Homework 4

Problem

1. Implement the following numerical methods:
   (a) The method of steepest descent
   (b) Newton's methods
   (c) Two Quasi Newton's methods (SR1, BFGS)
2. Compare their performance for the following three problems:
   (a) $f(x, y) = (x + 2y - 6)^2 + (2x + y - 6)^2$
   (b) $(x, y) = 50 * (y - x^2)^2 + (1 - x)^2$
   (c) $f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$
3. First start at (1.2, 1.2) at each function. Then use different starting points to discuss how approximate point is moving on the contour plot of f(x, y)

Implementation

1. Implementation
   (a) **Steepest Descent Method**
      i. For the steepest descent method, the magnitude of gradient is used as termination criterion
      ii. Also, inexact line search method is adapted

```
#ifndef __CAUCHYS__
#define __CAUCHYS__

#include "multivariate.h"
#include "multi/termination.hpp"

namespace numerical_optimization {

template<typename VectorTf>
class Cauchys : public Multivariate<VectorTf> {
public:
    using Base = Multivariate<VectorTf>;
    using Base::Base;
    using Base::plot;
    using Base::function;
```

```cpp
    using Base::gradient;
    using function_t = typename Base::function_t;

    // constructors
    Cauchys(function_t f):Base(f){};

    // generally works
    VectorTf eval(const VectorTf& init=VectorTf::Random(), float e=epsilon) override {
        // 1. initialize
        VectorTf xi = init;
        // 2. loop
        for(size_t i=0; i<this->iter; i++) {
#ifdef BUILD_WITH_PLOTTING
            plot.emplace_back(std::make_pair(xi, function(xi)));
#endif
            // 1. termination
            if(terminate<Termination::Condition::MagnitudeGradient>({xi}, e)) break;

            // 2. the steepest descent direction
            VectorTf p = -1*gradient(xi)/gradient(xi).norm();

            // 3. step length
            float alpha = this->line_search_inexact(xi, p);

            // 4. update gradient
            xi = xi + alpha*p;
        }
        return xi;
    };

    // termination
    template<Termination::Condition CType>
    bool terminate(const std::vector<VectorTf>& x, float h=epsilon) const {
        return Termination::eval<VectorTf, CType>(function, x, h);
    }
};
//// ///////////////////////////////////////////////////
}/// the end of namespace numerical_optimization ///
//////////////////////////////////////////////////////
#endif //__CAUCHYS__
```

(b) **Gradient and Hessian**

i. Gradient and Hessian are computed as finite difference method
ii. For accuracy of compututing gradient, 8 values approximation method is taken
iii. It is also same as in the hessian computation

```cpp
#include <cmath>
#include <iostream>

#include "multivariate.h"

using namespace Eigen;

namespace numerical_optimization {

// two-variables case
// specialization of the vector2f case
template<>
Vector2f _gradient<Vector2f>(const std::function<float(const Vector2f&)>& f, const Vector2f& x, float h) {
    using v2 = Vector2f;

    float dx = 3*f(v2(x[0]-4*h, x[1]))-32*f(v2(x[0]-3*h, 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, 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[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]+2*h))+672*f(v2(x[0], x[1]+h));

    float inv = (1/(h*840));
    return v2(dx, dy)*inv;
}

// specialization of the vector2f case
template<>
Matrix2f _hessian<Vector2f>(const std::function<float(const Vector2f&)>& f, const Vector2f& x, float h) {
    using vec2 = Vector2f;

    h=0.01;
    auto dfdx = [&](vec2 x){
        float inv = (1/(h*840));
```

```
        float app = 3*f(vec2(x[0]-4*h, x[1]))-32*f(vec2(x[0]-3*h, 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, x[1]))-168*f(vec2(x[0]+2*h, x[1]))+672*f(vec2(x[0]+h, x[1]));
            return app*inv;
        };

    auto dfdy = [&](vec2 x){
        float inv = (1/(h*840));
        float app = 3*f(vec2(x[0], x[1]-4*h))-32*f(vec2(x[0], 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], x[1]+3*h))-168*f(vec2(x[0], x[1]+2*h))+672*f(vec2(x[0], x[1]+h));
        return app*inv;
        };

    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]));
    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)) + 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)) + f(vec2(x[0]-h, x[1]-h));
    float dyy = f(vec2(x[0], x[1]+2*h))-2*f(vec2(x[0], x[1]))+f(vec2(x[0], x[1]-2*h));

    Matrix2f m;
    m << dxx, dxy, dyx, dyy;
    float inv = 1/(4*h*h);
    return m*= inv;
}

template<>
Vector2d _gradient<Vector2d>(const std::function<float(const Vector2d&)>& f, const Vector2d& x, float h) {
    using v2 = Vector2d;

    double dx = 3*f(v2(x[0]-4*h, x[1]))-32*f(v2(x[0]-3*h, 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, x[1]))-168*f(v2(x[0]+2*h, x[1]))+672*f(v2(x[0]+h, x[1]));
    double 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[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]+2*h))+672*f(v2(x[0], x[1]+h));
```

```cpp

    double inv = (1/(h*840));
    return v2(dx, dy)*inv;
}

// specialization of the vector2f case
template<>
Matrix2d _hessian<Vector2d>(const std::function<float(const Vector2d&)>& f, const Vector2d& x, float h) {
    using vec2 = Vector2d;

    h=0.01;
    auto dfdx = [&](vec2 x){
        double inv = (1/(h*840));
        double app = 3*f(vec2(x[0]-4*h, x[1]))-32*f(vec2(x[0]-3*h, 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, x[1]))-168*f(vec2(x[0]+2*h, x[1]))+672*f(vec2(x[0]+h, x[1]));
            return app*inv;
        };

    auto dfdy = [&](vec2 x){
        double inv = (1/(h*840));
        double app = 3*f(vec2(x[0], x[1]-4*h))-32*f(vec2(x[0], 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], x[1]+3*h))-168*f(vec2(x[0], x[1]+2*h))+672*f(vec2(x[0], x[1]+h));
        return app*inv;
        };

    double dxx = f(vec2(x[0]+2*h, x[1]))-2*f(vec2(x[0], x[1]))+f(vec2(x[0]-2*h, x[1]));
    double 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)) + f(vec2(x[0]-h, x[1]-h));
    double 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)) + f(vec2(x[0]-h, x[1]-h));
    double dyy = f(vec2(x[0], x[1]+2*h))-2*f(vec2(x[0], x[1]))+f(vec2(x[0], x[1]-2*h));

    Matrix2d m;
    m << dxx, dxy, dyx, dyy;
    double inv = 1/(4*h*h);
    return m*= inv;
}
```

```
/////////////////////////////////////////////////////
} /// the end of namespace numerical_optimization ///
/////////////////////////////////////////////////////
```

(c) **Newton's method**

i. For the steepest descent method, the magnitude of gradient is used as termination criterion

```cpp
#ifndef __NEWTONS__
#define __NEWTONS__

#include "multivariate.h"
#include "multi/termination.hpp"

namespace numerical_optimization {

template<typename VectorTf>
class Newtons : public Multivariate<VectorTf> {
public:
    using Base = Multivariate<VectorTf>;
    using Base::Base;
    using Base::plot;
    using Base::function;
    using Base::gradient;
    using Base::hessian;
    using function_t = typename Base::function_t;

    // constructors
    template<Termination::Condition CType>
    bool terminate(const std::vector<VectorTf>& x, float h=epsilon) const {
        return Termination::eval<VectorTf, CType>(function, x, h);
    }

    // generally works
    VectorTf eval(const VectorTf& init=VectorTf::Random(), float e=epsilon) override {
        VectorTf xi = init;
        for(size_t i=0; i<this->iter; i++) {
#ifdef BUILD_WITH_PLOTTING
            plot.emplace_back(std::make_pair(xi, function(xi)));
#endif
            // 1. termination
            if(terminate<Termination::Condition::MagnitudeGradient>({xi}, e)) break;

            // 2. gradient update
            xi = xi - hessian(xi).inverse()*gradient(xi);
        }
```

```
        return xi;
    };
};
//// ////////////////////////////////////////////////
}/// the end of namespace numerical_optimization ///
/////////////////////////////////////////////////////
#endif //__CAUCHYS__
```

(d) **Quasi-Newton's method**

i. Both exact line search (by golden section search method) and inexact line search methods are implemented

ii. I adapt the inexact line search method, because I have thought the overhead is lower than exact line search method

iii. The convergence is depend on finding a step lenght, alpha. In the case of inexact step length, depending on ρ and initial alpha, the step length could be computed badly. In that case, the function failed to find a optimal point.

```
#ifndef __QUASI_NEWTONS__
#define __QUASI_NEWTONS__

#include <math.h>
#include <cassert>
#include "multivariate.h"
#include "multi/termination.hpp"

namespace numerical_optimization {
namespace quasi_newtons {
enum Rank { SR1, BFGS, };
};

template<typename VectorTf, quasi_newtons::Rank RankMethod>
class QuasiNewtons : public Multivariate<VectorTf> {
public:
    using Base = Multivariate<VectorTf>;
    using Base::Base;
    using Base::plot;
    using Base::iter;
    using Base::function;
    using Base::gradient;
    using function_t = typename Base::function_t;
    using MatrixTf = Eigen::Matrix<typename VectorTf::Scalar,
    VectorTf::RowsAtCompileTime, VectorTf::RowsAtCompileTime>;

    template<Termination::Condition CType>
    bool terminate(const std::vector<VectorTf>& x, float h, float eps=epsilon) {
        return Termination::eval<VectorTf, CType>(function, x, h, eps);
    }
    VectorTf eval(const VectorTf& init=VectorTf::Random(), float e=epsilon)
    override {

        VectorTf xi = init;
```

```
        MatrixTf Hk = MatrixTf::Identity();

        size_t iteration = 0;
        for(size_t i=0; i<this->iter; i++) {

#ifdef BUILD_WITH_PLOTTING
            plot.emplace_back(std::make_pair(xi, function(xi)));
#endif
            // Compute a Search Direction
            VectorTf p = (-1*Hk*gradient(xi)).normalized();

            // Compute a step length Wolfe Condition
            double alpha = 0;
            if constexpr (RankMethod==quasi_newtons::Rank::SR1)
                alpha = this->line_search_inexact(xi, p, 0.99, 0.5, 3);
            else if constexpr (RankMethod==quasi_newtons::Rank::BFGS)
                alpha = this->line_search_inexact(xi, p, 0.8, 0.5, 3);

            // Define sk and yk
            VectorTf Sk = alpha*p;
            VectorTf yk = gradient(xi+Sk) - gradient(xi);

            // Compute Hk+1
            if constexpr (RankMethod==quasi_newtons::Rank::SR1)
                Hk = SR1(Hk, Sk, yk);
            else if constexpr (RankMethod==quasi_newtons::Rank::BFGS)
                Hk = BFGS(Hk, Sk, yk);

            xi = xi - Hk*gradient(xi);

            if constexpr (RankMethod==quasi_newtons::Rank::SR1) {
                if(terminate<Termination::Condition::MagnitudeGradient>({xi},
    ↪   0.01)) break;
            }
            else if constexpr (RankMethod==quasi_newtons::Rank::BFGS) {
                if(terminate<Termination::Condition::MagnitudeGradient>({xi},
    ↪   1e-8)) break;
            }
        }
        return xi;
    };

    inline MatrixTf SR1(const MatrixTf& Hk, const VectorTf& Sk, const VectorTf&
    ↪   yk) {
```

```cpp
        auto frac = 1/((Sk - Hk*yk).transpose()*yk);
        return Hk + ((Sk-Hk*yk) * (Sk-Hk*yk).transpose())*frac;
    }
    inline MatrixTf BFGS(const MatrixTf& Hk, const VectorTf& Sk, const VectorTf& yk) {
        auto pk = 1/(yk.transpose() * Sk);
        return (MatrixTf::Identity()-pk*Sk*yk.transpose())*Hk*(MatrixTf::Identity()-pk*yk*Sk.transpose())
    }
};
//// ////////////////////////////////////////////////
}/// the end of namespace numerical_optimization ///
////////////////////////////////////////////////////
#endif //__QUASI_NEWTONS__
```

Analysis

1. Convergence and Characteristic

- In the case of the method of steepest descent, it is reliable. Whatever the function is, Wherever to start, it converges to the optimal point.
- In Newton's method, depending on the initial point, it usaully failed to converge. Also, in the 3rd function case, it almost failed to converge. Only in the 1st function case, it showed the guaranteed optimal point.
- In Quasi-Newton's method, the convergence speed is not that fast as Newton's method. Moreover, it has a large dependecy on search a step length alpha. When the alpha value is not that good, it showed the failure cases. Especially, SR1 method is more sensitive to this alpha value.

initial point	$f(x, y)$	Convergence Points(x, y)			
		Steepest descent	Newton's	SR1	BFGS
[1.2, 1.2]	(a)	[2.003, 2.003]	[2, 2]	[2, 2]	[2, 2]
	(b)	[1, 1]	[1, 1]	[1.208, 1.467]	[1, 1]
	(c)	[2.999, 0.501]	fail	[2.914, 0.473]	[3, 0.5]
[5.6,-1.2]	(a)	[1.999, 1.999]	[2, 2]	[2, 2]	[2, 2]
	(b)	[0.999, 0.991]	fail[-152.879, 563.325]	[1, 1]	[1, 1.000]
	(c)	[3.000, 0.4980]	fail[0.007, 1.026]	fail[5.6, -1.2]	fail[5.6, -1.2]
[-3.5,2.3]	(a)	[1.999, 1.999]	[2, 2]	[2, 2]	[2, 2]
	(b)	[0.996, 0.986]	fail[-nan, -nan]	[1, 1]	[1, 1]
[10.5,-8.3]	(a)	[1.999, 1.999]	[2, 2]	[2, 2]	[2, 2]
	(b)	[0.991, 0.988]	fail	fail	fail
	(c)	[8.821, 0.972]	fail	fail	fail

Performance

1. Convergence speed

   - The method of steepest descent has the slowest convergence speed
   - As before said, I expected the backtracking line search gives more faster speed to search a step length.
   - But it is not. Because the Quasi-Newton's methods are depending on searching a step length, when the method failed to find an adquate step length, it takes a more time and even fails to converge
   - In the case of SR1, because of the vulnerability to find a step length, it takes a more time than BFGS

initial point	$f(x, y)$	Performance(x, y)			
		Steepest descent	Newton's	SR1	BFGS
[1.2, 1.2]	(a)	151826251312 ns	167781 ns	4302569 ns	1657282 ns
	(b)	147027616816 ns	200670041 ns	19256927 ns	2452604 ns
	(c)	270048645501 ns	fail	fail	20631360 ns
[5.6,-1.2]	(a)	149522732624 ns	186664 ns	7291947 ns	2341505 ns
[-3.5,2.3]	(a)	148679627984 ns	203709843 ns	5417397 ns	1443039 ns
[10.5,-8.3]	(a)	131232100735 ns	167805 ns	5858973 ns	3033800 ns

Homework 4

Plotting

1. Discuss for moving the approximate points

   - I choose the linear function, which always converges, to observe the movement of approximate points.
   - Excepted the method of steepest descent, we can observe that Newton's method and Quasi-Newton's method takes a large step length.
   - Therefore, the plottings draw the sharp lines in the kind of Newton's methods
   - It appears in Figure 1, 4, 5

figures/a SteepestDescent [1.2, 1.2].png figures/a Newtons [1.2, 1.2].png figures/a SR1 [1.2, 1.2].png figures/a BFGS [1.2, 1.2].png $f(x, y) = (x + 2y - 6)^2 + (2x + y - 6)^2$

figures/b SteepestDescent [1.2, 1.2].png figures/b Newtons [1.2, 1.2].png figures/b SR1 [1.2, 1.2].png figures/b BFGS [1.2, 1.2].png $(x, y) = 50 * (y - x^2)^2 + (1 - x)^2$

figures/c SteepestDescent [1.2, 1.2].png figures/c Newtons [1.2, 1.2].png figures/c SR1 [1.2, 1.2].png figures/c BFGS [1.2, 1.2].png $f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$

figures/a SteepestDescent [5.6,1.2].png figures/a Newtons [5.6,1.2].png figures/a SR1 [5.6,1.2].png figures/a BFGS [5.6,1.2].png Moving comparison[5.6,1.2]: $f(x, y) = (x+2y-6)^2+(2x+y-6)^2$

figures/a SteepestDescent [-3.5,2.3].png figures/a Newtons [-3.5,2.3].png figures/a SR1 [-3.5,2.3].png figures/a BFGS [-3.5,2.3].png Moving comparison[-3.5,2.3] $f(x, y) = (x + 2y - 6)^2 + (2x + y - 6)^2$

figures/a SteepestDescent [10.5,-8.3].png figures/a Newtons [10.5,-8.3].png figures/a SR1 [10.5,-8.3].png figures/a BFGS [10.5,-8.3].png Moving comparison[10.5,-8.3] $f(x, y) = (x+2y-6)^2 + (2x + y - 6)^2$