# MANE 6710 - Numerical Design Optimization Lab 5

Human 6966

December 18 2024

# Table of Contents:

## Executive Summery

Unconstrained optimization algorithms are powerful tools for engineers, as they form a basic method of solving complex multivariable optimization problems found across engineering disciplines. The purpose of this lab was for us to implement an unconstrained optimization problem from scratch to learn more about how they work. The algorithm I implemented was the Broyden–Fletcher–Goldfarb–Shanno algorithm, which is a nonlinear multivariable Quasi-Newton algorithm, that is relatively efficient for medium to large smooth optimization problems. I was successfully able to implement this algorithm in Python, which runs with reasonable efficiency and accuracy.

## 1 Introduction

Optimization algorithms are an important tool for engineers as they enable us to numerically determine a viable solution to a complex problem that balances various design criteria (such as weight, cost, or strength). Optimization algorithms solve these problems by numerically solving a given input equation set (called the objective function) for a local minimum. These algorithms use various methods to solve for these local minimums, which have different strengths and weaknesses.

For example, genetic algorithms can solve nonlinear equations that have a discontinuous first derivative as it rellies solly on the objective function. However, the downside of this paticular method is its computational cost as sampling the objective function several times each iteration to determine the best movement diretion takes more time and power than other methods like the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS).

The BFGS algorithm is a non-linear Quasi-Newton optimization algorithm that is significantly more efficient than the genetic algorithm. This is because it uses derivative and approximated second derivative information to reduce the number of function calls and iterations used to determine a local minimum. However, the downside to using this and other Quasi-Newton methods is they rely on the function having a continuous, easily computed first derivative. This means that there are problems the genetic algorithm is capable of solving that the BFGS algorithm cannot.

## 2 Methodology

The BFGS algorithm I implemented in this project is from algorithms 6.1, 3.5, and 3.6 [1] shown in Figures 1, 2, and 3 respectively.

My implementation of this algorithm works by using the complex step method to calculate the gradient ($\nabla f_k$) at the current position ($x_k$) using a step size of $10^{-30}$. Then my code calculates the step direction ($p_k$) from the gradient and the current approximation of the hessian ($H_k$) using the equation $p_k = -H_k \nabla f_k$. From this and the given objective function ($f(x)$), an anonymous scalar function is defined as $\phi(\alpha) = f(x_k + \alpha p_k)$ and an anonymous function for the derivative is defined as $\phi'(\alpha) = \nabla \phi(\alpha)$, which are given to the line search algorithm to solve for a minimum in the search direction.

The Line Search algorithm iterates through values of $\alpha$ until it either finds values on both sides of the directional minimum, finds an increasingly positive first derivative, or runs out of allowed iterations. In the first case, the line search method calls the Zoom algorithm, passing it the two endpoints and the scalar functions, and returns its output. In the second case, the function returns the current value of $\alpha$ and in the third case the algorithm throws a runtime error.

The Zoom algorithm performs a bisection search on the given interval of the scalar function, checking if the current $\alpha$ satisfies the strong Wolfe conditions for each iteration. When the strong Wolfe conditions are satisfied, the algorithm returns the current step length ($\alpha_{\text{current step}} = 0.5(\alpha_{\text{high bound}} + \alpha_{\text{low bound}})$). In iterations where the strong Wolf conditions aren't met, the algorithm sets the high bound of the interval to the current step if the sufficient decrease condition ($\phi(\alpha_{\text{current step}}) < \phi(0) +$

**Algorithm 6.1** (BFGS Method).

Given starting point $x_0$, convergence tolerance $\epsilon > 0$,
  inverse Hessian approximation $H_0$;
$k \leftarrow 0$;
**while** $\|\nabla f_k\| > \epsilon$;
  Compute search direction

$$p_k = -H_k \nabla f_k; \tag{6.18}$$

  Set $x_{k+1} = x_k + \alpha_k p_k$ where $\alpha_k$ is computed from a line search
    procedure to satisfy the Wolfe conditions (3.6);
  Define $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$;
  Compute $H_{k+1}$ by means of (6.17);
  $k \leftarrow k + 1$;
**end (while)**

Figure 1: Algorithm 6.1: BFGS Method

**Algorithm 3.5** (Line Search Algorithm).

Set $\alpha_0 \leftarrow 0$, choose $\alpha_{\max} > 0$ and $\alpha_1 \in (0, \alpha_{\max})$;
$i \leftarrow 1$;
**repeat**
  Evaluate $\phi(\alpha_i)$;
  **if** $\phi(\alpha_i) > \phi(0) + c_1 \alpha_i \phi'(0)$ or $[\phi(\alpha_i) \geq \phi(\alpha_{i-1})$ and $i > 1]$
    $\alpha_* \leftarrow$**zoom**$(\alpha_{i-1}, \alpha_i)$ and **stop;**
  Evaluate $\phi'(\alpha_i)$;
  **if** $|\phi'(\alpha_i)| \leq -c_2 \phi'(0)$
    **set** $\alpha_* \leftarrow \alpha_i$ and **stop;**
  **if** $\phi'(\alpha_i) \geq 0$
    **set** $\alpha_* \leftarrow$**zoom**$(\alpha_i, \alpha_{i-1})$ and **stop;**
  Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{\max})$;
  $i \leftarrow i + 1$;
**end (repeat)**

Figure 2: Algorithm 3.5: BFGS Line Search Method

$c_1 \alpha_{\text{current step}} \phi'(0))$ isn't satisfied and the low bound of the interval to the current step if the sufficient decrease condition is met but the flatness condition $(|\phi'(.\alpha_{\text{current step}})| \leq -c_2 \phi'(0))$. Note: $c_1$ and $c_2$ were chosen to be 0.0001 and 0.9 respectively based on the advice from the textbook [1].

Once the Zoom and Line Search algorithms return the current step length ($\alpha$), the BFGS algorithm updates the current position, gradient, and Hessian estimate. The Hessian estimate update equation is given in Equation 1 [1] ( $y_k$, and $s_k$, and $\rho_k$ are given in Equations 2, 3, and 4 respectively). Then the current step statistics are printed to the terminal and the ending condition $(||\nabla f(x_k)|| <$ Error Tolerance) is checked to determine if the algorithm needs another iteration and, if not, returns the final position.

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \tag{1}$$

$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k) \tag{2}$$

**Algorithm 3.6** (zoom).

**repeat**

  Interpolate (using quadratic, cubic, or bisection) to find
    a trial step length $\alpha_j$ between $\alpha_{\mathrm{lo}}$ and $\alpha_{\mathrm{hi}}$;
  Evaluate $\phi(\alpha_j)$;
  **if** $\phi(\alpha_j) > \phi(0) + c_1\alpha_j\phi'(0)$ or $\phi(\alpha_j) \geq \phi(\alpha_{\mathrm{lo}})$
    $\alpha_{\mathrm{hi}} \leftarrow \alpha_j$;
  **else**
    Evaluate $\phi'(\alpha_j)$;
    **if** $|\phi'(\alpha_j)| \leq -c_2\phi'(0)$
      Set $\alpha_* \leftarrow \alpha_j$ and **stop**;
    **if** $\phi'(\alpha_j)(\alpha_{\mathrm{hi}} - \alpha_{\mathrm{lo}}) \geq 0$
      $\alpha_{\mathrm{hi}} \leftarrow \alpha_{\mathrm{lo}}$;
    $\alpha_{\mathrm{lo}} \leftarrow \alpha_j$;

**end (repeat)**

Figure 3: Algorithm 3.6: BFGS Zoom Method

$$s_k = \alpha p_k \tag{3}$$

$$\rho_k = \frac{1}{y_k^T s_k} \tag{4}$$

## 3 Testing and Results

Once I implemented the algorithm in Python, the algorithm needed to be tested to ensure it worked. The test cases and the algorithm's results are summarized in Table 1. I chose a scalar function for the first test case, as it is a special case that isn't affected by matrix algebra errors while still testing the different algorithms and their integration. Once I ensured the algorithm was running as intended with the scalar case, I chose to implement two simple two-dimensional polynomial cases to test the matrix algebra implementation and how the code handles vectors. There were several matrix algebra errors due to how the Numpy library handles arrays that were fixed using these test cases. The final set of test cases I chose to test the algorithm's ability to handle more complex problems. The Two-dimensional Rosenbrock function was suggested by the instructor as a good single-objective optimization test case used in research. The fifth test case tested how the algorithms reacted to multimodal cases and the sixth tested the algorithm's ability to handle higher dimensional problems. See Table 2 for the initial conditions used in each test case.

As is shown in Table 1, the algorithm I implemented was able to consistently find a minimum within a reasonable time frame (the largest number of iterations used being 34 in the used test cases). While there is some error between the calculated and actual minima locations, particularly in cases with unequal scaling. As shown in the Appendix section for each test case, the First-order Optimality (magnitude of the gradient) decreased by at least nine orders of magnitude demonstrating a properly functioning optimization algorithm converging on a viable solution. Between the low error in the results and the convergence history, my implementation of the BFGS algorithm is shown to work in all of the test cases.

Table 1: Test Cases and Algorithm Results

| Test Case | Actual Minima Location | Calculated Minima Location | Apendix |
|---|---|---|---|
| $x^2$ | 0 | 0,0 | 5.1.1 |
| $x_1^4 + x_2^2$ | (0,0) | (1.28e-3,-2.10e-8),(1.28e-3,-2.10e-8) | 5.1.2 |
| $(x_1 - 1)^4 + (x_2 + 2)^2$ | (1,-2) | (0.999,-1.9999), (1.001,-1.99999) | 5.1.3 |
| two dimensional Rosenbrock | (1,1) | (1,1) | 5.1.4 |
| $x_1^4 + x_2^4 - 17x_2^3 + 45x_2^2$ | (1,3), (1,13.6342) | (1.0002,3),(0.9972,13.6342) | 5.1.5 |
| $(x_1 - 1)^4 + (x_2 + 2)^2 + 1 + 5(x_3 - 3)^4$ | (1,-2,3) | (0.9996,-2,3), (1.0015,-2,3) | 5.1.6 |

Table 2: Initial Conditions for Test Cases

| Test Case | $x_0$ | $H_0$ |
|---|---|---|
| $x^2$ | 10,-10 | $0.1I$ |
| $x_1^4 + x_2^2$ | (10,10),(-10,-10) | $0.1I$ |
| $(x_1 - 1)^4 + (x_2 + 2)^2$ | (4,4),(-7,-7) | $0.05I$ |
| two dimensional Rosenbrock | (1,1) | $0.1I$ |
| $x_1^4 + x_2^4 - 17x_2^3 + 45x_2^2$ | (4,4),(-7,-7) | $0.1I$ |
| $(x_1 - 1)^4 + (x_2 + 2)^2 + 1 + 5(x_3 - 3)^4$ | (4,4,4),(-7,-7,-7) | $0.1I$ |

# 4 Conclusions

I was able to successfully implement the nonlinear unconstrained optimization algorithm BFGS. As shown in the results, the implemented algorithm was able to handle simple and complex smooth nonlinear multivariable optimization problems efficiently. However, my implementation of the BFGS algorithm has significant limitations. Firstly, it only handles unconstrained optimization algorithms. This limits the potential applications for this algorithm as most problems in engineering have constraints. This can be somewhat mitigated by adding barrier functions to the objective function, however, it will likely be less efficient and accurate than an algorithm designed for handling constraints. Secondly, the algorithm only works with deterministic functions that are 'smooth' and have a continuous first derivative. Thirdly, this implementation is less efficient for linear optimization problems than purpose-made algorithms. Finally, this algorithm doesn't handle functions that go to negative infinity well (it usually throws an error), so it should only be used on functions with real minima.

# References


[1] S. W. Jorge Nocedal, *Numerical optimization second edition*, Book, 2006.

# 5 Appendix

## 5.1 Optimization Algorithm Test Case Output

### 5.1.1 One Dimentional Polynomial

```
Iterations:       |grad|
1                 16.0
2                 0.0
bfgs[0.]
Iterations:       |grad|
1                 16.0
2                 0.0
bfgs[0.]
```

Figure 4: Function output for one dimensional polynomial from two starting points

### 5.1.2 Two Dimentional Polynomial

```
Iterations:      |grad|
1                65.58403483318178
2                51.680716658179655
3                15.290466011992974
4                12.394032704204294
5                9.477798608410072
6                5.178661875102632
7                2.1773968474436325
8                0.8221514143987172
9                0.454889195063089
10               0.2857348166513772
11               0.13584582547710278
12               0.0464531951190135
13               0.023335755179815895
14               0.01747223023161133
15               0.008776985597580767
16               0.002147450337541808
17               0.0015736080969060552
18               0.0014467020121639686
19               0.0006259883329915933
20               7.49960770268807e-05
21               0.00016772994998276555
22               0.00013212992776993662
23               4.261952044649142e-05
24               8.982231749875275e-06
25               1.9101024019070267e-05
26               1.156345652089891e-05
27               2.318783372309847e-06
28               1.7494845845582582e-06
29               2.0258010729966285e-06
30               9.257729385853145e-07
31               4.274642526164209e-08
bfgs[-1.28237509e-03 -2.09529297e-08]
```

```
Iterations:      |grad|
1                65.58403483318178
2                51.680716658179655
3                15.290466011992974
4                12.394032704204294
5                9.477798608410072
6                5.178661875102632
7                2.1773968474436325
8                0.8221514143987172
9                0.454889195063089
10               0.2857348166513772
11               0.13584582547710278
12               0.0464531951190135
13               0.023335755179815895
14               0.01747223023161133
15               0.008776985597580767
16               0.002147450337541808
17               0.0015736080969060552
18               0.0014467020121639686
19               0.0006259883329915933
20               7.49960770268807e-05
21               0.00016772994998276555
22               0.00013212992776993662
23               4.261952044649142e-05
24               8.982231749875275e-06
25               1.9101024019070267e-05
26               1.156345652089891e-05
27               2.318783372309847e-06
28               1.7494845845582582e-06
29               2.0258010729966285e-06
30               9.257729385853145e-07
31               4.274642526164209e-08
bfgs[1.28237509e-03 2.09529297e-08]
```

Figure 5: Function output for two dimensional polynomial from two starting points

#### 5.1.3 Shifted Two Dimentional Polynomial

```
Iterations:      |grad|
1                56.340816607500486
2                8.975456017530629
3                6.806647524875654
4                0.14184863330781683
5                0.0009714744472591841
6                0.00013065326931259422
7                0.00015399075436447785
8                0.0005565454995007706
9                0.0005638851609230319
10               0.00024382693631504204
11               1.3841268955275838e-05
12               5.1579251356273843e-05
13               4.828845594752732e-05
14               1.8009211663722587e-05
15               1.1317081371780335e-06
16               5.779569120065623e-06
17               4.1401531711586605e-06
18               1.0917889818202174e-06
19               3.9756649753852955e-07
20               6.212506800217101e-07
21               3.2924237491447375e-07
22               4.199133178057899e-08
bfgs[ 0.9993471  -1.99999998]
Iterations:      |grad|
1                442.47820629834433
2                64.99383329095672
3                38.73384423230395
4                12.508293226775423
5                5.621957850248754
6                3.3311886758200204
7                2.5168181853149805
8                1.5894162989075442
9                0.7803443511994075
10               0.24323989043595762
11               0.0955112574814264
```

```
Iterations:      |grad|
1                442.47820629834433
2                64.99383329095672
3                38.73384423230395
4                12.508293226775423
5                5.621957850248754
6                3.3311886758200204
7                2.5168181853149805
8                1.5894162989075442
9                0.7803443511994075
10               0.24323989043595762
11               0.0955112574814264
12               0.0810742678993822
13               0.06356240622723838
14               0.033153143446662
15               0.007823257469121702
16               0.0048387151658952085
17               0.005123317104651325
18               0.0028512099436148177
19               0.0005466242011692326
20               0.00042434541697837887
21               0.0004814519071360643
22               0.00023268283105266655
23               2.3110256159098433e-05
24               5.053119100994786e-05
25               4.621663701041531e-05
26               1.7414490196882007e-05
27               1.3515939120023806e-06
28               6.051021320554987e-06
29               4.216913487507347e-06
30               1.1102194694811617e-06
31               4.4165021324450455e-07
32               6.709111904820717e-07
33               3.55566553371198e-07
34               4.674735595228893e-08
bfgs[ 1.00072178 -1.99999998]
```

Figure 6: Function output for two dimensional polynomial from two starting points

### 5.1.4 Two Dimentional Rosenbrock Function

| Iterations: | \|grad\| | Iterations: | \|grad\| |
|---|---|---|---|
| 1 | 65.58403483318178 | 1 | 65.58403483318178 |
| 2 | 51.680716658179655 | 2 | 51.680716658179655 |
| 3 | 15.290466011992974 | 3 | 15.290466011992974 |
| 4 | 12.394032704204294 | 4 | 12.394032704204294 |
| 5 | 9.477798608410072 | 5 | 9.477798608410072 |
| 6 | 5.178661875102632 | 6 | 5.178661875102632 |
| 7 | 2.1773968474436325 | 7 | 2.1773968474436325 |
| 8 | 0.8221514143987172 | 8 | 0.8221514143987172 |
| 9 | 0.454889195063089 | 9 | 0.454889195063089 |
| 10 | 0.2857348166513772 | 10 | 0.2857348166513772 |
| 11 | 0.13584582547710278 | 11 | 0.13584582547710278 |
| 12 | 0.0464531951190135 | 12 | 0.0464531951190135 |
| 13 | 0.023335755179815895 | 13 | 0.023335755179815895 |
| 14 | 0.01747223023161133 | 14 | 0.01747223023161133 |
| 15 | 0.008776985597580767 | 15 | 0.008776985597580767 |
| 16 | 0.002147450337541808 | 16 | 0.002147450337541808 |
| 17 | 0.0015736080969060552 | 17 | 0.0015736080969060552 |
| 18 | 0.0014467020121639686 | 18 | 0.0014467020121639686 |
| 19 | 0.0006259883329915933 | 19 | 0.0006259883329915933 |
| 20 | 7.49960770268807e-05 | 20 | 7.49960770268807e-05 |
| 21 | 0.00016772994998276555 | 21 | 0.00016772994998276555 |
| 22 | 0.00013212992776993662 | 22 | 0.00013212992776993662 |
| 23 | 4.261952044649142e-05 | 23 | 4.261952044649142e-05 |
| 24 | 8.982231749875275e-06 | 24 | 8.982231749875275e-06 |
| 25 | 1.9101024019070267e-05 | 25 | 1.9101024019070267e-05 |
| 26 | 1.156345652089891e-05 | 26 | 1.156345652089891e-05 |
| 27 | 2.318783372309847e-06 | 27 | 2.318783372309847e-06 |
| 28 | 1.7494845845582582e-06 | 28 | 1.7494845845582582e-06 |
| 29 | 2.0258010729966285e-06 | 29 | 2.0258010729966285e-06 |
| 30 | 9.257729385853145e-07 | 30 | 9.257729385853145e-07 |
| 31 | 4.274642526164209e-08 | 31 | 4.274642526164209e-08 |
| bfgs[-1.28237509e-03 -2.09529297e-08] | | bfgs[1.28237509e-03 2.09529297e-08] | |

Figure 7: Function output for two dimensional Rosenbrock Function from one starting point

### 5.1.5 Two Dimentional Polynomial with Multiple Minima

```
Iterations:      |grad|                          Iterations:      |grad|
1                7.039391029514307               1                593.3935263992481
2                0.12004658381253484             2                319.545937420049
3                0.08533043356255114             3                473.8574794033544
4                0.2120188355121932              4                418.86976790550654
5                0.3061268958883217              5                139.28932091766006
6                0.24986420826773392             6                66.1096109011661
7                0.084396360063551               7                30.966000339939256
8                0.008266180032129837            8                13.742972535708395
9                0.027488657118786147            9                4.511687908107179
10               0.022357723969299387            10               1.967816628197473
11               0.007168978343351536            11               0.8339069908811855
12               0.0011358951261645814           12               0.3584670399901734
13               0.0029053581940634977           13               0.15417390831930014
14               0.0019573841005107746           14               0.06631074550623645
15               0.000468341081580844            15               0.028524865867473246
16               0.0002137114339111993           16               0.01227008769633971
17               0.0003069119577110067           17               0.005278116286219785
18               0.00016073866662605544          18               0.0022704296340439224
19               1.972859591231157e-05           19               0.000976647846380209
20               3.1915001661787985e-05          20               0.0004201144698644053
21               3.0503928730416796e-05          21               0.00018071633162249212
22               1.1924910273030958e-05          22               7.773688236734789e-05
23               6.577248836736022e-07           23               3.34392750698766e-05
24               4.02999940591284e-06            24               1.4384228771538615e-05
25               2.800042786074979e-06           25               6.187515672800093e-06
26               7.406320693787607e-07           26               2.661619939565364e-06
27               3.02775543023223e-07            27               1.1449216589576383e-06
28               4.508901751938955e-07           28               4.92499167601005e-07
29               2.3417408605071746e-07          29               2.118532986312597e-07
30               2.8304108422669455e-08          30               9.113075333215174e-08
bfgs[1.00025018 3.        ]                      bfgs[ 0.99716512 13.63418126]
```

Figure 8: Function output for two dimensional polynomial from two starting points

### 5.1.6 Three Dimentional Polynomial

```
Iterations:      |grad|
1                65.58403483318178
2                51.680716658179655
3                15.290466011992974
4                12.394032704204294
5                9.477798608410072
6                5.178661875102632
7                2.1773968474436325
8                0.8221514143987172
9                0.454889195063089
10               0.2857348166513772
11               0.13584582547710278
12               0.0464531951190135
13               0.023335755179815895
14               0.01747223023161133
15               0.008776985597580767
16               0.002147450337541808
17               0.0015736080969060552
18               0.0014467020121639686
19               0.0006259883329915933
20               7.49960770268807e-05
21               0.00016772994998276555
22               0.00013212992776993662
23               4.261952044649142e-05
24               8.982231749875275e-06
25               1.9101024019070267e-05
26               1.156345652089891e-05
27               2.318783372309847e-06
28               1.7494845845582582e-06
29               2.0258010729966285e-06
30               9.257729385853145e-07
31               4.274642526164209e-08
bfgs[-1.28237509e-03 -2.09529297e-08]
```

```
Iterations:      |grad|
1                65.58403483318178
2                51.680716658179655
3                15.290466011992974
4                12.394032704204294
5                9.477798608410072
6                5.178661875102632
7                2.1773968474436325
8                0.8221514143987172
9                0.454889195063089
10               0.2857348166513772
11               0.13584582547710278
12               0.0464531951190135
13               0.023335755179815895
14               0.01747223023161133
15               0.008776985597580767
16               0.002147450337541808
17               0.0015736080969060552
18               0.0014467020121639686
19               0.0006259883329915933
20               7.49960770268807e-05
21               0.00016772994998276555
22               0.00013212992776993662
23               4.261952044649142e-05
24               8.982231749875275e-06
25               1.9101024019070267e-05
26               1.156345652089891e-05
27               2.318783372309847e-06
28               1.7494845845582582e-06
29               2.0258010729966285e-06
30               9.257729385853145e-07
31               4.274642526164209e-08
bfgs[1.28237509e-03 2.09529297e-08]
```

Figure 9: Function output for three dimensional polynomial from two starting points

### 5.2 Code

#### 5.2.1 BFGS Algorithm

```
from lineSearch import lineSearch
import numpy as np

def bfgs (funcs, x0,H0,e, maxIterations = 1000):

        # initialize variables
        h=np.array(H0);
        xk=np.array(x0);
        dim=max(np.shape(H0));
        fk=grad(funcs,xk, isfk=1);
        count=0;

        # print out headers
        print("Iterations:\t|grad|")

        # loop through until the norm of the gradient is less than e or the maximum number of iterations is reached
        while count<maxIterations and np.linalg.norm(fk)>e:
                #calculate search direction
                pk=np.array(1);
                if dim==1:
                        pk=np.array(-1*h*fk);
                        pk=pk[:,0];

                else:
                        pk=np.zeros(dim);
                        c=0;
                        for i in -1*np.matmul(h,fk):
                                pk[c]=i[0];
                                c+=1;

                #create scalar function in direction pk
                phi=lambda a:funcs(xk+a*pk)[0];

                #create anonomous function for getting the derivative
                der=lambda a: phi(complex(a,0.000000000000000000000000000001))).imag/0.000000000000000000000000000001

                #compute line search
                a=lineSearch(1,phi,der,maxIterations=1000000);

                #calc xk+1
                xk1=xk+a*pk;

                #calc sk
                sk=a*pk;
```

```
        #calc yk
        yk=grad(funcs,xk1)-grad(funcs,xk);
        #calc rho k
        rhok=1/((yk)@cols(sk))[0]

        #calc Hk+1
        #eqn 6.17
        if dim==1:
            h=((np.eye(dim)-rhok*(sk*np.transpose(yk))))*h*(np.eye(dim)-rhok*(yk*np.transpose(sk)))+rhok*(sk*np.transpose(sk));
        else:
            a1=rhok*mult(cols(sk),sk)
            # print(a1)
            a21=np.eye(dim)-rhok*mult(cols(sk),yk)
            a22=np.eye(dim)-rhok*mult(cols(yk),(sk))
            a2=a21@(h@a22)
            h=a2+a1;

        # iterate count
        count +=1;

        # update gradient
        fk=grad(funcs,xk1, isfk=1);

        # update position
        xk=xk1;

        # print step info
        print(str(count)+'\t\t'+str(np.linalg.norm(fk)))

    # print warning if maximum number of iterations are reached
    if count>=maxIterations:
        print("warning max maxIterations reached");

    # return final position
    return xk



def grad(func, x,step=0.000000000000000000000000000001,isfk=0):
    """use complex step to calculate function gradient

    Inputs:
        func - the function to calculate gradient of
        x - the point to evaluate at
        step - (optional) the imaginary step size to use
    Output:
        the complex step approximation of the gradient"""

```

```
        # initialize complex inputs
        y=np.zeros(np.shape(x), dtype=complex)
        for i in range(np.size(y)):
                y[i]=x[i];

        #       initialize outputs
        z=np.zeros(np.shape(y))

# iterate through the input directions and get derivative in each direction using complex step
        for i in range(np.size(z)):
                # keep indicies in correct bounds
                if i==np.size(x):
                        break;
                # add complex step to correct input direction
                y[i] =complex(x[i],step)
                # calculate derivative in that direction
                z[i]=func(y)[0].imag/step;
                # remove the complex step from the variable
                y[i] =x[i]
        # convert to column vector if needed
        if isfk:
                z=cols(z)
# return output
        return z

def cols(x):
        """define a row numpy array as a column matrix. Note this was implamented due to native python methods/python libraries acting oddly

        Inputs:
                x - the array to transpose
        Output:
                z - the transposed array"""
        y=np.zeros((np.size(x),1));
        cont=0;
        for i in x:
                y[cont,0]=i;
                cont+=1;
        return y;
def mult(x,y):
        """define a row numpy array as a column matrix. Note this was implamented due to native python methods/python libraries acting oddly

        Inputs:
                x - the first array to multiply
                y - the second array to multiply
        Output:
```

```
                z - the matrix resulting from the multiplication"""
        z=np.zeros((np.shape(x)[0],np.shape(x)[0]))
        for i in range(np.size(x)):
                for j in range(np.size(x)):
                        z[i,j]=x[i][0]*y[j]
                # print(z)
        return z

#-------------- test casses -----------------------

# one dimentional quadratic
func=lambda x: np.array(x*x);
h0=np.array(np.eye(1))*.1
x0=10*np.ones(1);
print("bfgs"+str(bfgs(func,x0,h0,.000001)))
x0=-10*np.ones(1);
print("bfgs"+str(bfgs(func,x0,h0,.000001)))

# two dimensional polynomial
func = lambda x: np.array([x[0]*x[0]*x[0]*x[0]+ x[1]*x[1]])
h0=np.array(np.eye(2))*.1;
x0=10*np.ones(2);
print("bfgs"+str(bfgs(func,x0,h0,.0000001)))
x0=-10*np.ones(2);
print("bfgs"+str(bfgs(func,x0,h0,.0000001)))

#two dimensional polynomial non-zeros minimum/position
func = lambda x: np.array([(x[0]-1)*(x[0]-1)*(x[0]-1)*(x[0]-1)+ (x[1]+2)*(x[1]+2)+1])
h0=np.array(np.eye(2))*.1;
x0=4*np.ones(2);
print("bfgs"+str(bfgs(func,x0,h0,.0000001)))
x0=-7*np.ones(2);
print("bfgs"+str(bfgs(func,x0,h0,.0000001)))

# two dimensional rosenbrock function
func = lambda x: np.array([(((1-x[0])*(1-x[0]))+ 100*((x[1]-(x[0]*x[0]))*(x[1]-(x[0]*x[0]))))])
h0=np.array(np.eye(2))*.05;
x0=0*np.ones(2);
print("bfgs"+str(bfgs(func,x0,h0,.0000001)))

# two dimensional polynomial, multiple minima
func = lambda x: np.array([(x[0]-1)*(x[0]-1)*(x[0]-1)*(x[0]-1)+ 45*(x[1]-3)*(x[1]-3)+(x[1]-3)*(x[1]-3)*(x[1]-3)*(x[1]-3)-17*(x[1]-3)*(x[1]-3)*(x[1]-3)])
h0=np.array(np.eye(2))*.1;
x0=4*np.ones(2);
print("bfgs"+str(bfgs(func,x0,h0,.0000001)))
x0=-7*np.ones(2);
```

```
print("bfgs"+str(bfgs(func,x0,h0,.0000001)))

#three dimensional polynomial

func = lambda x: np.array([(x[0]-1)*(x[0]-1)*(x[0]-1)*(x[0]-1)+ (x[1]+2)*(
    x[1]+2)+1+5*(x[2]-3)*(x[2]-3)+(x[2]-3)*(x[2]-3)*(x[2]-3)*(x[2]-3)])
h0=np.array(np.eye(3))*.1;
x0=4*np.ones(3);
print("bfgs"+str(bfgs(func,x0,h0,.0000001)))
x0=-7*np.ones(3);
print("bfgs"+str(bfgs(func,x0,h0,.0000001)))
```

### 5.2.2 Line Search Algorithm

```python
from zoom import zoom;
import numpy as np;

def lineSearch(amax,phi,phiprime,c1=0.0001,c2=.9,maxIterations=100000):

        ai=1.0
        # print(amax)
        # initialize variables
        ai2= 0.0;
        count=0;
        # print("eval0: "+str(phi(0)))
        # print("deval0: "+str(phiprime(0)))
        # print("eval1: "+str(phi(ai)))
        # print("deval1: "+str(phiprime(ai)))
        while count<maxIterations:

                # print(str(c1*phiprime(0)));
                # check sufficient decrease
                if (phi(ai)>(phi(0)+c1*ai*phiprime(0))) or ((phi(ai)>=phi(ai2)) and (count>0)):
                        # print("zoooominggg"+str(count))
                                # call zoom and return value
                        return zoom(ai2,ai,phi,phiprime,c1,c2);
                        #check curavture condition
                # print(abs(phiprime(ai)))
                # print(-1*c2*phiprime(0))
                # break
                if (abs(phiprime(ai))<=(-1*c2*phiprime(0))):
                        # print("succssesss")
                        #return current value
                        return ai;
                        #if phi increasing
                if phiprime(ai)>=0:
                        #call zoom and return
                        return zoom(ai, ai2,phi,phiprime,c1,c2);
                #reset ai2
                ai2=ai;
                count +=1;

                # interpolate ai
                ai*=1.1;

                # if viable point not found print warning and return Null
        print("Warning: max number of itterations reached"+ai);
        return None;
```

### 5.2.3 Zoom algorithm

```python
import numpy as np;


def zoom(a0,ahi,phi, phiPrime, c1, c2, maxIterations=100000):

        # print(a0)
        # print(ahi)
        alo=a0;
        count=0;
        aj=0;
        # print("eval0: "+str(phi(0)))
        # print("deval0: "+str(phiPrime(0)))
        # print("eval1: "+str(phi(ahi)))
        # print("deval1: "+str(phiPrime(ahi)))
        # while number of itterations < some max number (to prevent infinite loops)
        while count<maxIterations:
                count+=1;
                # interpolate aj
                aj=(ahi+alo)/2
                # print(ahi)
                # print(aj)
                # print(alo)
                # print("evalj: "+str(phi(aj)))
                # print("devalj: "+str(phiPrime(aj)))
                # check sufficient decrease condition
                if(( phi(aj)>phi(0)+c1*(aj)*phiPrime(0) )or (phi(aj)>= phi(alo))):
                        # if not viable move upper bound
                        ahi=aj;

                else:
                        # evaluate flatness condition
                        if abs(phiPrime(aj))<=-c2*phiPrime(0):
                                # if flat enough return aj
                                return aj;
                        if phiPrime(aj)*(ahi-alo)>=0:
                                # enforce phi(ahi)>phi(alo)
                                ahi=alo;
                                # print(ahi)
                        # if sufficient decrease satisfied but not flatness, update alo
                        alo=aj;
                        # print(alo)
                # if count>10:
                #       return
        # if viable point not found print warning and return Null
        print("Warning: max number of itterations reached -z"+str(aj));
```

```
        return a0;
```