Optimization Algorithms, Implementations and Discussions (technical report for self-reference)

By : Lam Ngok

Introduction
In this preliminary optimization study we tested and implemented six different optimization algorithms, three of them are for solving unconstrained optimization problems, the remaining three are for solving constrained optimization problems. The objective is to get myself familiar with some of the most common optimization algorithms in the literature and to get the hands dirty by implementing these six algorithms. The algorithms for solving unconstrained optimization problems are
1) Steepest descent algorithm
2) Conjugate gradient algorithm
3) Secant algorithm

The algorithms for solving constrained optimization problems are
1) Courant’s penalty method
2) Multiplier penalty method
3) Lagrange-Newton method

We end up with six big modules each of which includes a number of sub-modules. We test all these finished modules on different sets of problems to see how them perform and their strong points and their weak points. We shall discuss the results achieved in the following sections.

The algorithms and problem studied.
We assume the cost functions and constraint functions (if any) are always entered as strings in our programs. For unconstrained problems (algorithm 1-3) we expect the user to input a string containing the cost function and a column vector \( x \) that contains the initial point to start from. For constrained problem we expect the user to enter a cost function in a string, all the constraint functions in a string of array (actually it is like a set i.e. \( c\{1\}=’constraint function 1’,c\{2\}=’constraint function 2’,…etc \)), an initial starting point \( x \), and a number stating the number of constraints.

We use finite difference approximation for the gradient throughout the project, and we also use armijo step size rule to decide the suitable step size.

We studied 2 problems for unconstrained problems and test each of them with
different starting points. Here below are the results for them. We shall denote f(x) as the cost function of the problems throughout our discussion.

The first problem we studied is $f(x) = -x_1^3 + x_1^4 + x_2^2$ (non-convex), the interesting part of this problem is that we have an $x_1$ with powers of 3 and 4, since we know this is an minimization problem, we can see immediately if we make $x_1$ negative enough, $x_1^3$ will become very negative, but then $x_1^4$ may become very positive, so the optimal solution must be at a value of $x_1$ such that $-x_1^3 + x_1^4$ is minimized, $x_2^2$ is always non-negative so $x_2$ is in fact irrelevant here, because we know the optimal value of $f(x)$ must have $x_2=0$. We did some simple calculation and realized there are two points with gradient of 0 for this problem. One of them is at [0 0] the other of them is at [-3/4 0]. We plot a graph as shown below (with f(x) against $x_1$ for the reason of simplicity). The Y-axis below corresponding to the $f(x_1)$ value and X-axis corresponding to the $x_1$ value, we can see at $x_1 = -0.75$ and 0, the slope is indeed zero, but the point at 0 is in fact a point of inflexion and it is not an optimal, the real optimal point is at -0.75. With this in mind we do the experiment on our algorithms to see if they can find the real optimal. In fact we deliberately construct this special problem just to see whether the algorithms will stop at the real optimal or the point of inflexion. This will in some way illustrate how the algorithms behavior for a non-convex cost function.
We start with the point \([1;2]\) (i.e. \(x_1=1, x_2=2\) at the first iteration and get the following results:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(f(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steepest descent</td>
<td>0.0054</td>
<td>0.0000</td>
<td>1.6056e-007</td>
</tr>
<tr>
<td>Conjugate gradient</td>
<td>-0.7500</td>
<td>0.0000</td>
<td>-0.1055</td>
</tr>
<tr>
<td>Secant</td>
<td>0.0024</td>
<td>0.0000</td>
<td>1.4020e-008</td>
</tr>
</tbody>
</table>

It is easy to see that while the Steepest descent and Secant algorithm converge to the point of inflexion from the starting point \([1;2]\), Conjugate gradient successfully converge to the real optimal point.

We also gather the convergence pictures as shown below, please be minded that the initial points are not displaced in the convergence graphs. we can see that conjugate gradient method not only converges to the real optimal, but it also is the fastest, requiring less than half of the iteration number of the steepest descent algorithm.
We then study the same problem by letting the algorithms to start at point \([100;100]\).
which is much farther to the real optimal. Here below are the results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$f(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steepest descent</td>
<td>-0.750</td>
<td>0.000</td>
<td>-0.1055</td>
</tr>
<tr>
<td>Conjugate gradient</td>
<td>-0.750</td>
<td>0.000</td>
<td>-0.1055</td>
</tr>
<tr>
<td>Secant</td>
<td>0.0047</td>
<td>0.000</td>
<td>1.0234e-007</td>
</tr>
</tbody>
</table>

This time both the steepest descent and the conjugate gradient algorithms converge to the optimal point, but the secant algorithm still just converge to the point of inflexion. The following are the convergence graphs we captured. It can be easily seen that the convergence speed is roughly similar for all the three algorithms, with the steepest descent algorithm being slightly faster than the rest of two. This example shows that under non-convex cost functions, the optimization algorithm indeed can not guarantee arriving at a local optimal point ([0 0]' is not local optimal as the neighboring points with smaller $x_1$ will be better than it.).
Plot of the convergence rate for conjugate gradient method

Plot of the convergence rate for secant method
Now we study another unconstrained problem with the three algorithms, the cost functions is $f(x) = x_1^x(x_2^2)$, this is an interesting problem, because it is not a polynomial function. It is easy to see that the minimum value of this cost function is $-\infty$, which can be achieved by letting $x_2^2$ to go to an odd number and $x_1$ to go to $-\infty$. The gradient of this function is very complicated and is equal to $[x_2^2x_1^x(x_2^2-1)2x_2^2x_1^x(x_2)\log(x_1)]$. It is easy to see there are many points which will make this gradient to be zero, although not all of them are local minimum points. We test the starting point from point [1;2] and the results are shown below:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$f(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steepest descent</td>
<td>0.0381</td>
<td>2.1619</td>
<td>2.3257e-007</td>
</tr>
<tr>
<td>Conjugate gradient</td>
<td>0.0346</td>
<td>2.1313</td>
<td>2.2992e-007</td>
</tr>
<tr>
<td>Secant</td>
<td>0.1179</td>
<td>2.5717</td>
<td>7.2301e-007</td>
</tr>
</tbody>
</table>

We can see that both steepest descent and conjugate gradient yields similar result, the secant algorithm gives the worst result. All the three algorithms fail to give the local minimum as the final solution (the cost function $f(x)$ will have smaller value if $x_1$ is smaller for all the three solutions). This again shows our algorithms are not very effective for non-convex problems. The convergence rate graphs are shown below:
It can be seen that conjugated gradient and secant algorithms reach the final solution in similar number of iterations, while the steepest gradient algorithm is much slower,
We were unable to conduct study with starting points of larger value, because the function \( f(x) = x_1^{x_2}(x_2^2) \) can explode easily with even moderate values of \( x_1 \) and \( x_2 \) (especially for \( x_2 \)), and that makes our gradient finding algorithm (the finite difference gradient algorithm) fail.

We then turn to study the constrained optimization problems, like before we study two sets of problems, the first set of problem is the one post on webct, and it is of the form \( f(x) = -x_1 - x_2, \ C_1 = 1 - x_1^2 - x_2^2 = 0 \). We test the algorithm on two initial points, the first initial point is \([0;0]\), please be minded this is in fact an infeasible point (in fact both the two starting points are infeasible). Here below are the result achieved

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( f(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courant’s penalty</td>
<td>0.7102</td>
<td>0.7102</td>
<td>-1.4205</td>
</tr>
<tr>
<td>Multiplier-penalty</td>
<td>0.7056</td>
<td>0.7056</td>
<td>-1.4112</td>
</tr>
<tr>
<td>Lagrange-Newton</td>
<td>0.7073</td>
<td>0.7087</td>
<td>-1.4159</td>
</tr>
</tbody>
</table>

By simple calculation we found that the real optimal point is at \([1/2^{1/2};1/2^{1/2}]\), and the minimum value is at \(~1.4142\). All the three algorithms give similar results that are close to the real optimal point. If we set the stopping condition stricter, the final point will be even more close to the real optimal point.

Here below are the convergence plots for the algorithms
we see that Courant-penalty and multiplier-penalty algorithms converge in similar number of iterations, but the Lagrange-Newton algorithm converge much slower. In
fact the Lagrange matrix is extremely ill conditioned when the starting point is \([0;0]\), and it becomes almost like a singular matrix.

We also conduct studies on a second starting point \([1;2]\) just to see how would the performance difference for different initial values of \(x\). the following are the results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(f(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courant’s penalty</td>
<td>0.7102</td>
<td>0.7102</td>
<td>-1.4205</td>
</tr>
<tr>
<td>Multiplier-penalty</td>
<td>0.7056</td>
<td>0.7056</td>
<td>-1.4112</td>
</tr>
<tr>
<td>Lagrange-Newton</td>
<td>0.4461</td>
<td>0.8960</td>
<td>-1.3421</td>
</tr>
</tbody>
</table>

We can see that the Lagrange-Newton algorithm gives a much larger \(f(x)\) than before, the other two algorithm remains similar. The convergence rate graphs are similar with the previous case, with the exception of the Lagrange-Newton algorithm, so we will only include its convergence graph here.

The convergence rate is much faster for larger \(x\), and the Lagrange matrix is of much better form.

Now we also set up another constrained problem to test our algorithms, the problem has 2 sets of constraints instead of one. It is : \(f(x)=-x_1^2+x_2^2\), \(C_1=1- x_1^2-x_2^2=0\),
\( C_2 = x_1^2 - 1/2 = 0 \). This is an interesting problem and there are only four feasible points (because of the additional condition \( C_2 \), \( x_1 \) can only be either \( +1/2 \) or \(-1/2\), and the same is also true for \( x_2 \)), these four feasible points are all optimal, it is easy to see that the minimum \( f(x) \) is 0. The augmented cost function (with \( \lambda_i \) and \( \sigma_i \)) of this problem is very complicated and is no longer even similar to quadratic functions, this makes all our three unconstrained sub-algorithm performed extremely poor, as they were developed for quadratic cost functions. Since Courant’s penalty algorithm and the Multiplier penalty algorithm employs the unconstrained sub-algorithms, their performance also suffered. In fact we were unable to see the results because the unconstrained algorithms either keep on restarting or run without returning. We were only able to get the result for the Lagrange-Newton algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( f(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lagrange-Newton</td>
<td>0.7071</td>
<td>0.7071</td>
<td>-2.2204e-016</td>
</tr>
<tr>
<td>starting point ([0;0])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lagrange-Newton</td>
<td>0.7071</td>
<td>0.7071</td>
<td>1.4433e-015</td>
</tr>
<tr>
<td>starting point ([1;2])</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The result is indeed very close to the real optimal point (if not exactly the same due to floating point inaccuracies). Again the Lagrange matrix becomes ill conditioned when the \( x \) is small, the following are the convergence graph for the Lagrange-Newton algorithm, the first one starts off from \([0;0]\), the second one starts off from \([1;2]\), note the shapes are very different, and it takes longer to converge when starts off from initial point \([0;0]\).
This problem actually shows us the limitation of optimization algorithms. Although it is a very simple problem that can be easily solved by very simple hand calculations (or even by inspections), yet it turns out to have an extremely complicated augmented cost function for the Courant and Multiplier penalty methods (after augmenting it with $\lambda_i$ and/or $\sigma_i$), as the augmented cost function will then contain many complicated high order terms. This somehow illustrates the fact that although optimization algorithms are powerful tools, yet sometimes human intervention are necessary because human intuitions are even more powerful, it will be better if we can try to figure out whether the problem can be solved in an easier way first(by hand) than to blindly feed it to the optimization tools. It will also help a lot if we can do some further work of simplifying the problem because that will make the augmented cost function simpler. In this case the two constraints $C_1=1- x_1^2-x_2^2=0$ and $C_2=x_1^2-1/2=0$ can in fact be simplified to $C_{1\text{new}}=1/2-x_2^2=0$, $C_{2\text{new}}=x_1^2-1/2=0$, as this does not change the feasibility set, but with these two sets of simpler constraints the cost function will also be simpler and can be solved more efficiently.

**Discussions**

There are a couple of numerical difficulties we have been facing in doing this project. The first difficulty is that since we use finite difference method to approximate the gradients, it can be hard to choose a good $\Delta x$ value, the gradient found becomes inaccurate either when $\Delta x$ is being too small or too large, we spent a while to fine tune
our $\Delta x$ so as to make the gradient calculation more precise, but still there are some functions whose gradient can not be calculated accurately or can not be calculated at all, some examples of these functions are the exponential function and the cost function $f(x) = x_1^2(x_2^2)$ we have been using (but we managed to calculate the gradient for it, on small $x$ values). The second difficulty concerns the step size selection, since we can not use exact step size along the searching direction, we need to use the Armijo’s rule, which can be inaccurate. With inaccurate step size, the search algorithms may work differently than our anticipation, for example even in case of quadratic equations the solution might not converge in $n$ steps for conjugate gradient and secant algorithms. The third difficulty concerns with the Lagrange matrix used in the Lagrange-Newton’s algorithm, since we need to inverse the Lagrange matrix in order to find $\lambda$ and $x$, we will not be able to solve the problem whenever the Lagrange matrix is singular. This has limited the number of problems the Lagrange-Newton algorithm can solve, but we believe it might be possible to make modification to the scheme so that we may approximate the singular Lagrange matrix (for example: replace all the zero eigenvalues with a small non-zero value) and had the otherwise unsolvable problems be solved.
Matlab Codes: (the codes are listed in alphabetical order of their names, each function (module) starts with its name as the first comment)

%armijo.m
%implement the armijo algorithm to search for a suitable step size
function [val]=armijo(fx,x,s) % s is the search direction,
                          % the returning value val is the step size
[m n]=size(x);

r=1.1; %r must be larger than 1
u=0.9; %u must be smaller than 1

v1=0; %this to store the value of f(x)
v2=1; %this is to store the value of the linear function
p=0;
q=0;

%finding a w on the right of wj
while v1<v2

    y=x+r^p*s; %finding the step size
    v1=evalu(fx,y); %evaluting f(x) at y
    v2=evalu(fx,x)+0.5*r^p*[(gradient(fx,x))'*s]; %evaluating the linear function

    if v1<v2 %p indeed still needs to be increased
        p=p+1;
    end

end

%finding a w on the left of wj, near wj
while v1>v2

    y=x+r^p*u^q*s; %finding the step size
    v1=evalu(fx,y); %f(x) at y

v2=evalu(fx,x)+0.5*r^p*u^q*[(gradient(fx,x))'*s];%the linear

function at y

if v1>v2 % q still needs to be increased
    q=q+1;
end

end

val=r^p*u^q; %step size found, return it
%augment_fx.m
%assume x is always column vector
%this is for question 5, we use it to augment the fx to include
%the augmented lagrangian
function [a_fx]=augment_fx(fx,cx,cn,lambd,sigma)

a_fx=[fx]; %the original f(x)

%attach the sigma part
for i=1:cn
    a_fx=[a_fx '+0.5*'];
    a_fx=[a_fx int2str(sigma(i))];
    a_fx=[a_fx '*('];
    a_fx=[a_fx cx{i}];
    a_fx=[a_fx ')'];
    a_fx=[a_fx '*('];
    a_fx=[a_fx cx{i}];
    a_fx=[a_fx ')'];
end

%attach the lambda part
for i=1:cn
    a_fx=[a_fx int2str(-lambd(i))];
    a_fx=[a_fx '*('];
    a_fx=[a_fx cx{i}];
a_fx=[a_fx ')'];
end
global fx_convergence_cg; %store the f(x(j)) here to see convergence
e=0.0001; %stopping criteria
x=cg_iteration(fx,x,e); %pass the f(x),initial iteration point x, %and the stopping criteria to cg_iteration
eval(fx,x) %evaluate f(x) at the returned x
[m n]=size(fx_convergence_cg);
l=1:n; plot(l,fx_convergence_cg); %plot the convergence graph
global cg_4.m %this is to be called by functions in Q4
%the only difference btw this function and cg.m is %that it does not plot the convergence graph.
%assume x is always column vector
function [x]=cg_4(fx,x)
%global fx_convergence; %store the f(x(j)) here to see convergence
e=0.01; %stopping criteria
x=cg_iteration(fx,x,e);
%cg_beta.m
%function finding the beta(j+1) for the CG algorithm
function [beta]=cg_beta(fx,x,oldx)
g_old=gradient(fx,oldx); %gradient of V at x(j)
g_new=gradient(fx,x); %gradient of V at x(j+1)
beta=(g_new-g_old)'*g_new/(norm(g_old))^2 %beta(j+1)
cg_iteration.m
function [x]=cg_iteration(fx,x,e)
[m n]=size(x);
global fx_convergence_cg;
theta=(pi-0.1)/2; %the acceptable search direction (i.e cone direction)

i=1;

%whenever 1) direction not within the cone, gradient!=0 even after n iterations, 
%we will restart it at x(j) by setting i=0
while i<m

    if norm(gradient(fx,x))<=e %optimal point reached 
        break;
    end

    if i==1 
        s=-gradient(fx,x);
    end

    oldx=x; %x(j) 
    w=armijo(fx,x,s); %w(j) 
    x=x+w*s; %x(j+1)

    g_old=gradient(fx,oldx); %gradient of V at x(j)
    g_new=gradient(fx,x); %gradient of V at x(j+1)
    beta=(g_new-g_old)'*g_new/(norm(g_old))^2; %update beta(j+1)

    s=-g_new+beta*g_old; %update s(j+1) 

    if in_cone(fx,x,s,theta)==0 
        i=0; %not in the acceptable search direction, need to restart
        %ie can not use the s(j+1), need to use 
        -gradient(fx,x(j+1))
    end

    %store the f(x(j)) to see convergence 
    [a b]=size(fx_convergence_cg);
k=b+1;
fx_convergence_cg(k)=eval(fx,x);

i=i+1;

if (i==m) % not acting like a quadratic, we need to restart
    if (norm(gradient(fx,x))>e)
        i=0;
    end
end

end
end

courant.m
%fx is the cost function
%ccx is c(x)'*c(x)
%i.e fx='x(1)^2-x(2)^2+5+(1/2)*sigma*ccx'
%cx=[x(1)^2+x(2)^2-1=0;x(1)-x(2)=0]
%=> cx='[x(1)^2+x(2)^2-1;x(1)-x(2)]'
%=> ccx='((x(1)^2+x(2)^2-1)^2+(x(1)-x(2))^2)' %in our example fx='-(x(1)-x(2)+(1/2)*sigma*ccx'
% ccx='(1-x(1)^2-x(2)^2)^2'
%remember ccx must be enclose by a pair of brackets
function [x]=courant(fx,ccx,x)
e=0.01; %stopping criteria, when norm(cx,x) is smaller than e, we can stop
    %since c(x(sigma(k))) is small enough
sigma_int=10; %initial sigma value is 10
sigma=int2str(sigma_int); %need to make it as a string so that can be pass as
        %as a string with the augmented cost function
fx=strrep(fx,'ccx',ccx); %add the penalty function to the f(x)
k=1; %iteration number
while sqrt(eval(ccx,x))>e %norm of cx is sqrt(c'c)=sqrt(ccx)
x=secant_4(strrep(fx,'sigma',sigma),x); %pass the augmented cost function to secant_4.m
sigma_int=sigma_int*2; %increase the sigma value so that it -> infinite
sigma=int2str(sigma_int); %make it a string so that can be passed easily with the augmented f(x).

%evaluate the f(x(k)) to see convergence
temp1=strrep(fx,'sigma','0');
temp2=strrep(temp1,'cxx','0');
fx_convergence_cour(k)=eval(temp2,x);

k=k+1; %update iteration number
end

fx=strrep(fx,'sigma','0'); %de-augment the cost function by multiplying a zero to the square of constraints
eval(fx,x) %evaluate the f(x) at the optimal point
[m n]=size(fx_convergence_cour);
l=1:n;plot(l,fx_convergence_cour);

%c_hessian.m
%finding the hessian for the constraint functions %returns a SET
function [c_hess]=c_hessian(cx,x,cn)

[m n]=size(x);

%for general problem, need to define
for i=1:cn
c_hessian{i}=zeros(m,m);

end

%for our problem c_hess is

c_hess{1}=[-2 0; 0 -2];

c_hess{2}=[2 0; 0 0];

%finding the hessian for the constraint functions
%returns a SET
function [c_hess]=c_hessian(cx,x,cn)

[m n]=size(x);

%for general problem, need to define
for i=1:cn

    c_hessian{i}=zeros(m,m);

end

%for our problem c_hess is

c_hess{1}=[-2 0; 0 -2];
c_hess{2}=[2 0; 0 0];

%finding the updated H(j+1) by the DFP algorithm
function [dfp_val]=dfp(h,deta_x,deta_g)

dfp_val=(deta_x*(deta_x)')/((deta_x)'*deta_g) - (h*deta_g*(h*deta_g)')/((deta_g)'*h*deta_g);

%evaluating the f(x) at a given vector x
function [val]=evalu(fx,x)
val=eval(fx);
%find_a.m
%find the A matrix for Q5 and Q6, A is in fact the Jacobian matrix
%x holds the vector x, c is a set and holds all the c_i(x),cn tells how many
%ci(x) there are.
function [A]=find_a(cx,x,cn)

for i=1:cn
    A(:,i)=gradient(cx{i},x);
end

%find_w.m
%find w(X) for Q6
%cn is the constraint number
%definition of w at pg22-2
function [w]=find_w(fx,cx,x,cn,lambd)

[m n]=size(x);

f_hess=f_hessian(fx,x); %find the hessian of f(x)

if_hess=c_hessian(cx,x,cn); % returns hessian for all the c_i(x) in the set c_hess

stuff=zeros(m,m);
for i=1:cn
    stuff=stuff+(lambd(i))*eval(c_hess{i},x); %add the lambd part of w
end

w=f_hess-stuff; %return the value of w

%find_w.m
%find w(X) for Q5 and Q6
%cn is the constraint number
%definition of w at page 21-9
function [w]=find_w(fx,cx,x,cn,lambd,sigma)

[m n]=size(x);

f_hess=f_hessian(fx,x);
c_hess = c_hessian(cx, x, cn); % returns hessian for all the c_i(x) in the set c_hess

stuff = zeros(m, m);
for i = 1:cn
    stuff = stuff + (lambd(i) - sigma(i) * eval(cx{i}, x)) * eval(c_hess{i}, x);
end

w = f_hess - stuff;

function v = f(x)

v = x(1)^2 + 2*x(1)*x(2) + 3*x(1) + 5;

%f_hessian.m for the suggested problem
function [f_hess] = f_hessian(fx, x)

f_hess = [0 0; 0 0];
%f_hessian.m for the suggested problem
function [f_hess] = f_hessian(fx, x)

f_hess = [-2 0; 0 2];

%gradient.m
%assume x is a column vector!
%always return a column vector as the answer
function [val] = gradient(fx, x)

h = 0.00001; % h is a small value for calculating the gradient

[m n] = size(x);

% implement finite difference method for the gradient as state on 5.14
%i below is the ith element of a vector x
for i = 1:m
    oldval = eval(fx); % value of f(x)
temp=x(i); x(i)=x(i)+h;
newval=eval(fx); \% value of f(x+h)
x(i)=temp;
val(i)=(newval-oldval)/h; \% gradient = \( \frac{f(x+h)-f(x)}{h} \), h->0
end
val=val'; \% return the gradient as a column vector

%in_cone.m
% test to see if the search direction is still in the cone
% returns 1 if it is in the cone
% returns 0 otherwise
function [in]=in_cone(fx,x,s,theta)

\% check the direction
if s'*[-gradient(fx,x)]>norm(s)*norm(gradient(fx,x))*cos(theta)
in=1; \% in the cone
else
in=0; \% not in the cone
end

%lagrange.m
% lagrange-newton method
% assume X is always a column vector
% assume the lagrange matrix is non-singular

function [x]=lagrange(fx,cx,x,cn)
e=0.01; \% stopping criteria
[m n]=size(x);
infinite=99999; \% definite infinite to be a very large number

\% initializations
empty=zeros(cn,cn); \% this is the zero matrix at the lower right part of the lagrange matrix
for i=1:cn
    lambd(i)=-1000; \% set initial lambd
end
lambd=lambd'; \% make it a column matrix
deta_x=zeros(n,1); \% set initial deta_x
deta_lambd=zeros(cn,1); \hspace{1em} \% set initial deta_lambd

k=0; \hspace{1em} \% iteration number

\textbf{while} k<\text{infinite} \hspace{1em} \% endless loop until we ask it to stop or iteration reach
the number defined by infinite

\hspace{1em} \% definition of A can be found on pg22-2 and 21-8, we follow that
\hspace{1em} \% of
\hspace{1em} \% A is the jacobian matrix
A=find_A(cx,x,cn); \hspace{1em} \% finding A
w=find_w(fx,cx,x,cn,lambd); \hspace{1em} \% finding w
\hspace{1em} \% g_fx=gradient(fx,x); \hspace{1em} \% finding gradient of \( f(x) \)
\hspace{1em} \% for \( i=1:cn \) \hspace{1em} \% check to see the constraints values at a \( x \) value from
the previous iteration
\hspace{1em} c(i)=eval(cx{i},x); \hspace{1em} \%
end
\hspace{1em} if \( k==0 \)
\hspace{1em} \hspace{1em} \hspace{1em} c=c'; \hspace{1em} \% make it a column vector at the 1st iteration
end
\hspace{1em} if \hspace{1em} \text{norm}(c)<e
\hspace{1em} \hspace{1em} break; \hspace{1em} \% break the while loop, because stopping criteria
reached
\hspace{1em} end

\hspace{1em} \% evaluate the \( f(x(k)) \) to see convergence
fx_convergence_lag(k+1)=evalu(fx,x);
\hspace{1em} \% evaluate the \( f(x(k)) \) to see convergence

\hspace{1em} temp=inv([w -A; -A' empty])\hspace{1em} *\hspace{1em} [-g_fx;c]; \hspace{1em} \% using direct inversion of
lagrange matrix
\hspace{1em} deta_x=temp(1:m,1); \hspace{1em} \% m is dimension of \( x \) vector,actual is \( n \) in
our notes
\hspace{1em} lambd=temp(m+1:m+cn,1); \hspace{1em} \% lambd(k+1)
x=x+deta_x; \hspace{1em} \% x(k+1)
k=k+1; \hspace{1em} \% increase the iteration number
eval(fx,x)
[m n]=size(fx_convergence_lag);
l=1:n;plot(l,fx_convergence_lag);title('Plot of the convergence rate for Lagrange-Newton method');xlabel('Iteration');ylabel('V(x)');

%mp.m
%multiplier-penalty method
%assume X is always a column vector
%c is the constraint set
%cn is the number of constraints
%fx is the cost function
%i.e. fx='\text{-}x(1)\text{-}x(2)\text{-lambd}_c\text{+0.5*csc}';
% cn=2, c{1}=\text{'1-x(1)^2-x(2)^2'}, where c(1)=0, since it is an ECP
% c{2}=\text{'x(1)-x(2)'} , where c(2)=0, since it is an ECP
%let's use the given example
% fx='\text{-}x(1)\text{-}x(2)'; cx{1}=\text{'1-x(1)^2-x(2)^2'};

function [x]=mp(fx,cx,x,cn)

global fx_convergence_mp; %store the f(x(k)) to see convergence

[m n]=size(x); %get the dimension of vector x
infinite=99999; %define infinite
e=0.01;%stopping criteria

%a_fx=[fx '-lambd_c+0.5*csc']; %augmented fx to be solved

%fill the first lambd+pre-fill sigma, will fill the sigma again below
for i=1:cn
    lambd(i)=5;
    sigma(i)=1;
end
end

lambd=lambd';  % make it a column matrix
sigma=sigma';  % make it a column matrix

k=1;

while k<infinite
  if k==1

    % set the c to infinite
    for i=1:cn
      c_old(i)=infinite;  % setting all constraint functions to be infinite
    end

    % fill the first sigma, 10 times the original value, i.e.
    sigma=10 for the very first iteration, sigma=old_sigma*10 in the case of
    re-starting
    for i=1:cn
      sigma(i)=sigma(i)*10;
    end

    S=diag(sigma);
  end

  a_fx=augment_fx(fx,cx,cn,lambd,sigma);  % augment the fx to be
  theta(x,lambd,sigma)
  x=secant_4(a_fx,x);  % use secant_4 because it does not plot the
  graph and produce less noise, can
    % actually use secant, except it produce more
  noise

  % evaluate the constraints
  for i=1:cn
    c(i)=eval(cx{i},x);
  end
  c=c';  % make it a column vector
eval(fx,x); %evaluate f(x) at the new x

%see if the infinite norm of the constraint is small enough
if (norm(c,inf)<=0.25*norm(c_old,inf))
    
    k=0;%need to enlarge sigma and restart

else
    %finding the updating equation for lambd and updates lamb
    A=find_A(cx,x,cn);
    w=find_w_sigma(fx,cx,x,cn,lambd,sigma);
    w=w+A*S*A';

    lambd=lambd-inv(A'*w*inv(A))*c;
end

c_old=c; %store the c(k-1) for testing purpose in the next iteration

%evaluate the f(x(k)) to see convergence
fx_convergence_mp(k)=evalu(fx,x);
%evaluate the f(x(k)) to see convergence

k=k+1; %increase iteration number

%check to see if stopping criteria reached
if (norm(c))<e
    k=infinite+1;%stopping criteria reached
end

end
%secant.m
% a wrapper function for the secant_iteration.m function
function [x]=secant(fx,x)
global fx_convergence_sec; %store data to plot the convergence picture
e=0.0001; %stopping criteria

[x]=secant_iteration(fx,x,e); %perform secant iterations

eval(fx,x) %evaluate the final f(x)

[m n]=size(fx_convergence_sec);
l=1:n;plot(l,fx_convergence_sec); %plotting the convergence graph
%secant_4.m
% this is for Q4 only,
% the only difference between this and
% secant.m is secant_4.m does not plot the convergence graph
function [x]=secant(fx,x)
e=0.0001; %stopping criteria

[x]=secant_iteration(fx,x,e);

%secant_iteration.m
function [x]=secant_iteration(fx,x,e)
global fx_convergence_sec;
theta=(pi-0.1)/2; %the acceptable search direction within the cone

[m n]=size(x);
i=0;

while i<m+1

if i==0
\begin{verbatim}
    h=eye(m);
    s=-h*gradient(fx,x);

oldx=x; %x(j)
w=armijo(fx,x,s); %w(j)
x=x+w*s; %x(j+1);

    if norm(gradient(fx,x))<=e %optimal point reached
        break;
    end

deta_x=x-oldx;
deta_g=gradient(fx,x)-gradient(fx,oldx);

    %implement Davidon-Fletcher-Powell Algorithm to update the h matrix
    dfp_val=dfp(h,deta_x,deta_g);
    h=h+dfp_val;

    s=-h*gradient(fx,x); %cal s(j+1)=h(j+1)*gradient(fx,x(j+1)) for the next iteration to use

    if in_cone(fx,x,s,theta)==0
        i=0; %not in the cone, need to restart
    end

    %store the fx values to see convergence
    [a b]=size(fx_convergence_sec);
k=b+1;
fx_convergence_sec(k)=eval(fx,x);

    i=i+1;
\end{verbatim}
if i==m+1
    if norm(gradient(fx,x))>e
        i=0; % gradient(fx(m+1),x(m+1)) not equal to zero, need to restart
    end
end
end

%steepest descent
%need to pass the function fx as string and initial condition of x
%assume x is always column vector
%i.e fx='x(1)^2+x(2)^2';x=[1;2];
function [x]=stp_descent(fx,x)

global fx_convergence_sd;

e=0.0001; %stopping condition

k=1; %iteration number

%check to see if stopping criteria reached
while norm(gradient(fx,x))>e

    s=-gradient(fx,x); %find the searching direction
    w=armijo(fx,x,s); %find the step size by armijo algorithm
    x=x+w*s; %update the x

    fx_convergence_sd(k)=eval(fx,x); %store f(x(k)) to see convergence

    k=k+1;
end

optimal_fx=evalu(fx,x) %evaluate the f(x) at the returned x
[m n]=size(fx_convergence_sd);
l=1:n;plot(l,fx_convergence_sd);