The University of British Columbia

CPSC 406

Computational Optimization

Homework #2

Student name:

ALI SIAHKOOHI

Student ID:

84264167

Instructor:

Professor Ascher

Date Submitted:

October 6th, 2016

Question #1

Part (a)

In order to find the critical points, we need to compute the gradient of the given function first. Then by imposing the gradient to be equal to zero we can find the critical points

$$f(x)=\frac{1}{2}x_1{}^2-\frac{1}{4}x_2{}^4+\frac{1}{327},$$

$$\nabla f(x)=\begin{pmatrix}x_1\\ -x_2{}^3\end{pmatrix}=0\Rightarrow\begin{pmatrix}x_1\\ x_2\end{pmatrix}=0.$$

$$\nabla^2 f(x)=\begin{pmatrix}1 & 0\\ 0 & -3x_2{}^2\end{pmatrix}\Rightarrow\begin{cases}\lambda_1>0\\ \lambda_2\le 0\end{cases} \qquad (1-1)$$

So this function only has one critical point and that's a saddle point. Because one of the eigen-values of hessian matrix is always negative and the other one is non-negative.

Part (b)

We are going to compute the direction function analytically.

$$\nabla^2 f(x)=\begin{pmatrix}1 & 0\\ 0 & -3x_2{}^2\end{pmatrix}\Rightarrow\begin{cases}\lambda_1>0\\ \lambda_2\le 0\end{cases}.$$

$$H(x_k)p_k=-g(x_k),$$

$$x_{k+1}=x_k+p_k.$$

$$x_k=\begin{pmatrix}x_1\\ x_2\end{pmatrix}\rightarrow H(x_k)=\begin{pmatrix}1 & 0\\ 0 & -3x_2{}^2\end{pmatrix},\quad g(x_k)=\begin{pmatrix}x_1\\ -x_2{}^3\end{pmatrix},$$

$$\Rightarrow p_k=\begin{pmatrix}1 & 0\\ 0 & \dfrac{-1}{3x_2{}^2}\end{pmatrix}\begin{pmatrix}-x_1\\ x_2{}^3\end{pmatrix}=\begin{pmatrix}-x_1\\ \dfrac{-x_2}{3}\end{pmatrix},$$

$$\Rightarrow x_{k+1}=x_k+\begin{pmatrix}-x_1\\ \dfrac{-x_2}{3}\end{pmatrix}=\begin{pmatrix}x_1\\ x_2\end{pmatrix}+\begin{pmatrix}-x_1\\ \dfrac{-x_2}{3}\end{pmatrix}=\begin{pmatrix}0\\ \dfrac{2x_2}{3}\end{pmatrix}$$

$$\Rightarrow x_{k+1}=x_k-\frac{1}{3}x_k=\frac{2}{3}x_k \qquad (1-2)$$

As it can be seen, in the first iteration, independent of value of the first element of x, the first element is going to be zero. But the second element of x is converging to zero in a linear rate. So the Newton's method converges to the critical point, which is a saddle point, linearly.

Question #2

I have implemented the Newton and BFGS method, plus backtracking line search and quadratic interpolation scheme for line search. I am going to report the results of quadratic interpolation line search in the following. In the following tables, x_1 means first entry of vector x, and x_2 is the second one.

1) Newton's method without line search:

Iteration_Number	x_1	x_2	step_size	function_value
0	0	0	1	1
1	1	0	1	100
2	1	1	1	4.9304e-30
3	1	1	1	0

As we can see the newton's method has converged pretty fast!

2) Newton's method with quadratic interpolation line search:

Iteration_Number	x_1	x_2	step_size	function_value
0	0	0	1	1
1	0.25	0	0.25	0.95312
2	0.30556	0.090278	1	0.48321
3	0.52025	0.22302	0.5	0.45709
4	0.56582	0.31808	1	0.18894
5	0.60764	0.36569	0.13634	0.1552
6	0.67082	0.44344	0.27509	0.11266
7	0.81321	0.64103	1	0.076
8	0.85016	0.72141	1	0.022638
9	0.96786	0.9229	1	0.020224
10	0.97638	0.95325	1	0.00055827
11	0.99966	0.99878	1	2.9478e-05
12	0.99997	0.99993	1	1.0949e-09
13	1	1	1	1.2006e-16
14	1	1	1	1.2326e-32

As we expected, newton's method only converges quadratic when the step size is equal to one. Although, there is inly four iterations which step size is not one.

3) BFGS method without line search:

The results are in the next page.

Iteration_Number	x_1	x_2	step_size	function_value
0	0	0	1	1
1	2	0	1	1601
2	0.12594	0.49938	1	24.143
3	-24.35	-87.721	1	4.6326e+07
4	0.13886	-0.18743	1	5.0147
5	0.13334	0.10602	1	1.5297
6	0.135	0.018083	1	0.74822
7	0.13501	0.018066	1	0.74821
8	0.13708	0.015684	1	0.7456
9	0.14114	0.013878	1	0.7413
10	0.15457	0.012171	1	0.72849
11	0.18246	0.014347	1	0.70426
12	0.22676	0.025652	1	0.6643
13	0.28405	0.052416	1	0.59249
14	0.36391	0.11171	1	0.44754
15	0.4362	0.18395	1	0.32187
16	0.62706	0.3335	1	0.49562
17	0.50775	0.26133	1	0.24355
18	0.55086	0.30362	1	0.20173
19	1.0734	0.84111	1	9.6801
20	0.55959	0.31494	1	0.19428
21	0.56745	0.32438	1	0.18767
22	1.0312	0.86683	1	3.8599
23	0.57939	0.33926	1	0.17818
24	0.59033	0.35245	1	0.16941
25	1.787	1.7806	1	200.19
26	0.59145	0.35306	1	0.16797
27	0.59267	0.35402	1	0.16667
28	0.63643	0.39332	1	0.14592
29	0.63919	0.39915	1	0.13904
30	0.70999	0.50052	1	0.085375
31	0.82493	0.65925	1	0.075841
32	0.78636	0.61708	1	0.045806
33	0.82217	0.67409	1	0.031974
34	0.95181	0.8841	1	0.050053
35	0.8683	0.75477	1	0.017415
36	0.89309	0.79731	1	0.011438
37	0.99612	0.97723	1	0.022602
38	0.9233	0.85233	1	0.0058855
39	0.94026	0.88333	1	0.0036276
40	0.98946	0.97449	1	0.0021787
41	0.97261	0.9447	1	0.00091238
42	0.97981	0.95917	1	0.00048233
43	0.99591	0.99146	1	3.0431e-05
44	0.99902	0.99798	1	1.3164e-06
45	0.99997	0.99993	1	5.2506e-09
46	1	0.99999	1	5.3615e-11
47	1	1	1	4.9145e-13
48	1	1	1	7.281e-17
49	1	1	1	2.038e-20

This method has struggled so much till it converges. As it can be seen, when we get closer to the optimum point, convergence rate starts to increase. Just as we expect it has a super-linear convergence.

4) BFGS with line search

Iteration_Number	x_1	x_2	step_size	function_value
0	0	0	1	1
1	0.5	0	0.25	6.5
2	0.57413	0.10081	0.051412	5.4173
3	0.11338	0.42789	0.18241	18.012
4	-0.26972	-0.071723	1	3.6993
5	0.1566	0.12875	1	1.7976
6	0.26945	0.092669	1	0.57397
7	0.31368	0.091993	0.38733	0.47514
8	0.33253	0.098274	1	0.46065
9	0.37351	0.12023	1	0.42967
10	0.43029	0.16132	1	0.38136
11	0.49416	0.22462	1	0.2942
12	0.55953	0.30938	1	0.19538
13	0.61526	0.36758	0.34121	0.16005
14	0.71954	0.49596	1	0.12609
15	0.72444	0.52115	1	0.077275
16	0.79068	0.62056	1	0.045938
17	0.88885	0.77427	1	0.037242
18	0.86848	0.75424	1	0.017298
19	0.90183	0.81227	1	0.009742
20	0.94137	0.88259	0.46519	0.0047196
21	0.96638	0.93104	1	0.0019397
22	0.98069	0.96221	1	0.00039359
23	0.99983	0.99885	1	6.4407e-05
24	0.99777	0.99546	1	5.6408e-06
25	0.99947	0.99893	1	2.937e-07
26	0.99999	0.99998	1	1.5821e-10
27	1	1	1	1.0605e-14
28	1	1	1	3.9146e-22

We reach to the optimum point much faster using line search. There has been six iterations which the method has used a smaller step size that one.

Following are the codes used in this question. There are four functions, named, mynewton.m, myBFGS.m, bkt.m, and wlsearch.m, which are newton's method, BFGS method, backtracking line search, and quadratic interpolation line search respectively. I have also included at the very beginning the main script which buy running it results for question can be obtained.

```
clear; clc; close all;
% HW#2 main script
% Defining the function, its gradient, and hessian.
f = @(x) 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
fp = @(x) [100*2*(-2*x(1))*(x(2)-x(1)^2) - 2*(1 - x(1)); 100*2*(x(2) -
x(1)^2)];
fpp = @(x) [-400*(x(2) - x(1)^2) + 800*x(1)^2 + 2, -400*x(1); ...
    -400*x(1) 200];


% allocating the starting point, maximum iterations, and absolute
% tolerance.
x0 = [0; 0];
```

```
atol = 1e-7;
mitr = 2000;
C0 = eye(2, 2);

%%
% i = 0 means usning quadratic interpolation for line search, i = 1 means
% using backtracking and else means whithout line search

i = 1;

[x_n, xtmp_n, cp_n, count_n, alpha_n] = mynewton(f, fp, fpp, x0, atol, mitr,
i);

[x_BFGS, xtmp_BFGS, cp_BFGS, count_BFGS, alpha_BFGS] = myBFGS(f, fp, C0, x0,
atol, mitr, i);


Newton_method = table;
Newton_method.Iteration_Number = (0:count_n)';
Newton_method.x_1 = xtmp_n(1, :)';
Newton_method.x_2 = xtmp_n(2, :)';
Newton_method.step_size = alpha_n';
for i = 1:count_n+1
common(i, 1) = f(xtmp_n(:, i));
end
Newton_method.function_value = common;
Newton_method


BFGS_method = table;
BFGS_method.Iteration_Number = (0:count_BFGS)';
BFGS_method.x_1 = xtmp_BFGS(1, :)';
BFGS_method.x_2 = xtmp_BFGS(2, :)';
BFGS_method.step_size = alpha_BFGS';
for i = 1:count_BFGS+1
common(i, 1) = f(xtmp_BFGS(:, i));
end
BFGS_method.function_value = common;
BFGS_method
```

```
function [x, xtmp, cp, count, alpha_tmp] = mynewton(f, fp, fpp, x0, atol,
mitr, set)

x = x0;
cp = 1;
count = 0;
xtmp(:, 1) = x;
alpha = 1;
alpha_tmp(1) = alpha;
%%
while (cp>0 && count<mitr)
```

```
    % search direction
    p = - fpp(x)^-1*fp(x);

    % choosing whether use line search or not, and if yes, backtracking or
quadratic interpolation
    if set == 0;
        alpha = wlsearch(f, fp, p, x);
        xn = x + alpha*p;
    else if set == 1
        alpha = bkt(f, fp, p, x);
        xn = x + alpha*p;
        else
            xn = x + p;
        end
    end

    % computing the stopping criterion
    cp = norm(xn - x)-atol*(1+norm(x));

    x = xn;

    % allocating the outputs
    count = count + 1;
    xtmp(:, count + 1) = x;
    alpha_tmp(:, count + 1) = alpha;
end


end
```

```
function [x, xtmp, cp, count, alpha_tmp] = myBFGS(f, fp, C, x0, atol, mitr,
set)

x = x0;
alpha = 1;

cp = 1;
count = 0;
xtmp(:, 1) = x0;

alpha_tmp(1) = alpha;
%%
while (cp>0 && count<mitr)

    % computing the search direction
    p = -C*fp(x);

     % choosing whether use line search or not, and if yes, backtracking or
quadratic interpolation
    if set == 0
        alpha = wlsearch(f, fp, p, x);
```

```
        xn = x + alpha*p;
    else if set == 1
            alpha = bkt(f, fp, p, x);
            xn = x + alpha*p;
        else
            xn = x + alpha*p;
        end
    end


    w = alpha*p;
    y = fp(xn) - fp(x);
    rho = (y'*w)^-1;
    C = (eye(size(C))-rho*(w*y'))*C*(eye(size(C))-rho*(y*w')) + ...
        rho*(w*w');

    % allocating the outputs
    count = count + 1;
    xtmp(:, count + 1) = xn;
    cp = norm(xn - x) - atol*(1 + norm(x));
    alpha_tmp(:, count +1) = alpha;
    x = xn;
end

end
```

```
function alpha = bkt(f, fp, p, x)

alpha = 1;
rho = .5;
c = 1e-4;
xn = x + alpha*p;

while (f(xn)>(f(x)+c*alpha*fp(x)'*p) && alpha>.4)
    alpha = rho*alpha;
    xn = xn + alpha*p;
end
```

```
function alpha = wlsearch(f, fp, p, x)

alpha = 1;
sigma = 1e-4;
alphamin = 0.4;
xn = x + alpha*p;
fx = f(x);
fpx = fp(x)'*p;
fxn = f(xn);
```

```
while (fxn > (fx + sigma*alpha*fpx)) && (alpha>alphamin)

    mu = -.5*fpx*alpha^2/(fxn - fx - alpha*fpx);

    if mu < .1

        mu = .5;

    end

    alpha = mu*alpha;
    xn = x + alpha*p;
    fxn = f(xn);

end
```

Question #3

Part (a)

We assume that:

$$B_{k+1} = B_k + \gamma v v^T \qquad (3-1)$$

Which $\gamma \in \{-1,1\}$. Regarding to BFGS algorithm, we have:

$$y_k = B_{k+1}s_k \Rightarrow y_k = \left(B_k + \gamma v v^T\right)s_k \Rightarrow B_k s_k + \gamma v v^T s_k = y_k,$$
$$\Rightarrow \gamma v(v^T s_k) = y_k - B_k s_k, \qquad (3-2)$$

Since $v^T s_k$ is a scalar, v must be a multiple of $y_k - B_k s_k$, that is, $v = \lambda(y_k - B_k s_k)$ for some scalar λ. Therefore:

$$y_k - B_k s_k = \gamma\lambda^2\left[(y_k - B_k s_k)^T s_k\right](y_k - B_k s_k) \qquad (3-3)$$

This equation satisfies if and only if:

$$\gamma = \text{sign}((y_k - B_k s_k)^T s_k), \quad \lambda = \pm\left|(y_k - B_k s_k)^T s_k\right|^{-0.5} \qquad (3-4)$$

So there is only one symmetric rank-one update that satisfies the secant equation.

Part (b)

First of all, we show that the curvature condition is necessary to get a PD matrix B!

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k},$$

$$0 \le v^T B_{k+1} v = v^T B_k v + \frac{v^T (y_k - B_k s_k)\left(v^T (y_k - B_k s_k)\right)^T}{(y_k - B_k s_k)^T s_k}$$

$$v^T B_k v \ge 0, \quad v^T (y_k - B_k s_k)\left(v^T (y_k - B_k s_k)\right)^T \ge 0$$

$$\Rightarrow (y_k - B_k s_k)^T s_k \ge 0 \Rightarrow s_k{}^T (y_k - B_k s_k) = s_k{}^T y_k - s_k{}^T B_k s_k \ge 0 \qquad (3-5)$$

If we can show that te curvature condition $s_k{}^T y_k$ could be negative sometimes, it means that B_{k+1} is not always PD. Following is an example for that:

I am going to bring a counter-example to show the updates are not always positive definite.

Khalfan HF, Byrd RH, Schnabel RB. A theoretical and experimental study of the symmetric rank-one update. SIAM Journal on Optimization. 1993 Feb;3(1):1-24.

Regarding to the paper above, chapter 4 page 16 (Positive definiteness of SR1 update), there are several functions (Appendix, table 8) which during the convergence, the matrix B gets indefinite.

One of the above mentioned functions is “Chained Wood Function” which is defined at the following reference:

Conn AR, Gould NI, Toint PL. Testing a class of methods for solving minimization problems with simple bounds on the variables. Mathematics of computation. 1988;50(182):399-430.

In the page 423, the Chained Wood Function is defined as below:

$$f(x) = 1 + \sum_{i \in J} \left[100(x_{i+1} - x_i{}^2)^2 + (1 - x_i)^2 + 90(x_{i+3} - x_{i+2}{}^2)^2 + (1 - x_{i+2})^2 + 10(x_{i+1} + x_{i+3} - 2)^2 + 0.1(x_{i+1} - x_{i+3})^2 \right]$$

$$J = \{1, 3, 5, \ldots, 4n - 3\}, \qquad (3-6)$$

$$x_0 = (-3, -1, -3, -1, -2, 0, -2, 0, \ldots, -2, 0)$$

In (Khalfan HF, Byrd RH, Schnabel RB) paper (the first paper mentioned) in the Table 8 in Appendix, it has been shown that during SR1 algorithm for the above function for n=2, the matrix B has been evaluated as indefinite several times. The binary string below shows whether the B matrix in each iteration (140 iterations) is PD(=1) or indefinite(=0).

11111111011011111011101111111011011101 11111001101111110110111001101111010011011000000000111101111110011101001111101100111010011011111010111111

I hope there is no simple proof and my answer gets credit! ☺

Part (c)

We can easily see that this equation holds:

$$B_{k+1} = B_k - \frac{B_k s_k \left(B_k s_k\right)^T}{s_k{}^T B_k s_k} + \frac{y_k y_k{}^T}{y_k^T s_k},$$

$$\Rightarrow \left(B_k - \frac{B_k s_k \left(B_k s_k\right)^T}{s_k{}^T B_k s_k} + \frac{y_k y_k{}^T}{y_k^T s_k} \right) s_k = B_k s_k - \frac{B_k s_k \left(\left(B_k s_k\right)^T s_k\right)}{\left(\left(B_k s_k\right)^T s_k\right)^T} + \frac{y_k \left(y_k{}^T s_k\right)}{y_k^T s_k},$$

$$\Rightarrow B_{k+1} s_k = B_k s_k - B_k s_k + y_k, \qquad (3-7)$$

$$\Rightarrow B_{k+1} s_k = y_k.$$

Question #4

Part (a)

1) Using fminsearch function and starting point $x_0 = \left[.75, -1.25\right]^T$:

Number of iterations: 23

Number of function evaluations: 45

2) Using fminunc without specifying the gradient and starting point $x_0 = \left[.75, -1.25\right]^T$:

Number of iterations: 5

Number of function evaluations: 21

3) Using fminunc with specifying the gradient and starting point $x_0 = \left[.75, -1.25\right]^T$:

Number of iterations: 4

Number of function evaluations: 5

4) Using fminsearch function and starting point $x_0 = [0, 0.3]^T$:

Number of iterations: 69

Number of function evaluations: 129

5) Using fminunc without specifying the gradient and starting point $x_0 = [0, 0.3]^T$:

Number of iterations: 8

Number of function evaluations: 36

6) Using fminunc with specifying the gradient and starting point $x_0 = [0, 0.3]^T$:

Number of iterations: 9

Number of function evaluations: 10

Part (b)

1) Using fminsearch function and starting point $x_0 = [8, 0.2]^T$:

Number of iterations: 54

Number of function evaluations: 102

2) Using fminunc without specifying the gradient and starting point $x_0 = [8, 0.2]^T$:

Number of iterations: 28

Number of function evaluations: 108

3) Using fminunc with specifying the gradient and starting point $x_0 = [8, 0.2]^T$:

Number of iterations: 19

Number of function evaluations: 20

4) Using fminsearch function and starting point $x_0 = [8, 0.8]^T$:

Number of iterations: 62

Number of function evaluations: 115

5) Using fminunc without specifying the gradient and starting point $x_0 = [8, 0.8]^T$:

Number of iterations: 28

Number of function evaluations: 105

6) Using fminunc with specifying the gradient and starting point $x_0 = [8, 0.8]^T$:

Number of iterations: 18

Number of function evaluations: 19

Several points can be extracted from the results.

1. `Nelder-Mead simplex direct search' needs more iteration to converge. Because it does not use the gradient information and therefore it needs a lot of function evaluation
2. When using Quasi-Newton, in other words, not specifying the gradient for fminunc, the function needs to evaluate it and therefore takes a lot of function evaluations to do so. Even more that 'Nelder-Mead simplex direct search'. On the other hand, since its using the gradient information, definitely take less iteration to converge comparing to Nelder-Mead simplex direct search'.
3. When specifying the gradient for fminunc function, in other words by trust-region method, the algorithm converges to the solution way faster than Nelder-Mead simplex direct search' but nearly as the same number of iterations for Quasi-Newton. On the other hand, since the gradient has been specified analytically, number of function evaluations is very low comparing to two other methods.
4. In the part (a), it is obvious that when the starting point is far away from the solution, it takes more iterations to converge.
5. In part (b), to starting points actually have the same Euclidean distance from the solution, and as a result, the number of iterations doesn’t differ much when using each of the starting points.

The following codes had been used in this question. There is two functions for each for part a or b, which the gradient of functions are in the second output argument and there is a main script which by running it results get obtained.

```
function [f,g] = myfunc1(x)

f = x(1).^4 + x(1).*x(2) + (1 + x(2)).^2;

if nargout > 1
    g = [4*x(1).^3 + x(2);
        x(1) + 2*(1 + x(2))];
end
```

```
function [f,g] = myfunc2(x)

f = .5*((1.5 - x(1).*(1 - x(2))).^2 + (2.25 - x(1).*(1 - x(2).^2)).^2 + ...
    (2.625 - x(1).*(1 - x(2).^3)).^2);

if nargout > 1
    g = [-(1 - x(2)).*(1.5 - x(1).*(1 - x(2))) - ...
        (1 - x(2).^2).*(2.25 - x(1).*(1 - x(2).^2)) - ...
        (1 - x(2).^3).*(2.625 - x(1).*(1 - x(2).^3));
        x(1).*(1.5 - x(1).*(1 - x(2))) + ...
        2*x(1).*x(2).*(2.25 - x(1).*(1 - x(2).^2)) + ...
        3*x(1).*x(2).^2.*(2.625 - x(1).*(1 - x(2).^3))];
end
```

```
clear; clc; close all;

f = @(x) myfunc1(x);
x0 = [.75, -1.25]';

[x1_1, fx1_1, flag1_1, output1_1] = fminsearch(f, x0);
[x2_1, fx2_1, flag2_1, output2_1] = fminunc(f, x0);
options = optimoptions('fminunc', 'GradObj', 'on');
[x3_1, fx3_1, flag3_1, output3_1] = fminunc(f, x0, options);


x0 = [0, 0.3]';

[x1_2, fx1_2, flag1_2, output1_2] = fminsearch(f, x0);
[x2_2, fx2_2, flag2_2, output2_2] = fminunc(f, x0);
options = optimoptions('fminunc', 'GradObj', 'on');
```

```
[x3_2, fx3_2, flag3_2, output3_2] = fminunc(f, x0, options);

%


f = @(x) myfunc2(x);
x0 = [8, .2]';

[x1_3, fx1_3, flag1_3, output1_3] = fminsearch(f, x0);
[x2_3, fx2_3, flag2_3, output2_3] = fminunc(f, x0);
options = optimoptions('fminunc', 'GradObj', 'on');
[x3_3, fx3_3, flag3_3, output3_3] = fminunc(f, x0, options);


x0 = [8, 0.8]';

[x1_4, fx1_4, flag1_4, output1_4] = fminsearch(f, x0);
[x2_4, fx2_4, flag2_4, output2_4] = fminunc(f, x0);
options = optimoptions('fminunc', 'GradObj', 'on');
[x3_4, fx3_4, flag3_4, output3_4] = fminunc(f, x0, options);
```

OUTPUT:

```
Question 4, Part (a). x0 = [.75, -1.25]

number of iterations for fminsearch = 23

number of function evaluations for fminsearch = 45

number of iterations for fminunc without gradient = 5

number of function evaluations for fminunc without gradient = 21

number of iterations for fminunc with gradient = 4

number of function evaluations for fminunc with gradient = 5


Question 4, Part (a). x0 = [0, 0.3]

number of iterations for fminsearch = 69

number of function evaluations for fminsearch = 129

number of iterations for fminunc without gradient = 8

number of function evaluations for fminunc without gradient = 36

number of iterations for fminunc with gradient = 9

number of function evaluations for fminunc with gradient = 10
```

```
Question 4, Part (b). x0 = [8, .2]
number of iterations for fminsearch = 54
number of function evaluations for fminsearch = 102
number of iterations for fminunc without gradient = 28
number of function evaluations for fminunc without gradient = 108
number of iterations for fminunc with gradient = 19
number of function evaluations for fminunc with gradient = 20

Question 4, Part (b). x0 = [8, 0.8]
number of iterations for fminsearch = 62
number of function evaluations for fminsearch = 115
number of iterations for fminunc without gradient = 28
number of function evaluations for fminunc without gradient = 105
number of iterations for fminunc with gradient = 18
number of function evaluations for fminunc with gradient = 19
```