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

EC6301
Numerical Opimization
October 21, 2021

Homework 3

Problem

1. Implement the Nelder-Mead method and the Powell's method to find the minimum of
   (a) $f(x, y) = (x + 2y)^2 + (2x + y)^2$
   (b) $(x, y) = 50 * (y - x^2)^2 + (1 - x)^2$
   (c) $f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$
2. Use your own termination criterion. Compare and discuss their performances. If possible, show how the best point is moving on the contour plot of f(x, y)

Implementation - methods

1. Computation result: convergence points
   In the case of the first function, the results are the approximation of the value zero.
   In the case of the second function, Powell's method is sometimes stucked to local minima. [0.510332, 0.263043] is the failure case to find optimization points.

function $f(x, y)$	Convergence Points(x, y)	
	Nelder-Mead	Powell's
(a)	[1.17932e-07, -1.85511e-07]	[-7.20495e-23, 5.73423e-23]
(b)	[1, 1]	[0.510332, 0.264043] / [0.99972, 0.999437]
(c)	[3, 0.5]	[2.99989, 0.499973]

2. Implementation
   initial points are randomly given.
   (a) **Nelder-Mead method**
   Three control parameters are set as $\alpha = 1$, $\beta = 2$, $\gamma = 0.5$
   The maximum iteration is 10000.
   The termination condition is the "magnitude of gradient".

```
#ifndef __NELDER_MEAD__
#define __NELDER_MEAD__

#include "multivariate.h"

namespace numerical_optimization {

template<typename VectorTf>
class NelderMead : public Multivariate<VectorTf> {
```

```
public:
    using Base = Multivariate<VectorTf>;
    using Base::Base;
    using Base::plot;
    using Base::function;
    using function_t = typename Base::function_t;

    // constructors
    NelderMead(Base base):Base(base),alpha(1),beta(2),gamma(0.5){};
    NelderMead(function_t
↪   func):Base(func),alpha(1),beta(2),gamma(0.5){};
    NelderMead(Base base, float a, float b, float
↪   c):Base(base),alpha(a),beta(b),gamma(c){};
    NelderMead(function_t func, float a, float b, float
↪   c):Base(func),alpha(a),beta(b),gamma(c){};

    // generally works
    VectorTf eval(float e=epsilon) override {
        // 1. get the number of dimension and select threshold
        constexpr size_t dim = VectorTf::RowsAtCompileTime;

        // 2. initialize with random
        std::vector<VectorTf> x(dim+1);
        for(auto& s:x) { s=VectorTf::Random(); }

        for(size_t i=0; i<10000; i++) {
            // 0. termination
            if(this->magnitude_gradient(x, e)) break;

#ifdef BUILD_WITH_PLOTTING
        for(auto t:x) plot.emplace_back(std::make_pair(t,
↪   function(t)));
#endif
            // 1. reflection
            std::sort(
                x.begin(), x.end(),
                [&](VectorTf l, VectorTf& r){ return
↪   function(l)<function(r); }
                );

            VectorTf c =
                (std::accumulate(x.begin(), x.end()-1,
↪   VectorTf::Zero().eval()))/(x.size()-1);

```

```cpp
            auto xr = reflecting(x.back(), c);
            auto f1 = function(x[0]), fr = function(xr), fN =
    function(x[x.size()-2]);

            if(f1<=fr && fr<=fN) {
                x.back() = xr;
                continue;

            // 2. expansion
            } else if(fr<=f1) {
                auto xe = expanding(xr, c);
                x.back() = xe;

            // 3. contraction
            } else if(fr>=fN) {
                // last value evalution
                auto fN1 = function(x.back());

                auto xc = contracting(xr, x.back(), c, fr<fN1);
                auto fc = function(xc);

                // contraction evaluation
                if(fc<std::min(fr, fN1)) {
                    x.back() = xc;
                } else {
                    for(auto& xi : x)
                        xi = (xi + x.front())/2;
                }
            }
        }
        return x[0];
    };

    inline VectorTf reflecting(const VectorTf& x_last, const VectorTf&
    center) {
        return center + alpha*(center-x_last);
    };

    inline VectorTf expanding(const VectorTf& xr, const VectorTf&
    center) {
        VectorTf xe = center + beta*(xr-center);
        return (function(xe)<=function(xr)) ? xe : xr;
    };

    inline VectorTf contracting(const VectorTf& xr, const VectorTf&
↪   x_last, const VectorTf& center, bool check) {
        return check ?
↪   (center+gamma*(xr-center)):(center+gamma*(x_last-center));
    }

private:
    float alpha, beta, gamma;
};
//// ////////////////////////////////////////////////
}/// the end of namespace numerical_optimization ///
////////////////////////////////////////////////////
#endif //__NEDLER_MEAD__
```

(b) **Powell's method**
For univarite searching, I used the golden section search method.
The maximum iteration is 10000 as same as Nelder-Mead method.
When using the termination criterion as the "magnitude of gradient", it occurs to be stucked in local minima. So the termination condition is changed to "consecutive relative difference". However this does not largely affect to the performance.

```cpp
#ifndef __POWELLS__
#define __POWELLS__

#include "univariate.h"
#include "multivariate.h"

namespace numerical_optimization {

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

    VectorTf eval(float e=epsilon) override {
        constexpr size_t dim = VectorTf::RowsAtCompileTime;

        // 1. initialize
        std::vector<VectorTf> p(dim);  // points
        std::vector<VectorTf> u(dim); // unit directions
        for(size_t i=0; i<p.size(); i++) p[i] = VectorTf::Random()*3;
        for(size_t i=0; i<u.size(); i++) u[i][i] = 1;

        // 2. algorithm start
        VectorTf xi = p[0]; // S1
        for(size_t j=0; j<10000; j++) {
            for(size_t k=0; k<dim-1; k++) {
                uni::function_t func0 = [&](float gamma){ return
↪   function(p[k] + gamma*u[k]); }; // S2
                Univariate uni0 = Univariate(func0);
                float min_gamma0 = uni0.golden_section();
                p[k+1] = p[k] + min_gamma0*u[k];
            }
            for(size_t k=0; k<dim-1; k++) u[k] = u[k+1];    // S4
            u[dim-1] = p[dim-1] - p[0];                     // S4
            uni::function_t func1 = [&](float gamma){ return
    ↪   function(p[0] + gamma*u[dim-1]); }; // S5
            Univariate uni1 = Univariate(func1);
            float min_gamma1 = uni1.golden_section();
            auto tmp = xi;
            xi = p[0] + min_gamma1*u[dim-1];

#ifdef BUILD_WITH_PLOTTING
            plot.emplace_back(std::make_pair(xi, function(xi)));
#endif
            p[0] = xi;
            if(this->consecutive_difference_relative({xi, tmp}, e))
    ↪   break;
        }
        return p[0];
    }
};
/////////////////////////////////////////////////////
} /// the end of namespace numerical_optimization ///
/////////////////////////////////////////////////////
#endif //__POWELLS__
```

Implementation - Terminatination criterion

3. Termination criterion
   I have implemented all six conditions. Also, to calculate the gradient of functions, I used numerical method to derive gradient from function given point (x, y)

   - Termination criterion

```
#ifndef __MULTIVARIATE_H__
#define __MULTIVARIATE_H__

#include <algorithm>
#include <vector>
#include <numeric>
#include <functional>
#include <Eigen/Dense>
#include "fwd.h"
#include "method.h"

#define Log(x) printf("position: [%f, %f], value: %f\n", x[0], x[1],
↪  function(x))

namespace numerical_optimization {

template<typename VectorTf>
VectorTf _gradient(const std::function<float(const VectorTf&)>& f,
↪  const VectorTf& x, float h=1);
template<typename VectorTf, typename ReturnType =
↪  Eigen::Matrix<typename VectorTf::Scalar,
↪  VectorTf::RowsAtCompileTime, VectorTf::RowsAtCompileTime>>
ReturnType _hessian(const std::function<float(const VectorTf&)>& f,
↪  const VectorTf& x, float h=1);

template<typename VectorTf>
class Multivariate : public Method {
public:
    using function_t = multi::function_t<VectorTf>;
    Multivariate(){};
    Multivariate(function_t f):function(f){};

    // functions
    virtual VectorTf eval(float _=epsilon){ return VectorTf(); }
    virtual VectorTf eval(const VectorTf& init, float _=epsilon){
↪  return VectorTf(); }
```

Homework 3

```cpp

#ifdef BUILD_WITH_PLOTTING
    std::vector<std::pair<VectorTf, float>> plot;
#endif
protected:
    size_t     iter=0;
    function_t function;

public:
    // line search for alpha
    inline float line_search_inexact(const VectorTf& xk, const
 ↪  VectorTf& pk, float p, float c) const {

        // initial alpha value
        float alpha = 0.1;

        // check satisfying wolfe 1st condition
        auto wolfe_1st = [&](float a){ return
 ↪  function(xk+a*pk)<=(a*c*gradient(xk).transpose()*pk +
 ↪  function(xk)); };

        while(!wolfe_1st(alpha)) {
            alpha = alpha*p;
        }

        return alpha;
    }

    // calculate gradient
    inline VectorTf gradient(VectorTf x, float h=epsilon) const {
        return _gradient(function, x, h);
    }

    // calculate hessian & inverse
    inline decltype(auto) hessian(VectorTf x, float h=epsilon) const {
        return _hessian(function, x, h);
    }

    // 1. Difference of two consecutive estimates
    inline bool consecutive_difference(const std::vector<VectorTf>& x,
 ↪  float eps=epsilon) const {
        bool flag = true;
        for(size_t k=0; k<x.size(); k++) {
            size_t k1 = (k+1)%x.size(); // indexing
            flag &= (x[k1]-x[k]).norm()<eps;
        }
        return flag;
    };
    // 2. Relative Difference of two consecutive estimates
    inline bool consecutive_difference_relative(const
↪   std::vector<VectorTf>& x, float eps=epsilon) const {
        bool flag = true;
        for(size_t k=0; k<x.size(); k++) {
            size_t k1 = (k+1)%x.size();
            flag &= (x[k1]-x[k]).norm()/x[k1].norm()<eps;
        }
        return flag;
    };
    // 3. Magnitude of Gradient
    inline bool magnitude_gradient(const std::vector<VectorTf>& x,
↪   float eps=epsilon) const {
        bool flag = true;
        for(size_t k=0; k<x.size(); k++) {
            flag &= gradient(x[k]).norm()<eps;
        }
        return flag;
    };
    // 4. Relative Difference of function values
    inline bool function_value_difference_relative(const
↪   std::vector<VectorTf>& x, float eps=epsilon) const {
        bool flag = true;
        for(size_t k=0; k<x.size(); k++) {
            size_t k1 = (k+1)%x.size();
            flag &=
↪   std::abs(function(x[k1])-function(x[k]))/std::abs(function(x[k1]))
↪   < eps;
        }
        return flag;
    };
    // 5. Descent direction change
    inline bool descent_direction_change(const std::vector<VectorTf>&
↪   x, const std::vector<VectorTf>& p) const {
        bool flag = true;
        for(size_t k=0; x.size(); k++) {
            flag &= (p[k]*gradient(x[k]))>=0.f;
        }
        return flag;
    };
    // 6. Maximum number of iterations
    inline bool over_maximum_iteration() const {
        return iter >= max_iter;
    };

};
/////////////////////////////////////////////////////
} /// the end of namespace numerical_optimization ///
/////////////////////////////////////////////////////
#endif // __MULTIVARIATE_H__
```

- Gradient calculation

```
#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-calcu
template<>
Vector2f _gradient<Vector2f>(const std::function<float(const
↪   Vector2f&)>& f, const Vector2f& x, float h) {

    Vector2f result = Vector2f::Zero();

    Vector2f eps = Vector2f(h, h);
    // relative 'h' value
    // cannot work for vector has zero: it results NaN
    if(x[0]!=0 && x[1]!=0) {
        eps[0] = x[0]*sqrtf(eps[0]);
        eps[1] = x[1]*sqrtf(eps[1]);
    }

    result[0] = (f(Vector2f(x[0]+eps[0],
↪   x[1]))-f(Vector2f(x[0]-eps[0], x[1]))) / (2*eps[0]);
    result[1] = (f(Vector2f(x[0], x[1]+eps[1]))-f(Vector2f(x[0],
↪   x[1]-eps[1]))) / (2*eps[1]);
```

```

    return result;
}

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

    float a = 0.01;

    float dxx = (f(Vector2f(x[0]+2*a, x[1])) - 2*f(Vector2f(x[0],
↪ x[1]))
                + f(Vector2f(x[0]-2*a, x[1])));
    float dxy = (f(Vector2f(x[0]+a, x[1]+a)) - f(Vector2f(x[0]-a,
↪ x[1]+a))
                - f(Vector2f(x[0]+a, x[1]-a)) + f(Vector2f(x[0]-a,
↪ x[1]-a)));
    float dyx = (f(Vector2f(x[0]+a, x[1]+a)) - f(Vector2f(x[0]+a,
↪ x[1]-a))
                - f(Vector2f(x[0]-a, x[1]+a)) + f(Vector2f(x[0]-a,
↪ x[1]-a)));
    float dyy = (f(Vector2f(x[0], x[1]+2*a)) - 2*f(Vector2f(x[0],
↪ x[1]))
                + f(Vector2f(x[0], x[1]-2*a)));

    Matrix2f m;
    m << dxx, dxy, dyx, dyy;
    m /= 4*a*a;

    return m;
}

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

Performance and Plot

4. Performace

Homework 3

	function $f(x, y)$	performance	
		Nelder-Mead	Powell's
(a)	$(x + 2y)^2 + (2x + y)^2$	872972 ns	84036883 ns
(b)	$50 * (y - x^2)^2 + (1 - x)^2$	132298767 ns	526430665 ns
(c)	$(1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$	214017830 ns	336396522 ns

- The convergence speed of Powell's method is worse than Nelder-Mead method for every given functions.
- It is because Powell's method has a dependency on the univarite method.
- Because I have given the initial points randomly, it happens not to converge. The Figure 2, which shows the result of Powell's method of the second function, is the case which cannot find the global minima.

Homework 3

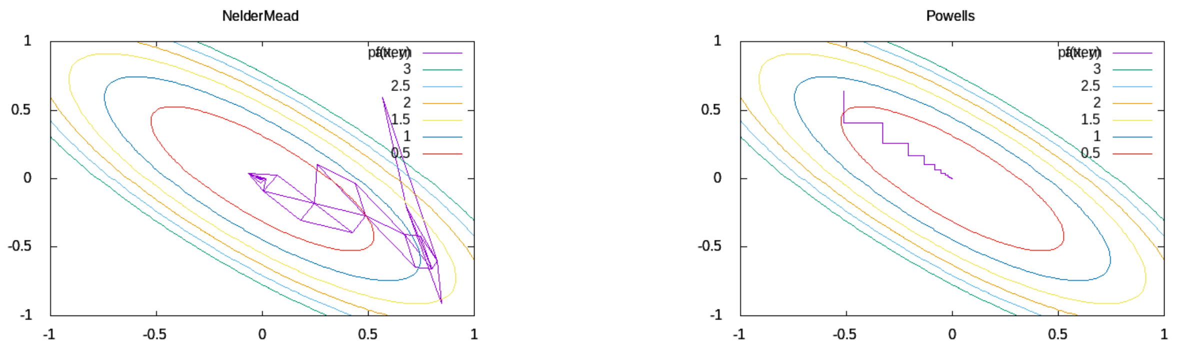
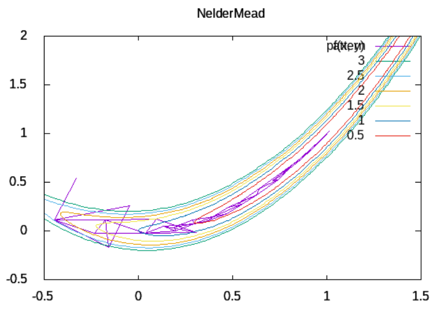
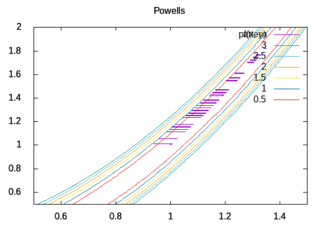


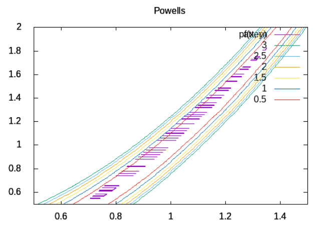
Figure 1: $f(x,y) = (x+2y)^2 + (2x+y)^2$



(a) Nelder-Mead



(b) Powell's: global



(c) Powell's: local

Figure 2: $f(x,y) = 50 * (y - x^2)^2 + (1-x)^2$

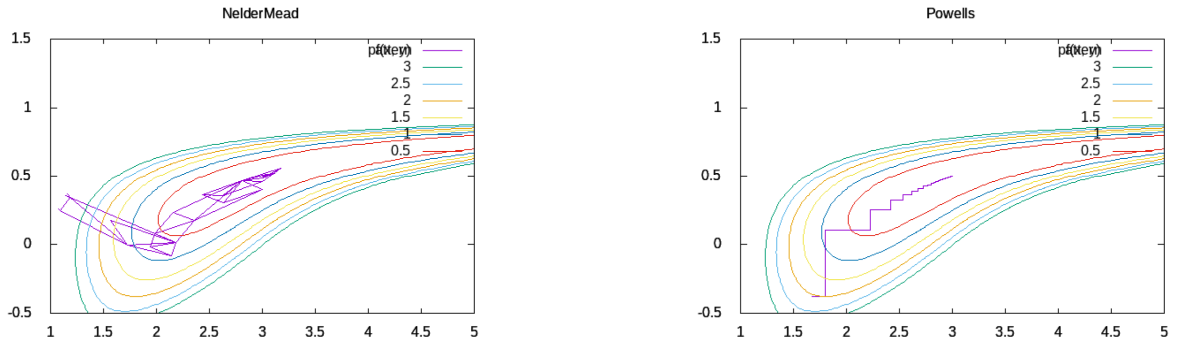


Figure 3: $f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$