**ISE 520**

# Programming Homework 3

**Aristotelis-Angelos Papadopoulos**
**USC ID: 3804-2945-23**

In this Homework assignment, we were asked to minimize the extended Rosenbrock function which is described by the following equation:

$$f(x) = \sum_{i=1}^{n/2} \left[ \alpha \left( x_{2i} - x_{2i-1}^2 \right)^2 + (1 - x_{2i-1})^2 \right]$$

The simulations were done for different values of $\alpha$ ($\alpha = 1$ and $\alpha = 100$), different values of $n = 2,4,6,10,20$ and we also considered the initial point to be $(-1, -1, \dots, -1)^T$ as it is suggested in the book.

The algorithms that we used in order to solve the above unconstrained optimization problem are: 1) Steepest Descent with Armijo rule, 2) Pure Newton, 3) Modified Newton with Armijo rule, 4) DFP with Armijo rule and 5) BFGS with Armijo rule. The reason for which we used Armijo rule for the step selection was in order to follow the suggestions that you gave us during the class and in order to be able to make a valid comparison between the different methods.

Before presenting the simulation results, it is necessary to say that all the algorithms were implemented using **C programming language**. Moreover, below, we make some comments regarding the way we implemented these different methods in our program.

1) Steepest Descent with Armijo rule

In general, for all the different methods that we used, we faced a problem with Armijo rule. The problem was the fact that sometimes, even though we still a had a negative directional derivative (all my programs print this value at each iteration in order to check that I have a descent direction), Armijo function was unable to satisfy both Armijo conditions and started dividing the step by $\eta = 2$.

Sometimes, it happened that Armijo function continuously divided the step and it could not provide a step which is sufficiently larger than zero in the sense that it can be described by a variable of type double. Thus, even a variable of type double (in C, such a variable can reach a precision of 14-15 decimal digits) was not able to describe the value of the step that Armijo function produced and the whole program was stucked.

To this end, in the Steepest Descent method, we always divided the direction vector with its norm in order to avoid as much as possible the extremely small steps. This modification was able to make the method move closer to the minimum before it is stucked. At this point, it is also worth noting that there are also different heuristics methods which we could use in order to avoid such a situation. For example, we could construct a condition that if the step produced by Armijo function is less than $\delta = 10^{-5}$, then take a step $10^{-5}$. However, you told us during the class to avoid using this type of heuristic methods.

2) Pure Newton

For this method, we considered $step = 1$. In order to be able to check if the Hessian matrix is positive definite, we factorized the Hessian at each iteration using the Modified Cholesky Factorization and thus we were able to write $H = LDL^T$ where $L$ is a lower triangular matrix and $D$ is a diagonal matrix. Then, using the suggestions of the book, we checked if all the elements of matrix $D$ were sufficiently positive. If it happened that they were not, then we took a Steepest Descent step with Armijo rule. In the case where they were sufficiently positive, we used the normal Newton direction but in order to avoid inverting the Hessian at each iteration, we applied the Gaussian elimination technique in the modified Cholesky factorization that we had taken from the previous step and we solved for the unknown direction.

At this point, it is worth noting that Pure Newton method was the only method at which we were able to observe a possible increase in the function value between two consecutive iterations even though the directional derivative had a strictly negative value. This behavior can be easily explained from the fact that a step of length 1 may be big enough and could possibly lead us to a higher function value. Despite this behavior, Pure Newton method was still able to converge very fast as it was expected. The only case where this method was stucked, was in the case where the Hessian was not positive definite and we were obligated to take a Steepest Descent step with Armijo rule and where Armijo function was not able to provide us a step which was sufficiently larger than zero as it was also described previously.

3) Modified Newton with Armijo rule

The only difference between this method and the Pure Newton method was the fact that when we had a positive definite Hessian and we took a Newton step, the selection of the step length was done using Armijo rule instead of taking a step of length 1.

4) DFP with Armijo rule

The illustration of this method was done using the update formula given in the equation (6.15) page 139 of the book of Nocedal & Wright.

It is worth noting that in the initialization of the Hessian inverse approximation, the identity matrix was used.

Again, this method is only stucked in the case where the Armijo function is not able to provide a sufficiently positive step.

5) BFGS with Armijo rule

The illustration of this method was done using the update formula given in the equation (6.17) page 140 of the book of Nocedal & Wright. The reason for which we used this update formula for the BFGS illustrations was in order to avoid the calculations needed for the modified Cholesky factorization.

It is worth noting that in the initialization of the Hessian inverse approximation, the identity matrix was used.

Again, this method is only stucked in the case where the Armijo function is not able to provide a sufficiently positive step. However, this method showed very promising results in general.

# Simulation Results

**Case 1 ($\alpha = 1$ and $n = 2$)**

a) Steepest Descent with Armijo rule

```
After 31 iter, we get:
x1,opt=1.00001
x2,opt=1.00002
f(xopt)=0.00000000
```

b) Pure Newton

```
After 2 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
f(xopt)=0.00000000
```

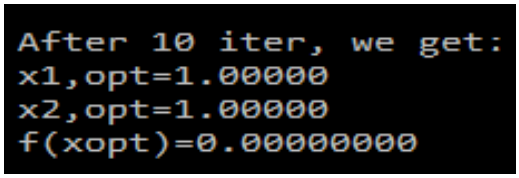
c) Modified Newton with Armijo rule

```
After 7 iter, we get:
x1,opt=1.00000
x2,opt=0.99999
f(xopt)=0.00000000
```

d) DFP with Armijo rule

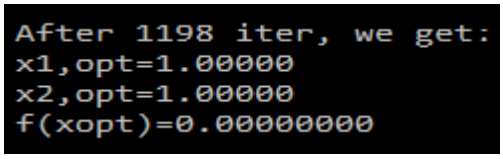
```
After 14 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
f(xopt)=0.00000000
```

e) BFGS with Armijo rule

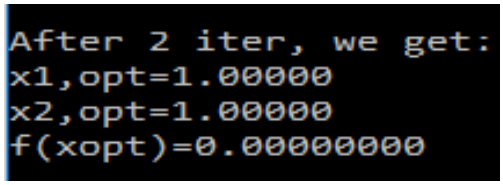



**Case 2 ($\alpha = 100$ and $n = 2$)**

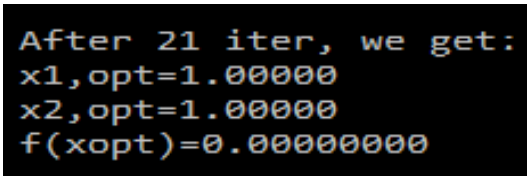
a) Steepest Descent with Armijo rule




b) Pure Newton




c) Modified Newton with Armijo rule



d) DFP with Armijo rule

```
After 84 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
f(xopt)=0.00000000
```

e) BFGS with Armijo rule

```
After 27 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
f(xopt)=0.00000000
```

**Case 3 ($\alpha = 1$ and $n = 4$)**

For these values of the parameters, some of the methods are stucked because of the Armijo function which is unable to provide a sufficiently positive step. Thus, the comparison between the methods will happen comparing how close the function value goes to the minimum value of zero and how many iterations it took the method to reach this value.

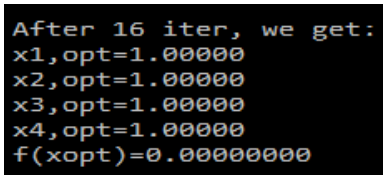
a) Steepest Descent with Armijo rule

```
directional=-1.400282,step=0.000000000000001776
xk1=0.789215
xk2=0.785550
xk3=0.789215
xk4=0.785550
f(x)=0.14651464
Iteration 6 finished!
```

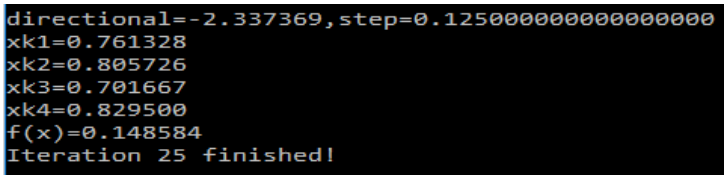
b) Pure Newton

```
After 2 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
f(xopt)=0.00000000
```

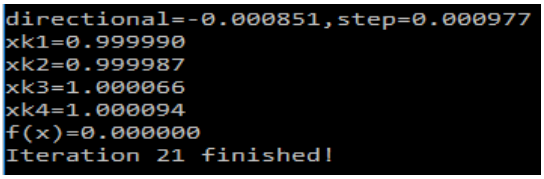
c) Modified Newton with Armijo rule




d) DFP with Armijo rule



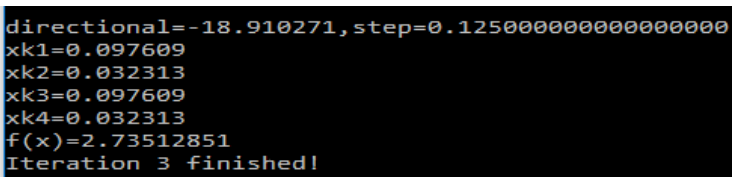

e) BFGS with Armijo rule




## Case 4 ($\alpha = 100$ and $n = 4$)

For these values of the parameters, some of the methods are stucked because of the Armijo function which is unable to provide a sufficiently positive step. Thus, the comparison between the methods will happen comparing how close the function value goes to the minimum value of zero and how many iterations it took the method to reach this value. Some asymmetries between the odd indexed and the even indexed points are due to numerical errors in the last iterations before Armijo is stucked.

a) Steepest Descent with Armijo rule



b) Pure Newton

```
After 2 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
f(xopt)=0.00000000
```

c) Modified Newton with Armijo rule

```
After 10748 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
f(xopt)=0.00000000
```

d) DFP with Armijo rule

```
directional=-16.820242,step=0.125000000000000000
xk1=0.125867
xk2=0.048270
xk3=0.125867
xk4=0.048270
f(x)=3.301188
Iteration 3 finished!
```

e) BFGS with Armijo rule

```
directional=-8.677343,step=0.000000
xk1=0.150565
xk2=-0.002230
xk3=0.113212
xk4=0.072475
f(x)=3.069585
Iteration 6 finished!
```

**Case 5 ($\alpha = 1$ and $n = 6$)**

For these values of the parameters, some of the methods are stucked because of the Armijo function which is unable to provide a sufficiently positive step. Thus, the comparison between the methods will happen comparing how close the function value goes to the minimum value of zero and how many iterations it took the method to reach this value. Some asymmetries between the odd indexed and the even indexed points are due to numerical errors in the last iterations before Armijo is stucked.

a) Steepest Descent with Armijo rule

```
directional=-0.574161,step=1.000000000000000000
xk1=0.932518
xk2=0.896855
xk3=0.932518
xk4=0.896855
xk5=0.932518
xk6=0.896855
f(x)=0.03766063
Iteration 7 finished!
```

b) Pure Newton

```
After 2 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
x5,opt=1.00000
x6,opt=1.00000
f(xopt)=0.00000000
```

c) Modified Newton with Armijo rule

```
After 8 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
x5,opt=1.00000
x6,opt=1.00000
f(xopt)=0.00000000
```

d) DFP with Armijo rule

```
directional=-0.233076,step=0.25000000000000000
xk1=0.996235
xk2=0.845860
xk3=0.886990
xk4=0.894041
xk5=1.061302
xk6=0.816935
f(x)=0.098970
Iteration 20 finished!
```

e) BFGS with Armijo rule

```
After 23 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
x5,opt=1.00000
x6,opt=1.00000
f(xopt)=0.00000000
```

## Case 6 ($\alpha = 100$ and $n = 6$)

For these values of the parameters, some of the methods are stucked because of the Armijo function which is unable to provide a sufficiently positive step. Thus, the comparison between the methods will happen comparing how close the function value goes to the minimum value of zero and how many iterations it took the method to reach this value. Some asymmetries between the odd indexed and the even indexed points are due to numerical errors in the last iterations before Armijo is stucked.

a) Steepest Descent with Armijo rule

```
directional=-5.117644,step=0.00000000000003553
xk1=0.019671
xk2=0.011052
xk3=0.019671
xk4=0.011052
xk5=0.019671
xk6=0.011052
f(x)=2.96107265
Iteration 13 finished!
```

b) Pure Newton

```
After 2 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
x5,opt=1.00000
x6,opt=1.00000
f(xopt)=0.00000000
```

c) Modified Newton with Armijo rule

```
After 10730 iter, we get:
x1,opt=0.99999
x2,opt=0.99999
x3,opt=0.99999
x4,opt=0.99999
x5,opt=0.99999
x6,opt=0.99999
f(xopt)=0.00000003
```

d) DFP with Armijo rule

```
directional=-15.700304,step=0.015625000000000000
xk1=-0.111786
xk2=0.055040
xk3=-0.111378
xk4=0.057147
xk5=-0.111378
xk6=0.057147
f(x)=5.054263
Iteration 8 finished!
```

e) BFGS with Armijo rule

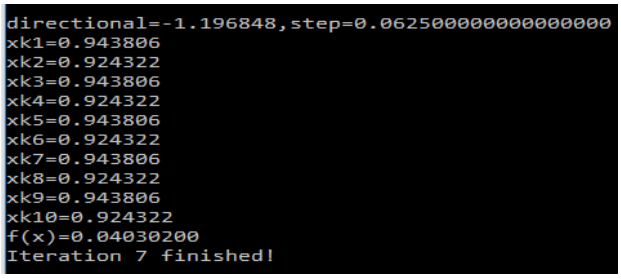
```
directional=-7.748655,step=0.125000
xk1=0.211524
xk2=0.046463
xk3=0.043148
xk4=0.013997
xk5=0.074720
xk6=-0.039942
f(x)=2.629523
Iteration 13 finished!
```

### Case 7 ($\alpha = 1$ and $n = 10$)

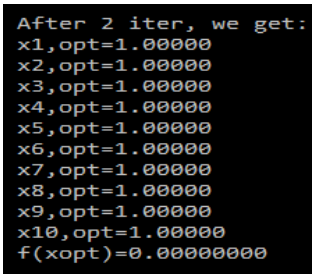
For these values of the parameters, some of the methods are stucked because of the Armijo function which is unable to provide a sufficiently positive step. Thus, the comparison between the methods will happen comparing how close the function value goes to the minimum value of

zero and how many iterations it took the method to reach this value. Some asymmetries between the odd indexed and the even indexed points are due to numerical errors in the last iterations before Armijo is stucked.

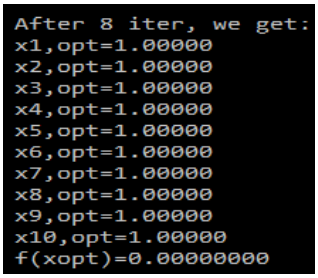
a) Steepest Descent with Armijo rule




b) Pure Newton




c) Modified Newton with Armijo rule



d) DFP with Armijo rule

```
directional=-2.797096,step=0.00000000000000444
xk1=0.492547
xk2=0.436418
xk3=0.492547
xk4=0.436418
xk5=0.492547
xk6=0.436418
xk7=0.492547
xk8=0.436418
xk9=0.492547
xk10=0.436418
f(x)=1.702979
Iteration 3 finished!
```

e) BFGS with Armijo rule

```
directional=-0.543099,step=2.000000
xk1=1.191419
xk2=1.213941
xk3=1.191419
xk4=1.213941
xk5=1.191419
xk6=1.213941
xk7=1.441071
xk8=0.985247
xk9=1.465962
xk10=1.287786
f(x)=0.487516
Iteration 7 finished!
```

## Case 8 ($\alpha = 100$ and $n = 10$)

For these values of the parameters, some of the methods are stucked because of the Armijo function which is unable to provide a sufficiently positive step. Thus, the comparison between the methods will happen comparing how close the function value goes to the minimum value of zero and how many iterations it took the method to reach this value. Some asymmetries between the odd indexed and the even indexed points are due to numerical errors in the last iterations before Armijo is stucked.

a) Steepest Descent with Armijo rule

```
directional=-10.425857,step=0.00000000000000111
xk1=0.258487
xk2=0.083656
xk3=0.258487
xk4=0.083656
xk5=0.258487
xk6=0.083656
xk7=0.258487
xk8=0.083656
xk9=0.258487
xk10=0.083656
f(x)=16.06326818
Iteration 3 finished!
```

b) Pure Newton

```
After 2 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
x5,opt=1.00000
x6,opt=1.00000
x7,opt=1.00000
x8,opt=1.00000
x9,opt=1.00000
x10,opt=1.00000
f(xopt)=0.00000000
```

c) Modified Newton with Armijo rule

```
After 3677 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
x5,opt=1.00000
x6,opt=1.00000
x7,opt=1.00000
x8,opt=1.00000
x9,opt=1.00000
x10,opt=1.00000
f(xopt)=0.00000000
```

d) DFP with Armijo rule

```
directional=-3.040845,step=0.000000000000000111
xk1=0.470531
xk2=0.224390
xk3=0.470531
xk4=0.224390
xk5=0.470531
xk6=0.224390
xk7=0.470531
xk8=0.224390
xk9=0.470531
xk10=0.224390
f(x)=37.357065
Iteration 5 finished!
```

e) BFGS with Armijo rule

```
directional=-330.207198,step=1.000000
xk1=0.470541
xk2=0.224393
xk3=0.470541
xk4=0.224393
xk5=0.470541
xk6=0.224393
xk7=0.470541
xk8=0.224393
xk9=0.470541
xk10=0.224393
f(x)=37.358473
Iteration 2 finished!
```

### Case 9 ($\alpha = 1$ and $n = 20$)

For these values of the parameters, some of the methods are stucked because of the Armijo function which is unable to provide a sufficiently positive step. Thus, the comparison between the methods will happen comparing how close the function value goes to the minimum value of zero and how many iterations it took the method to reach this value. Some asymmetries between the odd indexed and the even indexed points are due to numerical errors in the last iterations before Armijo is stucked.

a) Steepest Descent with Armijo rule

```
directional=-6.573783,step=0.50000000000000000
xk1=0.593367
xk2=0.572783
xk3=0.593367
xk4=0.572783
xk5=0.593367
xk6=0.572783
xk7=0.593367
xk8=0.572783
xk9=0.593367
xk10=0.572783
xk11=0.593367
xk12=0.572783
xk13=0.593367
xk14=0.572783
xk15=0.593367
xk16=0.572783
xk17=0.593367
xk18=0.572783
xk19=0.593367
xk20=0.572783
f(x)=2.33474913
Iteration 4 finished!
```

b) Pure Newton

```
After 2 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
x5,opt=1.00000
x6,opt=1.00000
x7,opt=1.00000
x8,opt=1.00000
x9,opt=1.00000
x10,opt=1.00000
x11,opt=1.00000
x12,opt=1.00000
x13,opt=1.00000
x14,opt=1.00000
x15,opt=1.00000
x16,opt=1.00000
x17,opt=1.00000
x18,opt=1.00000
x19,opt=1.00000
x20,opt=1.00000
f(xopt)=0.00000000
```

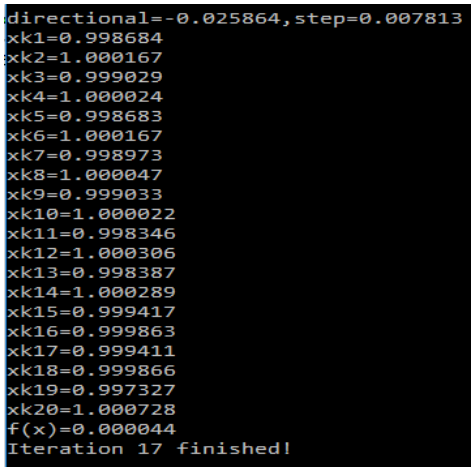
c) Modified Newton with Armijo rule

```
After 16 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
x5,opt=1.00000
x6,opt=1.00000
x7,opt=1.00000
x8,opt=1.00000
x9,opt=1.00000
x10,opt=1.00000
x11,opt=1.00000
x12,opt=1.00000
x13,opt=1.00000
x14,opt=1.00000
x15,opt=1.00000
x16,opt=1.00000
x17,opt=1.00000
x18,opt=1.00000
x19,opt=1.00000
x20,opt=1.00000
f(xopt)=0.00000000
```

d) DFP with Armijo rule

```
directional=-0.620022,step=1.0000000000000000
xk1=0.717714
xk2=0.679645
xk3=0.717714
xk4=0.679645
xk5=0.990596
xk6=0.543122
xk7=0.990596
xk8=0.543122
xk9=0.897331
xk10=0.544724
xk11=1.023480
xk12=0.600181
xk13=1.172403
xk14=0.525404
xk15=0.916370
xk16=0.577690
xk17=1.149728
xk18=0.548460
xk19=0.990596
xk20=0.543122
f(x)=3.225863
Iteration 20 finished!
```

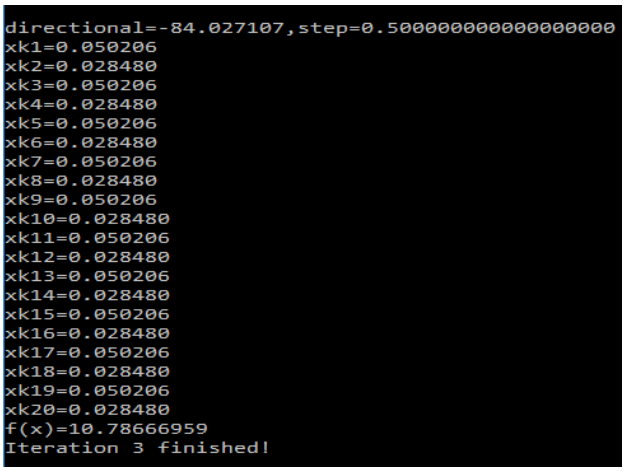
e) BFGS with Armijo rule




## Case 10 ($\alpha = 100$ and $n = 20$)

For these values of the parameters, some of the methods are stucked because of the Armijo function which is unable to provide a sufficiently positive step. Thus, the comparison between the methods will happen comparing how close the function value goes to the minimum value of zero and how many iterations it took the method to reach this value. Some asymmetries between the odd indexed and the even indexed points are due to numerical errors in the last iterations before Armijo is stucked.

a) Steepest Descent with Armijo rule



b) Pure Newton

```
After 2 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
x5,opt=1.00000
x6,opt=1.00000
x7,opt=1.00000
x8,opt=1.00000
x9,opt=1.00000
x10,opt=1.00000
x11,opt=1.00000
x12,opt=1.00000
x13,opt=1.00000
x14,opt=1.00000
x15,opt=1.00000
x16,opt=1.00000
x17,opt=1.00000
x18,opt=1.00000
x19,opt=1.00000
x20,opt=1.00000
f(xopt)=0.00000000
```

c) Modified Newton with Armijo rule

```
After 10373 iter, we get:
x1,opt=1.00000
x2,opt=1.00000
x3,opt=1.00000
x4,opt=1.00000
x5,opt=1.00000
x6,opt=1.00000
x7,opt=1.00000
x8,opt=1.00000
x9,opt=1.00000
x10,opt=1.00000
x11,opt=1.00000
x12,opt=1.00000
x13,opt=1.00000
x14,opt=1.00000
x15,opt=1.00000
x16,opt=1.00000
x17,opt=1.00000
x18,opt=1.00000
x19,opt=1.00000
x20,opt=1.00000
f(xopt)=0.00000001
```

d) DFP with Armijo rule

```
directional=-2.091633,step=0.007812500000000000
xk1=0.009953
xk2=-0.006593
xk3=0.009326
xk4=-0.006881
xk5=0.009043
xk6=-0.006089
xk7=0.005172
xk8=-0.009550
xk9=0.010539
xk10=-0.009795
xk11=0.007179
xk12=-0.014138
xk13=0.006630
xk14=-0.009907
xk15=0.006479
xk16=-0.015622
xk17=0.008378
xk18=-0.013364
xk19=0.009682
xk20=-0.014744
f(x)=10.057874
Iteration 16 finished!
```

e) BFGS with Armijo rule

```
directional=-3.352336,step=0.062500
xk1=0.013736
xk2=0.003492
xk3=0.013728
xk4=0.006370
xk5=0.013721
xk6=0.008706
xk7=0.013728
xk8=0.006410
xk9=0.013764
xk10=-0.006379
xk11=0.013770
xk12=-0.008457
xk13=0.013800
xk14=-0.019181
xk15=0.013799
xk16=-0.018787
xk17=0.013826
xk18=-0.028299
xk19=0.013850
xk20=-0.036913
f(x)=9.940799
Iteration 8 finished!
```

# Conclusion

From the extensive simulations that were shown above, we can conclude that Pure Newton's is the fastest method in order to find the minimum of the function since it is able to find it in just 2 iterations. This result was expected since we know that Newton's method has a quadratic rate of convergence.

As far as it concerns modified Newton's method with Armijo rule, we could say that this method showed the second best behavior since it needed more iterations than Pure Newton to converge, but it always converged!

BFGS method and DFP method had a similar performance but in general, BFGS converged more times than DFP and it also converged faster.

Last, as far as it concerns the Steepest Descent method, we saw that it was stucked too many times because of the Armijo rule and while it was approaching the minimum, it required many iterations in order to actually reach it. This behavior was also expected since it is known that Steepest Descent method presents the zigzagging effect.

# Appendix

Here we present the code that we wrote in **C programming language** in order to illustrate the above methods. As it was requested, the code was written as a whole package with some if commands who asked the user to "Press 1 if you want to solve it with…"etc. However, I preferred to present my codes for each method separately since otherwise, it would be impossible for you to check its validity!

1) Steepest Descent with Armijo rule

```
#include<stdio.h>

#include<math.h>


double fun (double x[],int a,int n)

{

   int i;

            double func=0;

            for (i=0;i<n/2;i++)

            {

                        func=func+a*pow(x[i+1]-pow(x[i],2),2)+pow(1-x[i],2);

            }

   return (func);

}


double subgradient_odd (double xi,double xiplus1,int a)

{

   return ((-4)*a*xi*(xiplus1-pow(xi,2))+2*(xi-1));

}
```

```
double subgradient_even (double xi,double xiplus1,int a)
{
   return (2*a*(xiplus1-pow(xi,2)));
}

double armijo (double x[],double d[],double gradient[],int a,int n)
{
        int i;
        float theta=0.2,eta=2;
        double lam=1,funpr,fdd=0,funx,fxeta;
        funpr=fun(x,a,n);
        for (i=0;i<n;i++)
        {
                fdd=fdd+gradient[i]*d[i];
                x[i]=x[i]+lam*d[i];
        }
        funx=fun(x,a,n);
        if (fun(x,a,n)<=funpr+theta*lam*fdd)
        {
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]+lam*d[i];
                }
                fxeta=fun(x,a,n);
                if (fxeta<=funpr+eta*theta*lam*fdd)
                {
                        lam=2;
```

```
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]+lam*d[i];
                }
                fxeta=fun(x,a,n);
        }
        while (fun(x,a,n)<=funpr+eta*theta*lam*fdd)
        {
                lam=eta*lam;
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]+lam*d[i];
                }
                fxeta=fun(x,a,n);
        }
}
else
{
        lam=0.5;
        for (i=0;i<n;i++)
        {
                x[i]=x[i]-lam*d[i];
        }
        funx=fun(x,a,n);
        while (fun(x,a,n)>funpr+theta*lam*fdd)
        {
                lam=lam/eta;
```

```
                        for (i=0;i<n;i++)
                        {
                                x[i]=x[i]-lam*d[i];
                        }
                        funx=fun(x,a,n);
                }
        }
        printf("directional=%f,step=%2.18f\n",fdd,lam);
        return (lam);
}



int main()
{
        /* Steepest Descent with Armijo step */
        int a,i,j,choice,n,iter=0;
        printf("Choose the parameters a and n:\n");
        scanf("%d %d",&a,&n);
        double step=1,x[n],xk[n],d[n],gradient[n],fun_eval;
        double epsilon=pow(10,-5),norm=0,ssgrad=0;
        for (i=0;i<n;i++)
        {
                gradient[i]=0;
        }
        for (i=0;i<n;i=i+2)
        {
```

```
            xk[i]=-1;
            x[i]=-1;
}
for (i=1;i<n;i=i+2)
{
            xk[i]=-1;
            x[i]=-1;
}
fun_eval=fun(x,a,n);
for (i=0;i<n;i=i+2)
{
            gradient[i]=subgradient_odd(x[i],x[i+1],a);
            d[i]=-gradient[i];
}
for (i=1;i<n;i=i+2)
{
            gradient[i]=subgradient_even(x[i-1],x[i],a);
            d[i]=-gradient[i];
}
for (i=0;i<n;i++)
{
            printf("gradient%d=%f\n",i+1,gradient[i]);
}
for (i=0;i<n;i++)
{
            ssgrad=ssgrad+pow(gradient[i],2);
}
```

```
norm=sqrt(ssgrad);
for (i=0;i<n;i++)
{
	d[i]=d[i]/norm;
}
while (norm>=epsilon)
{
	step=armijo(x,d,gradient,a,n);
	for (i=0;i<n;i++)
	{
		xk[i]=xk[i]+step*d[i];
		x[i]=xk[i];
		printf("xk%d=%f\n",i+1,xk[i]);
	}
	for (i=0;i<n;i=i+2)
	{
		gradient[i]=subgradient_odd(x[i],x[i+1],a);
		d[i]=-gradient[i];
	}
	for (i=1;i<n;i=i+2)
	{
		gradient[i]=subgradient_even(x[i-1],x[i],a);
		d[i]=-gradient[i];
	}
	ssgrad=0;
	for (i=0;i<n;i++)
	{
```

```
                ssgrad=ssgrad+pow(gradient[i],2);
            }
            norm=sqrt(ssgrad);
            for (i=0;i<n;i++)
            {
                d[i]=d[i]/norm;
            }
            iter=iter+1;
            printf("f(x)=%3.8f\n",fun(xk,a,n));
            printf("Iteration %d finished!\n\n",iter);
        }
        printf("\tAfter %d iter, we get:\n",iter);
        for (i=0;i<n;i++)
        {
            printf("\tx%d,opt=%2.5f\n",i+1,xk[i]);
        }
        printf("\tf(xopt)=%3.8f\n",fun(xk,a,n));
        return 0;
}
```

2) Pure Newton

```
#include<stdio.h>
#include<math.h>

double fun (double x[],int a,int n)
{
   int i;
            double func=0;
            for (i=0;i<n/2;i++)
            {
                        func=func+a*pow(x[i+1]-pow(x[i],2),2)+pow(1-x[i],2);
            }
   return (func);
}

double subgradient_odd (double xi,double xiplus1,int a)
{
   return ((-4)*a*xi*(xiplus1-pow(xi,2))+2*(xi-1));
}

double subgradient_even (double xi,double xiplus1,int a)
{
   return (2*a*(xiplus1-pow(xi,2)));
}
```

```
double hess_diag_i_i_odd (double xi,double xiplus1,int a)
{
	return (2+8*a*pow(xi,2)-4*a*(xiplus1-xi));
}

double hess_diag_i_i_even (double xi,double xiplus1,int a)
{
	return (2*a);
}

double hess_i_iplus1 (double xi,int a)
{
	return ((-4)*a*xi);
}

double armijo (double x[],double d[],double gradient[],int a,int n)
{
	int i;
	float theta=0.2,eta=2;
	double lam=1,funpr,fdd=0,funx,fxeta;
	funpr=fun(x,a,n);
	for (i=0;i<n;i++)
	{
		fdd=fdd+gradient[i]*d[i];
		x[i]=x[i]+lam*d[i];
```

```
}
funx=fun(x,a,n);
if (fun(x,a,n)<=funpr+theta*lam*fdd)
{
        for (i=0;i<n;i++)
        {
                x[i]=x[i]+lam*d[i];
        }
        fxeta=fun(x,a,n);
        if (fxeta<=funpr+eta*theta*lam*fdd)
        {
                lam=2;
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]+lam*d[i];
                }
                fxeta=fun(x,a,n);
        }
        while (fun(x,a,n)<=funpr+eta*theta*lam*fdd)
        {
                lam=eta*lam;
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]+lam*d[i];
                }
                fxeta=fun(x,a,n);
        }
```

```
	}
	else
	{
		lam=0.5;
		for (i=0;i<n;i++)
		{
			x[i]=x[i]-lam*d[i];
		}
		funx=fun(x,a,n);
		while (fun(x,a,n)>funpr+theta*lam*fdd)
		{
			lam=lam/eta;
			for (i=0;i<n;i++)
			{
				x[i]=x[i]-lam*d[i];
			}
			funx=fun(x,a,n);
		}
	}
	printf("directional=%f,step=%2.18f\n",fdd,lam);
	return (lam);
}



int main()
{
	/* Pure Newton and Steepest descent with Armijo if necessary*/
```

```
int a,i,j,n,iter=0;
printf("Choose the parameters a and n:\n");
scanf("%d %d",&a,&n);
double x[n],xk[n],dir[n],gradient[n],fun_eval,hessian[n][n];
double epsilon=pow(10,-5),norm=0,ssgrad=0,step,norm2=0,ssgrad2=0;
for (i=0;i<n;i++)
{
        gradient[i]=0;
        for (j=0;j<n;j++)
        {
                hessian[i][j]=0;
        }
}
for (i=0;i<n;i=i+2)
{
        xk[i]=-1;
        x[i]=-1;
}
for (i=1;i<n;i=i+2)
{
        xk[i]=-1;
        x[i]=-1;
}
fun_eval=fun(x,a,n);
printf("initial f(x)=%f\n",fun_eval);
for (i=0;i<n-1;i=i+2)
{
```

```
            hessian[i][i]=hess_diag_i_i_odd(x[i],x[i+1],a);
            hessian[i][i+1]=hess_i_iplus1(x[i],a);
}
for (i=1;i<n;i=i+2)
{
            hessian[i][i]=hess_diag_i_i_even(x[i-1],x[i],a);
            hessian[i][i-1]=hess_i_iplus1(x[i-1],a);
}
for (i=0;i<n;i++)
{
            for (j=0;j<n;j++)
            {
                        printf("%f\t",hessian[i][j]);
            }
            printf("\n");
}
for (i=0;i<n;i=i+2)
{
            gradient[i]=subgradient_odd(x[i],x[i+1],a);
}
for (i=1;i<n;i=i+2)
{
            gradient[i]=subgradient_even(x[i-1],x[i],a);
}
for (i=0;i<n;i++)
{
            printf("gradient%d=%f\n",i+1,gradient[i]);
```

```
}
printf("\n");
for (i=0;i<n;i++)
{
        ssgrad=ssgrad+pow(gradient[i],2);
}
norm=sqrt(ssgrad);
int s;
double dchol[n],c[n][n],lchol[n][n];
while (norm>=epsilon)
{
        /* Cholesky factorization in order to check p.d.*/
        for (i=0;i<n;i++)
        {
                dchol[i]=0;
                for (j=0;j<n;j++)
                {
                        c[i][j]=0;
                        lchol[i][j]=0;
                }
        }
        for (i=0;i<n;i++)
        {
                lchol[i][i]=1;
        }
        double sum1,sum2;
        for (j=0;j<n;j++)
```

```
{
        sum1=0;
        for (s=0;s<=j-1;s++)
        {
                sum1=sum1+dchol[s]*pow(lchol[j][s],2);
        }
        c[j][j]=hessian[j][j]-sum1;
        dchol[j]=c[j][j];
        for (i=j+1;i<n;i++)
        {
                sum2=0;
                for (s=0;s<=j-1;s++)
                {
                        sum2=sum2+dchol[s]*lchol[i][s]*lchol[j][s];
                }
                c[i][j]=hessian[i][j]-sum2;
                lchol[i][j]=c[i][j]/dchol[j];
        }
}
/* Check for p.d. and Steepest Descent step if necessary*/
int test=0;
for (i=0;i<n;i++)
{
        if (dchol[i]<=0)
        {
                test=test+1;
        }
```

```
}
if (test>0)
{
        printf("Steepest Descent step\n");
        for (i=0;i<n;i++)
        {
                dir[i]=-gradient[i]/norm;
        }
}
else
/* Solve the system in order to find the direction */
{
        double delta[n],epsilon[n];
        for (i=0;i<n;i++)
        {
                dir[i]=0;
                delta[i]=0;
                epsilon[i]=0;
        }
        for (i=0;i<n;i++)
        {
                for (j=0;j<=i-1;j++)
                {
                        delta[i]=delta[i]-(lchol[i][j]*delta[j]/lchol[i][i]);
                }
                delta[i]=delta[i]-gradient[i]/lchol[i][i];
        }
```

```
        for (i=0;i<n;i++)
        {
                epsilon[i]=delta[i]/dchol[i];
        }
        for (i=n-1;i>=0;i=i-1)
        {
                for (j=i+1;j<n;j++)
                {
                        dir[i]=dir[i]-(lchol[j][i]*dir[j]/lchol[i][i]);
                }
                dir[i]=dir[i]+epsilon[i]/lchol[i][i];
        }
}
/* Update points */
step=1;
if (test>0)
{
        step=armijo(x,dir,gradient,a,n);
}
for (i=0;i<n;i++)
{
        xk[i]=xk[i]+step*dir[i];
        printf("xk%d=%f\n",i+1,xk[i]);
        x[i]=xk[i];
}
/* Update gradient and Hessian */
for (i=0;i<n;i=i+2)
```

```
{
        hessian[i][i]=hess_diag_i_i_odd(x[i],x[i+1],a);
        hessian[i][i+1]=hess_i_iplus1(x[i],a);
}
for (i=1;i<n;i=i+2)
{
        hessian[i][i]=hess_diag_i_i_even(x[i-1],x[i],a);
        hessian[i][i-1]=hess_i_iplus1(x[i-1],a);
}
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                printf("%f\t",hessian[i][j]);
        }
        printf("\n");
}
for (i=0;i<n;i=i+2)
{
        gradient[i]=subgradient_odd(x[i],x[i+1],a);
}
for (i=1;i<n;i=i+2)
{
        gradient[i]=subgradient_even(x[i-1],x[i],a);
}
for (i=0;i<n;i++)
{
```

```
                printf("gradient%d=%f\n",i+1,gradient[i]);
        }
        ssgrad=0;
        for (i=0;i<n;i++)
        {
                ssgrad=ssgrad+pow(gradient[i],2);
        }
        fun_eval=fun(x,a,n);
        printf("f(x)=%f\n",fun_eval);
        norm=sqrt(ssgrad);
        iter=iter+1;
        printf("Iteration %d finished!\n\n",iter);
    }
    printf("\tAfter %d iter, we get:\n",iter);
    for (i=0;i<n;i++)
    {
        printf("\tx%d,opt=%2.5f\n",i+1,xk[i]);
    }
    printf("\tf(xopt)=%3.8f\n",fun(xk,a,n));
    return 0;
}
```

3) Modified Newton with Armijo rule

```
#include<stdio.h>

#include<math.h>


double fun (double x[],int a,int n)

{

   int i;

            double func=0;

            for (i=0;i<n/2;i++)

            {

                        func=func+a*pow(x[i+1]-pow(x[i],2),2)+pow(1-x[i],2);

            }

   return (func);

}


double subgradient_odd (double xi,double xiplus1,int a)

{

   return ((-4)*a*xi*(xiplus1-pow(xi,2))+2*(xi-1));

}


double subgradient_even (double xi,double xiplus1,int a)

{

   return (2*a*(xiplus1-pow(xi,2)));

}
```

```
double hess_diag_i_i_odd (double xi,double xiplus1,int a)
{
	return (2+8*a*pow(xi,2)-4*a*(xiplus1-xi));
}

double hess_diag_i_i_even (double xi,double xiplus1,int a)
{
	return (2*a);
}

double hess_i_iplus1 (double xi,int a)
{
	return ((-4)*a*xi);
}

double armijo (double x[],double d[],double gradient[],int a,int n)
{
	int i;
	float theta=0.2,eta=2;
	double lam=1,funpr,fdd=0,funx,fxeta;
	funpr=fun(x,a,n);
	for (i=0;i<n;i++)
	{
		fdd=fdd+gradient[i]*d[i];
		x[i]=x[i]+lam*d[i];
	}
	funx=fun(x,a,n);
```

```
if (fun(x,a,n)<=funpr+theta*lam*fdd)
{
        for (i=0;i<n;i++)
        {
                x[i]=x[i]+lam*d[i];
        }
        fxeta=fun(x,a,n);
        if (fxeta<=funpr+eta*theta*lam*fdd)
        {
                lam=2;
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]+lam*d[i];
                }
                fxeta=fun(x,a,n);
        }
        while (fun(x,a,n)<=funpr+eta*theta*lam*fdd)
        {
                lam=eta*lam;
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]+lam*d[i];
                }
                fxeta=fun(x,a,n);
        }
}
else
```

```
	{
		lam=0.5;
		for (i=0;i<n;i++)
		{
			x[i]=x[i]-lam*d[i];
		}
		funx=fun(x,a,n);
		while ((fun(x,a,n)>funpr+theta*lam*fdd)&&(lam>0.005))
		{
			lam=lam/eta;
			for (i=0;i<n;i++)
			{
				x[i]=x[i]-lam*d[i];
			}
			funx=fun(x,a,n);
		}
	}
	printf("directional=%f,step=%f\n",fdd,lam);
	return (lam);
}

int main()
{
	/* Newton with Armijo and Steepest descent step if necessary*/
	int a,i,j,n,iter=0;
	printf("Choose the parameters a and n:\n");
	scanf("%d %d",&a,&n);
```

```
double x[n],xk[n],dir[n],gradient[n],fun_eval,hessian[n][n];
double epsilon=pow(10,-5),norm=0,ssgrad=0,step;
for (i=0;i<n;i++)
{
        gradient[i]=0;
        for (j=0;j<n;j++)
        {
                hessian[i][j]=0;
        }
}
for (i=0;i<n;i=i+2)
{
        xk[i]=-1;
        x[i]=-1;
}
for (i=1;i<n;i=i+2)
{
        xk[i]=-1;
        x[i]=-1;
}
fun_eval=fun(x,a,n);
printf("initial f(x)=%f\n",fun_eval);
for (i=0;i<n-1;i=i+2)
{
        hessian[i][i]=hess_diag_i_i_odd(x[i],x[i+1],a);
        hessian[i][i+1]=hess_i_iplus1(x[i],a);
}
```

```
for (i=1;i<n;i=i+2)
{
        hessian[i][i]=hess_diag_i_i_even(x[i],x[i+1],a);
        hessian[i][i-1]=hess_i_iplus1(x[i-1],a);
}
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                printf("%f\t",hessian[i][j]);
        }
        printf("\n");
}
for (i=0;i<n;i=i+2)
{
        gradient[i]=subgradient_odd(x[i],x[i+1],a);
}
for (i=1;i<n;i=i+2)
{
        gradient[i]=subgradient_even(x[i-1],x[i],a);
}
for (i=0;i<n;i++)
{
        printf("gradient%d=%f\n",i+1,gradient[i]);
}
printf("\n");
for (i=0;i<n;i++)
```

```
{
        ssgrad=ssgrad+pow(gradient[i],2);
}
norm=sqrt(ssgrad);
int s;
double dchol[n],c[n][n],lchol[n][n];
while (norm>=epsilon)
{
        /* Cholesky factorization in order to check p.d.*/
        for (i=0;i<n;i++)
        {
                dchol[i]=0;
                for (j=0;j<n;j++)
                {
                        c[i][j]=0;
                        lchol[i][j]=0;
                }
        }
        for (i=0;i<n;i++)
        {
                lchol[i][i]=1;
        }
        double sum1,sum2;
        for (j=0;j<n;j++)
        {
                sum1=0;
                for (s=0;s<=j-1;s++)
```

```
        {
                sum1=sum1+dchol[s]*pow(lchol[j][s],2);
        }
        c[j][j]=hessian[j][j]-sum1;
        dchol[j]=c[j][j];
        for (i=j+1;i<n;i++)
        {
                sum2=0;
                for (s=0;s<=j-1;s++)
                {
                        sum2=sum2+dchol[s]*lchol[i][s]*lchol[j][s];
                }
                c[i][j]=hessian[i][j]-sum2;
                lchol[i][j]=c[i][j]/dchol[j];
        }
}
/* Check for p.d. and Steepest Descent step if necessary*/
int test=0;
for (i=0;i<n;i++)
{
        if (dchol[i]<=0)
        {
                test=test+1;
        }
}
if (test>0)
{
```

```
		printf("Steepest Descent step\n");
		for (i=0;i<n;i++)
		{
			dir[i]=-gradient[i]/norm;
		}
}
else
/* Solve the system in order to find the direction */
{
		double delta[n],epsilon[n];
		for (i=0;i<n;i++)
		{
			dir[i]=0;
			delta[i]=0;
			epsilon[i]=0;
		}
		for (i=0;i<n;i++)
		{
			for (j=0;j<=i-1;j++)
			{
				delta[i]=delta[i]-(lchol[i][j]*delta[j]/lchol[i][i]);
			}
			delta[i]=delta[i]-gradient[i]/lchol[i][i];
		}
		for (i=0;i<n;i++)
		{
			epsilon[i]=delta[i]/dchol[i];
```

```
		}
		for (i=n-1;i>=0;i=i-1)
		{
			for (j=i+1;j<n;j++)
			{
				dir[i]=dir[i]-(lchol[j][i]*dir[j]/lchol[i][i]);
			}
			dir[i]=dir[i]+epsilon[i]/lchol[i][i];
		}
}
/* Update points */
step=armijo(x,dir,gradient,a,n);
for (i=0;i<n;i++)
{
		xk[i]=xk[i]+step*dir[i];
		printf("xk%d=%f\n",i+1,xk[i]);
		x[i]=xk[i];
}
/* Update gradient and Hessian */
for (i=0;i<n-1;i=i+2)
{
		hessian[i][i]=hess_diag_i_i_odd(x[i],x[i+1],a);
		hessian[i][i+1]=hess_i_iplus1(x[i],a);
}
for (i=1;i<n;i=i+2)
{
		hessian[i][i]=hess_diag_i_i_even(x[i],x[i+1],a);
```

```
                hessian[i][i-1]=hess_i_iplus1(x[i-1],a);
        }
        for (i=0;i<n;i=i+2)
        {
                gradient[i]=subgradient_odd(x[i],x[i+1],a);
        }
        for (i=1;i<n;i=i+2)
        {
                gradient[i]=subgradient_even(x[i-1],x[i],a);
        }
        ssgrad=0;
        for (i=0;i<n;i++)
        {
                ssgrad=ssgrad+pow(gradient[i],2);
        }
        fun_eval=fun(x,a,n);
        printf("f(x)=%f\n\n",fun_eval);
        norm=sqrt(ssgrad);
        iter=iter+1;
}
printf("\tAfter %d iter, we get:\n",iter);
for (i=0;i<n;i++)
{
        printf("\tx%d,opt=%2.5f\n",i+1,xk[i]);
}
printf("\tf(xopt)=%3.8f\n",fun(xk,a,n));
return 0;
```

4) DFP with Armijo rule

```
#include<stdio.h>
#include<math.h>

double fun (double x[],int a,int n)
{
   int i;
            double func=0;
            for (i=0;i<n/2;i++)
            {
                        func=func+a*pow(x[i+1]-pow(x[i],2),2)+pow(1-x[i],2);
            }
   return (func);
}

double subgradient_odd (double xi,double xiplus1,int a)
{
            double value=0;
            value=(-4)*a*xi*(xiplus1-pow(xi,2))+2*(xi-1);
   return (value);
}

double subgradient_even (double xi,double xiplus1,int a)
{
   return (2*a*(xiplus1-pow(xi,2)));
}
```

```
double armijo (double x[],double d[],double gradient[],int a,int n)
{
        int i;
        float theta=0.2,eta=2;
        double lam=1,funpr,fdd=0,funx,fxeta;
        funpr=fun(x,a,n);
        for (i=0;i<n;i++)
        {
                fdd=fdd+gradient[i]*d[i];
                x[i]=x[i]+lam*d[i];
        }
        funx=fun(x,a,n);
        if (fun(x,a,n)<=funpr+theta*lam*fdd)
        {
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]+lam*d[i];
                }
                fxeta=fun(x,a,n);
                if (fxeta<=funpr+eta*theta*lam*fdd)
                {
                        lam=2;
                        for (i=0;i<n;i++)
                        {
                                x[i]=x[i]+lam*d[i];
                        }
                        fxeta=fun(x,a,n);
```

```
        }
        while (fun(x,a,n)<=funpr+eta*theta*lam*fdd)
        {
                lam=eta*lam;
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]+lam*d[i];
                }
                fxeta=fun(x,a,n);
        }
}
else
{
        lam=0.5;
        for (i=0;i<n;i++)
        {
                x[i]=x[i]-lam*d[i];
        }
        funx=fun(x,a,n);
        while (fun(x,a,n)>funpr+theta*lam*fdd)
        {
                lam=lam/eta;
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]-lam*d[i];
                }
                funx=fun(x,a,n);
```

```
                }
        }
        printf("directional=%f,step=%2.18f\n",fdd,lam);
        return (lam);
}



int main()
{
        /* DFP with Armijo using equation (6.15) p.139 Nocedal & Wright*/
        int a,i,j,n,iter=0;
        printf("Choose the parameters a and n:\n");
        scanf("%d %d",&a,&n);
        double x[n],xk[n],dir[n],gradient[n],gradientk[n],fun_eval,H_dfp[n][n];
        double epsilon=pow(10,-5),norm=0,ssgrad=0,step,s[n],y[n],test;
        double Hy[n],ytH[n],HyytH[n][n],ytHy,frac1[n][n],sst[n][n],frac2[n][n],ssdir;
        for (i=0;i<n;i++)
        {
                s[i]=0;
                y[i]=0;
                dir[i]=0;
                gradientk[i]=0;
                gradient[i]=0;
                for (j=0;j<n;j++)
                {
                        H_dfp[i][j]=0;
                }
```

```
}
for (i=0;i<n;i=i+2)
{
        xk[i]=-1;
        x[i]=-1;
}
for (i=1;i<n;i=i+2)
{
        xk[i]=-1;
        x[i]=-1;
}
/* Take the initial H_dfp to be identity */
for (i=0;i<n;i++)
{
        H_dfp[i][i]=1;
}
printf("\nInitialization\n");
printf("f(x)=%f\n",fun(x,a,n));
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                printf("%f\t",H_dfp[i][j]);
        }
        printf("\n");
}
fun_eval=fun(x,a,n);
```

```
for (i=0;i<n;i=i+2)
{
        gradient[i]=subgradient_odd(x[i],x[i+1],a);
        gradientk[i]=gradient[i];
}
for (i=1;i<n;i=i+2)
{
        gradient[i]=subgradient_even(x[i-1],x[i],a);
        gradientk[i]=gradient[i];
}
for (i=0;i<n;i++)
{
        printf("gradient%d=%f\n",i+1,gradient[i]);
}
printf("\n");
for (i=0;i<n;i++)
{
        ssgrad=ssgrad+pow(gradient[i],2);
}
norm=sqrt(ssgrad);
while (norm>=epsilon)
{
        /* Find initial direction */
        for (i=0;i<n;i++)
        {
                dir[i]=0;
                for (j=0;j<n;j++)
```

```
            {
                        dir[i]=dir[i]-H_dfp[i][j]*gradient[j];
            }
}
ssdir=0;
for (i=0;i<n;i++)
{
            ssdir=ssdir+pow(dir[i],2);
}
ssdir=sqrt(ssdir);
for (i=0;i<n;i++)
{
            dir[i]=dir[i]/ssdir;
}
for (i=0;i<n;i++)
{
            printf("dir%d=%f\n",i+1,dir[i]);
}
step=armijo(x,dir,gradient,a,n);
/* Initial update of points */
for (i=0;i<n;i++)
{
            x[i]=xk[i];
            x[i]=x[i]+step*dir[i];
}
/* Define sk and yk and check if their inner product is positive */
for (i=0;i<n;i=i+2)
```

```
{
        gradient[i]=subgradient_odd(x[i],x[i+1],a);
}
for (i=1;i<n;i=i+2)
{
        gradient[i]=subgradient_even(x[i-1],x[i],a);
}
for (i=0;i<n;i++)
{
        s[i]=x[i]-xk[i];
        y[i]=gradient[i]-gradientk[i];
}
test=0;
for (i=0;i<n;i++)
{
        test=test+s[i]*y[i];
}
/* DFP Inverse Hessian Update */
if (test>0)
{
        /* Create Hk*yk */
        for (i=0;i<n;i++)
        {
                Hy[i]=0;
                for (j=0;j<n;j++)
                {
                        Hy[i]=Hy[i]+H_dfp[i][j]*y[j];
```

```
            }
}
/* Create yk(transpose)*Hk */
for (i=0;i<n;i++)
{
            ytH[i]=0;
            for (j=0;j<n;j++)
            {
                        ytH[i]=ytH[i]+y[j]*H_dfp[j][i];
            }
}
/* Create Hk*yk*yk(transpose)*Hk */
for (i=0;i<n;i++)
{
            for (j=0;j<n;j++)
            {
                        HyytH[i][j]=Hy[i]*ytH[j];
            }
}
/* Create yk(transpose)*Hk*yk */
ytHy=0;
for (i=0;i<n;i++)
{
            ytHy=ytHy+ytH[i]*y[i];
}
/* Create the 1st fraction of the equation 6.15 */
for (i=0;i<n;i++)
```

```
{
        for (j=0;j<n;j++)
        {
                frac1[i][j]=HyytH[i][j]/ytHy;
        }
}
/* Create sk*sk(transpose) */
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                sst[i][j]=s[i]*s[j];
        }
}
/* Create the 2nd fraction of the equation 6.15 */
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                frac2[i][j]=sst[i][j]/test;
        }
}
/* Update Hk+1 */
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
```

```
                                H_dfp[i][j]=H_dfp[i][j]-frac1[i][j]+frac2[i][j];
                        }
                }
        }
        else
        {
                /* if sk(transpose)*yk<=0 then do not do the update! */
                for (i=0;i<n;i++)
                {
                        for (j=0;j<n;j++)
                        {
                                H_dfp[i][j]=H_dfp[i][j];
                        }
                }
        }
        for (i=0;i<n;i++)
        {
                xk[i]=x[i];
                printf("xk%d=%f\n",i+1,xk[i]);
                gradientk[i]=gradient[i];
        }
        ssgrad=0;
        for (i=0;i<n;i++)
        {
                ssgrad=ssgrad+pow(gradient[i],2);
        }
        fun_eval=fun(x,a,n);
```

```
            printf("f(x)=%f\n",fun_eval);
            norm=sqrt(ssgrad);
            iter=iter+1;
            printf("Iteration %d finished!\n\n",iter);
        }
        printf("\tAfter %d iter, we get:\n",iter);
        for (i=0;i<n;i++)
        {
            printf("\tx%d,opt=%2.5f\n",i+1,xk[i]);
        }
        printf("\tf(xopt)=%3.8f\n",fun(xk,a,n));
        return 0;
}
```

5) BFGS with Armijo rule

```
#include<stdio.h>
#include<math.h>

double fun (double x[],int a,int n)
{
   int i;
            double func=0;
            for (i=0;i<n/2;i++)
            {
                        func=func+a*pow(x[i+1]-pow(x[i],2),2)+pow(1-x[i],2);
            }
   return (func);
}

double subgradient_odd (double xi,double xiplus1,int a)
{
            double value=0;
            value=(-4)*a*xi*(xiplus1-pow(xi,2))+2*(xi-1);
   return (value);
}

double subgradient_even (double xi,double xiplus1,int a)
{
   return (2*a*(xiplus1-pow(xi,2)));
}
```

```
double armijo (double x[],double d[],double gradient[],int a,int n)
{
	int i;
	float theta=0.2,eta=2;
	double lam=1,funpr,fdd=0,funx,fxeta;
	funpr=fun(x,a,n);
	for (i=0;i<n;i++)
	{
		fdd=fdd+gradient[i]*d[i];
		x[i]=x[i]+lam*d[i];
	}
	funx=fun(x,a,n);
	if (fun(x,a,n)<=funpr+theta*lam*fdd)
	{
		for (i=0;i<n;i++)
		{
			x[i]=x[i]+lam*d[i];
		}
		fxeta=fun(x,a,n);
		if (fxeta<=funpr+eta*theta*lam*fdd)
		{
			lam=2;
			for (i=0;i<n;i++)
			{
				x[i]=x[i]+lam*d[i];
			}
			fxeta=fun(x,a,n);
```

```
        }
        while (fun(x,a,n)<=funpr+eta*theta*lam*fdd)
        {
                lam=eta*lam;
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]+lam*d[i];
                }
                fxeta=fun(x,a,n);
        }
}
else
{
        lam=0.5;
        for (i=0;i<n;i++)
        {
                x[i]=x[i]-lam*d[i];
        }
        funx=fun(x,a,n);
        while (fun(x,a,n)>funpr+theta*lam*fdd)
        {
                lam=lam/eta;
                for (i=0;i<n;i++)
                {
                        x[i]=x[i]-lam*d[i];
                }
                funx=fun(x,a,n);
```

```
                }
        }
        printf("directional=%f,step=%f\n",fdd,lam);
        return (lam);
}



int main()
{
        /* BFGS with Armijo using equation (6.17) p.140 Nocedal & Wright*/
        int a,i,j,k,n,iter=0;
        printf("Choose the parameters a and n:\n");
        scanf("%d %d",&a,&n);
        double x[n],xk[n],dir[n],gradient[n],gradientk[n],fun_eval,H_bfgs[n][n],eye[n][n];
        double epsilon=pow(10,-5),norm=0,ssgrad=0,step,s[n],y[n],test;
        double rok,ssdir,syt[n][n],term1[n][n],yst[n][n],term3[n][n],sst[n][n];
        double term1H[n][n],mm[n][n];
        for (i=0;i<n;i++)
        {
                s[i]=0;
                y[i]=0;
                dir[i]=0;
                gradientk[i]=0;
                gradient[i]=0;
                for (j=0;j<n;j++)
                {
                        eye[i][j]=0;
```

```
                H_bfgs[i][j]=0;
        }
}
for (i=0;i<n;i++)
{
        eye[i][i]=1;
}
for (i=0;i<n;i=i+2)
{
        xk[i]=-1;
        x[i]=-1;
}
for (i=1;i<n;i=i+2)
{
        xk[i]=-1;
        x[i]=-1;
}
/* Take the initial H_bfgs to be identity */
for (i=0;i<n;i++)
{
        H_bfgs[i][i]=1;
}
printf("\nInitialization\n");
printf("f(x)=%f\n",fun(x,a,n));
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
```

```
		{
			printf("%f\t",H_bfgs[i][j]);
		}
		printf("\n");
}
fun_eval=fun(x,a,n);
for (i=0;i<n;i=i+2)
{
		gradient[i]=subgradient_odd(x[i],x[i+1],a);
		gradientk[i]=gradient[i];
}
for (i=1;i<n;i=i+2)
{
		gradient[i]=subgradient_even(x[i-1],x[i],a);
		gradientk[i]=gradient[i];
}
for (i=0;i<n;i++)
{
		printf("gradient%d=%f\n",i+1,gradient[i]);
}
printf("\n");
for (i=0;i<n;i++)
{
		ssgrad=ssgrad+pow(gradient[i],2);
}
norm=sqrt(ssgrad);
while (norm>=epsilon)
```

```
{
        /* Find initial direction */
        for (i=0;i<n;i++)
        {
                dir[i]=0;
                for (j=0;j<n;j++)
                {
                        dir[i]=dir[i]-H_bfgs[i][j]*gradient[j];
                }
        }
        ssdir=0;
        for (i=0;i<n;i++)
        {
                ssdir=ssdir+pow(dir[i],2);
        }
        ssdir=sqrt(ssdir);
        for (i=0;i<n;i++)
        {
                dir[i]=dir[i]/ssdir;
        }
        for (i=0;i<n;i++)
        {
                printf("dir%d=%f\n",i+1,dir[i]);
        }
        step=armijo(x,dir,gradient,a,n);
        /* Initial update of points */
        for (i=0;i<n;i++)
```

```
{
        x[i]=xk[i];
        x[i]=x[i]+step*dir[i];
}
/* Define sk and yk and check if their inner product is positive */
for (i=0;i<n;i=i+2)
{
        gradient[i]=subgradient_odd(x[i],x[i+1],a);
}
for (i=1;i<n;i=i+2)
{
        gradient[i]=subgradient_even(x[i-1],x[i],a);
}
for (i=0;i<n;i++)
{
        s[i]=x[i]-xk[i];
        y[i]=gradient[i]-gradientk[i];
}
test=0;
for (i=0;i<n;i++)
{
        test=test+s[i]*y[i];
}
/* BFGS Inverse Hessian Update */
if (test>0)
{
        /* Create rok */
```

```
rok=1/test;
/* Create sk*yk(transpose) */
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                syt[i][j]=s[i]*y[j];
        }
}
/* Create 1st term of the matrix multiplication */
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                term1[i][j]=eye[i][j]-rok*syt[i][j];
        }
}
/* Create yk*sk(transpose) */
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                yst[i][j]=y[i]*s[j];
        }
}
/* Create 3rd term of the matrix multiplication */
for (i=0;i<n;i++)
```

```
{
        for (j=0;j<n;j++)
        {
                term3[i][j]=eye[i][j]-rok*yst[i][j];
        }
}
/* Create sk*sk(transpose) */
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                sst[i][j]=s[i]*s[j];
        }
}
/* Create term1*Hk */
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                term1H[i][j]=0;
        }
}
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                for (k=0;k<n;k++)
```

```
                {
                        term1H[i][j]=term1H[i][j]+term1[i][k]*H_bfgs[k][j];
                }
        }
}
/* Create term1*Hk*term3 */
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                mm[i][j]=0;
        }
}
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                for (k=0;k<n;k++)
                {
                        mm[i][j]=mm[i][j]+term1H[i][k]*term3[k][j];
                }
        }
}
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
```

```
                                H_bfgs[i][j]=mm[i][j]+rok*sst[i][j];
                        }
                }
        }
        else
        {
                /* if sk(transpose)*yk<=0 then do not do the update! */
                for (i=0;i<n;i++)
                {
                        for (j=0;j<n;j++)
                        {
                                H_bfgs[i][j]=H_bfgs[i][j];
                        }
                }
        }
        for (i=0;i<n;i++)
        {
                xk[i]=x[i];
                printf("xk%d=%f\n",i+1,xk[i]);
                gradientk[i]=gradient[i];
        }
        ssgrad=0;
        for (i=0;i<n;i++)
        {
                ssgrad=ssgrad+pow(gradient[i],2);
        }
        fun_eval=fun(x,a,n);
```

```
            printf("f(x)=%f\n",fun_eval);
            norm=sqrt(ssgrad);
            iter=iter+1;
            printf("Iteration %d finished!\n\n",iter);
        }
        printf("\tAfter %d iter, we get:\n",iter);
        for (i=0;i<n;i++)
        {
            printf("\tx%d,opt=%2.5f\n",i+1,xk[i]);
        }
        printf("\tf(xopt)=%3.8f\n",fun(xk,a,n));
        return 0;
}
```