1. Implementation

(a) Steepest Descent Method

i. Three control parameters are set as $\alpha = 1$, $\beta = 2$, $\gamma = 0.5$

```
#ifndef __CAUCHYS__
    #define __CAUCHYS__
2
3
    #include "multivariate.h"
4
   #include "multi/termination.hpp"
5
6
   namespace numerical_optimization {
   template<typename VectorTf>
9
   class Cauchys : public Multivariate<VectorTf> {
10
   public:
11
        using Base = Multivariate<VectorTf>;
12
        using Base::Base;
13
       using Base::plot;
14
        using Base::function;
15
        using Base::gradient;
16
        using function_t = typename Base::function_t;
17
18
        // constructors
19
        Cauchys(function_t f):Base(f){};
20
21
        // generally works
22
       VectorTf eval(const VectorTf% init=VectorTf::Random(), float e=epsilon)
23
    → override {
            // 1. initialize
24
            VectorTf xi = init;
25
            // 2. loop
26
            for(size_t i=0; i<this->iter; i++) {
27
    #ifdef BUILD_WITH_PLOTTING
28
                plot.emplace_back(std::make_pair(xi, function(xi)));
29
    #endif
30
                // 1. termination
31
                if(terminate<Termination::Condition::MagnitudeGradient>({xi}, e))
32
    → break;
33
                // 2. the steepest descent direction
34
                VectorTf p = -1*gradient(xi)/gradient(xi).norm();
35
36
                // 3. step length
37
```

Homework 3

```
float alpha = this->line_search_inexact(xi, p);
38
39
             // 4. update gradient
40
             xi = xi + alpha*p;
41
         }
42
         return xi;
43
      };
44
45
      // termination
46
      template<Termination::Condition CType>
47
      bool terminate(const std::vector<VectorTf>& x, float h=epsilon) const {
48
         return Termination::eval<VectorTf, CType>(function, x, h);
49
      }
50
  };
51
  52
  }/// the end of namespace numerical_optimization ///
53
  54
  #endif //__CAUCHYS__
55
```

October 30, 2021

(b) Gradient and Hessian

i. Three control parameters are set as $\alpha = 1$, $\beta = 2$, $\gamma = 0.5$

```
#include <cmath>
  1
                #include <iostream>
  2
  3
               #include "multivariate.h"
  4
              using namespace Eigen;
  6
  7
              namespace numerical_optimization {
  8
  9
              // two-variables case
10
               // specialization of the vector2f case
11
               //
12
                \rightarrow https://stackoverflow.com/questions/38854363/is-there-any-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-negative-standard-way-to-calculate-the-neg
              template<>
13
              Vector2f _gradient<Vector2f>(const std::function<float(const Vector2f&)>& f,
14
                 using v2 = Vector2f;
15
16
                              float dx = 3*f(v2(x[0]-4*h, x[1]))-32*f(v2(x[0]-3*h,
17
                 \rightarrow x[1])+168*f(v2(x[0]-2*h, x[1]))-672*f(v2(x[0]-h, x[1]))
                                                                        -3*f(v2(x[0]+4*h, x[1]))+32*f(v2(x[0]+3*h,
18
                 \rightarrow x[1]))-168*f(v2(x[0]+2*h, x[1]))+672*f(v2(x[0]+h, x[1]));
                               float dy = 3*f(v2(x[0], x[1]-4*h))-32*f(v2(x[0], x[1]-3*h))+168*f(v2(x[0], x[0], x[0],
19
                 \rightarrow x[1]-2*h))-672*f(v2(x[0], x[1]-h))
                                                                         -3*f(v2(x[0], x[1]+4*h))+32*f(v2(x[0], x[1]+3*h))-168*f(v2(x[0], x[1]+3*h))
20
                 \rightarrow x[1]+2*h))+672*f(v2(x[0], x[1]+h));
^{21}
                              float inv = (1/(h*840));
22
                              return v2(dx, dy)*inv;
23
              }
24
25
               // specialization of the vector2f case
26
27
              Matrix2f _hessian<Vector2f>(const std::function<float(const Vector2f&)>& f,
28
                 → const Vector2f& x, float h) {
                              using vec2 = Vector2f;
29
30
                              h=0.01;
31
                               auto dfdx = [\&](vec2 x){
32
                                               float inv = (1/(h*840));
33
                                                float app = 3*f(vec2(x[0]-4*h, x[1]))-32*f(vec2(x[0]-3*h,
34
                              x[1])+168*f(vec2(x[0]-2*h, x[1]))-672*f(vec2(x[0]-h, x[1]))
```

```
-3*f(vec2(x[0]+4*h, x[1]))+32*f(vec2(x[0]+3*h,
35
       x[1])-168*f(vec2(x[0]+2*h, x[1]))+672*f(vec2(x[0]+h, x[1]));
                return app*inv;
36
           };
37
38
       auto dfdy = [\&](vec2 x){
39
           float inv = (1/(h*840));
40
            float app = 3*f(vec2(x[0], x[1]-4*h))-32*f(vec2(x[0],
41
       x[1]-3*h)+168*f(vec2(x[0], x[1]-2*h))-672*f(vec2(x[0], x[1]-h))
                        -3*f(vec2(x[0], x[1]+4*h))+32*f(vec2(x[0],
42
       x[1]+3*h)-168*f(vec2(x[0], x[1]+2*h))+672*f(vec2(x[0], x[1]+h));
           return app*inv;
43
           };
44
45
       float dxx = f(vec2(x[0]+2*h, x[1]))-2*f(vec2(x[0], x[1]))+f(vec2(x[0]-2*h, x[1]))
46
    \rightarrow x[1]));
       float dxy = f(vec2(x[0]+h, x[1]+h))-f(vec2(x[0]-h, x[1]+h))-f(vec2(x[0]+h, x[1]+h))
47
    \rightarrow x[1]-h)) + f(vec2(x[0]-h, x[1]-h));
       float dyx = f(vec2(x[0]+h, x[1]+h))-f(vec2(x[0]+h, x[1]-h))-f(vec2(x[0]-h, x[1]-h))
48
    \rightarrow x[1]+h)) + f(vec2(x[0]-h, x[1]-h));
       float dyy = f(\text{vec2}(x[0], x[1]+2*h))-2*f(\text{vec2}(x[0], x[1]))+f(\text{vec2}(x[0], x[1]))
49
    \rightarrow x[1]-2*h));
50
       Matrix2f m;
51
       m << dxx, dxy, dyx, dyy;
52
       float inv = 1/(4*h*h);
53
       return m*= inv;
54
   }
55
   56
   } /// the end of namespace numerical_optimization ///
57
```

October 30, 2021

Homework 3

(c) Newton's method i.

```
#ifndef __NEWTONS__
    #define __NEWTONS__
2
3
    #include "multivariate.h"
4
    #include "multi/termination.hpp"
5
   namespace numerical_optimization {
7
8
   template<typename VectorTf>
9
   class Newtons : public Multivariate<VectorTf> {
10
   public:
11
       using Base = Multivariate<VectorTf>;
12
       using Base::Base;
13
       using Base::plot;
14
        using Base::function;
15
        using Base::gradient;
16
        using Base::hessian;
17
        using function_t = typename Base::function_t;
18
19
       // constructors
20
        template<Termination::Condition CType>
21
       bool terminate(const std::vector<VectorTf>& x, float h=epsilon) const {
22
            return Termination::eval<VectorTf, CType>(function, x, h);
23
       }
24
25
        // generally works
26
       VectorTf eval(const VectorTf% init=VectorTf::Random(), float e=epsilon)
27
    → override {
            VectorTf xi = init;
28
            for(size_t i=0; i<this->iter; i++) {
29
    #ifdef BUILD_WITH_PLOTTING
30
                plot.emplace_back(std::make_pair(xi, function(xi)));
31
    #endif
32
                // 1. termination
33
                if(terminate<Termination::Condition::MagnitudeGradient>({xi}, e))
34
    → break;
35
                // 2. gradient update
36
                xi = xi - hessian(xi).inverse()*gradient(xi);
37
            }
38
            return xi;
39
```

20211046 Hyeonjang An

Homework 3

Numerical Opimization October 30, 2021

October 30, 2021

Homework 3

$(d) \ \, \mathbf{Quasi\text{-}Newton's} \ \, \mathbf{method}$

```
#ifndef __QUASI_NEWTONS__
    #define __QUASI_NEWTONS__
2
3
   #include <math.h>
4
   #include <cassert>
5
    #include "multivariate.h"
   #include "multi/termination.hpp"
8
   namespace numerical_optimization {
9
   namespace quasi_newtons {
10
   enum Rank { SR1, BFGS, };
11
   };
12
13
   template<typename VectorTf, quasi_newtons::Rank RankMethod>
14
   class QuasiNewtons : public Multivariate<VectorTf> {
15
   public:
16
        using Base = Multivariate<VectorTf>;
17
        using Base::Base;
18
       using Base::plot;
19
       using Base::iter;
20
       using Base::function;
21
        using Base::gradient;
22
       using function_t = typename Base::function_t;
23
       using MatrixTf = Eigen::Matrix<typename VectorTf::Scalar,</pre>
24
    → VectorTf::RowsAtCompileTime, VectorTf::RowsAtCompileTime>;
25
        template<Termination::Condition CType>
26
        bool terminate(const std::vector<VectorTf>& x, float h, float eps=epsilon) {
27
            return Termination::eval<VectorTf, CType>(function, x, h, eps);
28
29
        VectorTf eval(const VectorTf& init=VectorTf::Random(), float e=epsilon)
30
    → override {
31
            VectorTf xi = init;
32
            MatrixTf Hk = MatrixTf::Identity();
33
            size_t iteration = 0;
35
            for(size_t i=0; i<this->iter; i++) {
36
   #ifdef BUILD_WITH_PLOTTING
37
                plot.emplace_back(std::make_pair(xi, function(xi)));
38
   #endif
```

```
// @@todo other termination method
40
                if(terminate<Termination::Condition::MagnitudeGradient
41
                 |Termination::Condition::FunctionValueDifferenceRelative>({xi},
42
        1e-5)) {
                     break;
43
                     }
44
45
                // Compute a Search Direction
46
                VectorTf p = -1 * Hk*gradient(xi);
47
48
                // Compute a step length Wolfe Condition
49
                // float alpha = this->line_search_inexact(xi, p, 0.99, 0.5);
50
51
                // Compute a step length exactly
                float alpha = this->line_search_exact(xi, p);
53
54
                // Define sk and yk
55
                VectorTf Sk = alpha*p;
56
                VectorTf yk = gradient(xi+Sk) - gradient(xi);
57
58
                // Compute Hk+1
59
                if constexpr (RankMethod==quasi_newtons::Rank::SR1)
                     Hk = SR1(Hk, Sk, yk);
61
                else if constexpr (RankMethod==quasi_newtons::Rank::BFGS)
62
                     Hk = BFGS(Hk, Sk, yk);
63
64
                xi = xi - Hk*gradient(xi);
65
66
                if(!xi.allFinite()) break;
67
                iteration++;
69
            }
70
            return xi;
71
        };
72
73
        inline MatrixTf SR1(const MatrixTf& Hk, const VectorTf& Sk, const VectorTf&
74
    \rightarrow yk) {
            auto tmp = 1/((Sk - Hk*yk).transpose() * yk);
75
            return Hk + ((Sk - Hk*yk) * (Sk - Hk*yk).transpose()) * tmp;
76
        }
77
        inline MatrixTf BFGS(const MatrixTf& Hk, const VectorTf& Sk, const VectorTf&
78
        yk) {
            auto pk = 1/(yk.transpose() * Sk);
79
80
        (MatrixTf::Identity()-pk*Sk*yk.transpose())*Hk*(MatrixTf::Identity()-pk*yk*Sk.transpose())
```

20211046 Hyeonjang An

Homework 3

Numerical Opimization October 30, 2021

```
}
 };
82
 83
 }/// the end of namespace numerical_optimization ///
84
 85
 #endif //__QUASI_NEWTONS__
86
```