https://github.com/hyeonjang/numerical-optimization
20211046
Hyeonjang An

Homework 3

EC6301
Numerical Opimization
October 30, 2021

1. Implementation

   (a) **Steepest Descent Method**

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

```cpp
#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;
    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. Three control parameters are set as $\alpha = 1$, $\beta = 2$, $\gamma = 0.5$

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

#include "multivariate.h"

using namespace Eigen;

namespace numerical_optimization {

// two-variables case
// specialization of the vector2f case
// https://stackoverflow.com/questions/38854363/is-there-any-standard-way-to-calculate-the-nu
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;
}
//////////////////////////////////////////////////////
} /// the end of namespace numerical_optimization ///
//////////////////////////////////////////////////////
```

(c) **Newton's method**

i.

```
#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.

```
#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
```

```
            // @@todo other termination method
            if(terminate<Termination::Condition::MagnitudeGradient
            |Termination::Condition::FunctionValueDifferenceRelative>({xi}, 1e-5)) {
                break;
                }

            // Compute a Search Direction
            VectorTf p = -1 * Hk*gradient(xi);

            // Compute a step length Wolfe Condition
            // float alpha = this->line_search_inexact(xi, p, 0.99, 0.5);

            // Compute a step length exactly
            float alpha = this->line_search_exact(xi, p);

            // 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(!xi.allFinite()) break;

            iteration++;
        }
        return xi;
    };

    inline MatrixTf SR1(const MatrixTf& Hk, const VectorTf& Sk, const VectorTf& yk) {
        auto tmp = 1/((Sk - Hk*yk).transpose() * yk);
        return Hk + ((Sk - Hk*yk) * (Sk - Hk*yk).transpose()) * tmp ;
    }
    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__
```