepest descent, but avoids the zig-zag pattern (repeated alternate searching in directions previously explored) by removing components from previous directions.
- The algorithm for a particular version of the CG-algorithm due to Fletcher and Reeves is described next.

Conjugate-Gradient Algorithm

1. $k = 0$; $\mathbf{x}_0 =$ initial guess
2. Compute the gradient $\mathbf{g}_0 = \nabla f(\mathbf{x}_k)$
3. Set $\mathbf{s}_0 = -\mathbf{g}_0$
4. Perform a line search. Choose α_k to minimize $f(\mathbf{x}_k + \alpha_k \mathbf{s}_k)$.
5. Update the solution: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{s}_k$
6. Set $\mathbf{g}_{k+1} = \nabla f(\mathbf{x}_{k+1})$
7. Set $\beta_{k+1} = (\mathbf{g}_{k+1}^T \mathbf{g}_{k+1})/(\mathbf{g}_k^T \mathbf{g}_k)$
8. Set $\mathbf{s}_{k+1} = -\mathbf{g}_{k+1} + \beta_{k+1} \mathbf{s}_k$
9. Set $k = k + 1$, and go back to step 4, and repeat until convergence.

Example

Problem: Solve the previous example again, this time using conjugate-gradient method

$$f(\mathbf{x}) = 0.5x_1^2 + 2.5x_2^2,$$

Solution: We can easily repurpose our old code for steepest descent for this problem.

Note that this method requires a line search, which is carried out by using the intrinsic matlab function `fminsearch`.

Conjugate Gradient: Matlab Code

```
% Conjugate-Gradient Demo: [x f n] = conjGrad([5;1],1e-3)

function [xopt fopt nopt] = conjGrad(x0, tol)

x = x0;
k = 0;
g = gradf(x);
s = -g;

while(norm(gradf(x)) > tol)
falpha = @(alpha) f(x + alpha*s);
alpha  = fminsearch(falpha,0.1);
x      = x + alpha * s
beta   = (g'*g);
g      = gradf(x);
beta   = (g'*g)/beta;
s      = -g + beta * s;
k      = k + 1;
end

xopt = x;   fopt = f(x);   nopt = k;

end

function Z = f(x)
Z = 0.5*x(1)^2 + 2.5*x(2)^2;
end

function Z = gradf(x)
Z = [x(1);5*x(2)];
end
```


Appendix: Scripts included

Try these commands in your own workstation, i.e. have the lectures on one half side of your screen and Matlab/Octave-GUI on the other half.

Check the scripts/functions under the directory for this note number (X): /NX_Notes_directory