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

EC6301
Numerical Opimization
October 30, 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. Added termination criterion
   -

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]	[1.000, 1.000]	[1.016, 1.027]
	(c)	[2.999, 0.501]	fail[0.041, 1.001]	fail[1.2, 1.2]	fail[7.11,-1.59]
[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]
	(a)	[1.999, 1.999]	[2, 2]	[2, 2]	[2, 2]
[10.5,-8.3]	(b)	[0.991, 0.988]	fail	fail	fail
	(c)	[8.821, 0.972]	fail	fail	fail

Implementation

1. Implementation
   (a) **Steepest Descent Method**
      i. Three control parameters are set as $\alpha = 1$, $\beta = 2$, $\gamma = 0.5$

```
#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
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.

```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.

```cpp
#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()
```

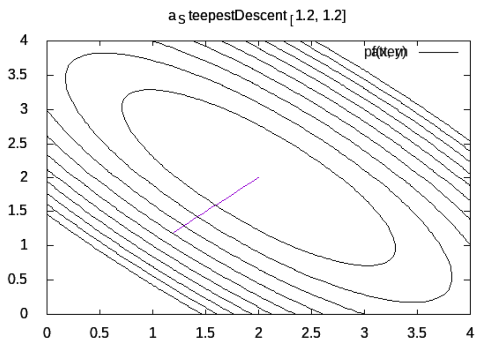
```
    }
};
//// ////////////////////////////////////////////////
}/// the end of namespace numerical_optimization ///
////////////////////////////////////////////////////
#endif //__QUASI_NEWTONS__
```

Performance and Plot

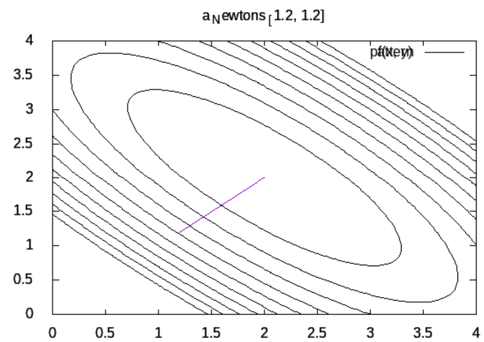
initial point	$f(x,y)$	Performance(x,y)			
		Steepest descent	Newton's	SR1	BFGS
[1.2, 1.2]	(a)	[2.003, 2.003]	[2, 2]	[2, 2]	[2, 2]
	(b)	[1, 1]	[1, 1]	[1.000, 1.000]	[1.016, 1.027]
	(c)	[2.999, 0.501]	fail[0.041, 1.001]	fail[1.2, 1.2]	fail[7.11,-1.59]
[5.6,-1.2]	(a)	[1.999, 1.999]	[2, 2]	[2, 2]	[2, 2]
	(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

- 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.

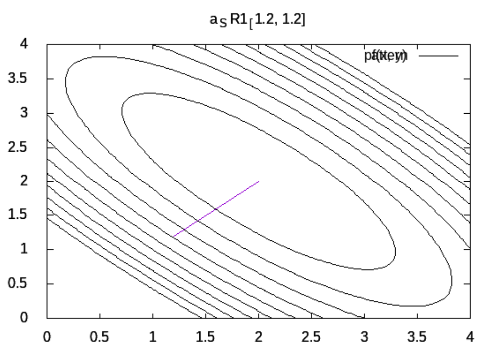
Plotting



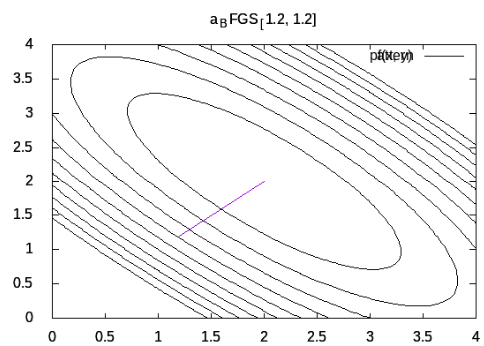
(a) SteepestDescent



(b) Newtons

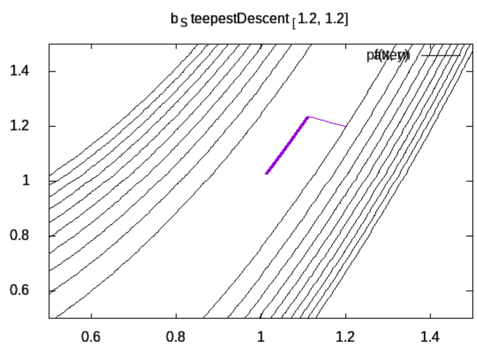


(c) SR1

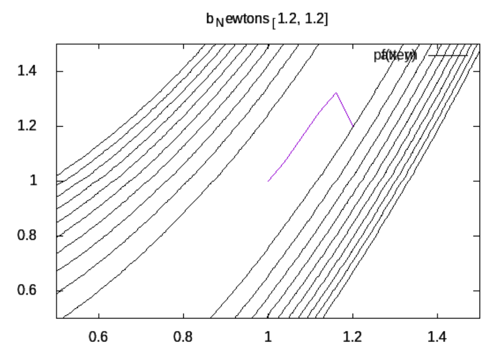


(d) BFGS

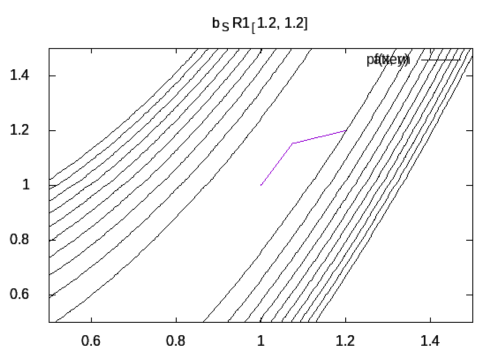
Figure 1: $f(x, y) = (x + 2y - 6)^2 + (2x + y - 6)^2$



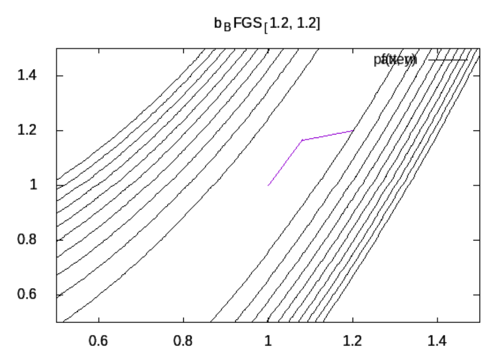
(a) SteepestDescent



(b) Newtons

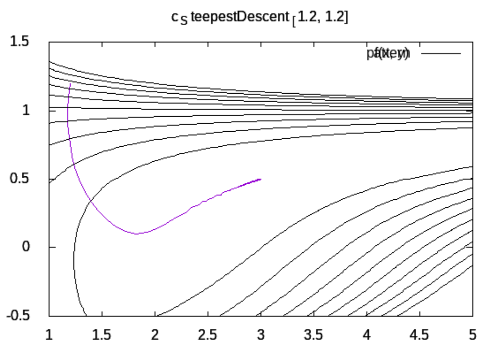


(c) SR1

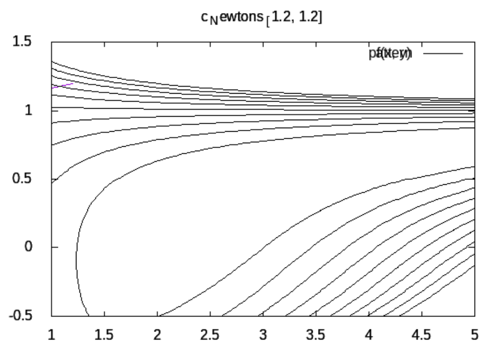


(d) BFGS

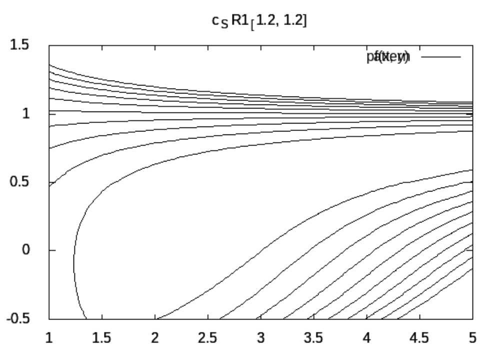
Figure 2: $(x, y) = 50 * (y - x^2)^2 + (1 - x)^2$



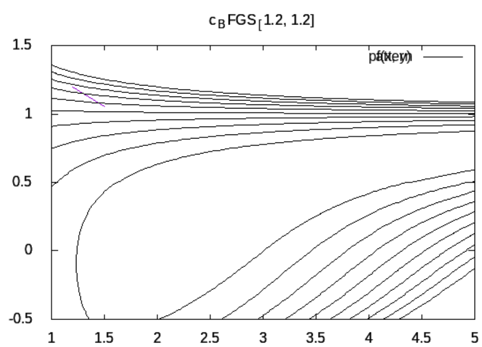
(a) SteepestDescent



(b) Newtons

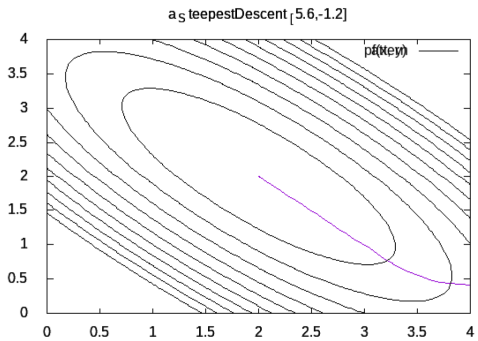


(c) SR1

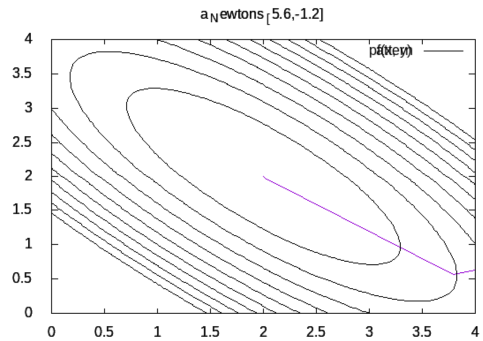


(d) BFGS

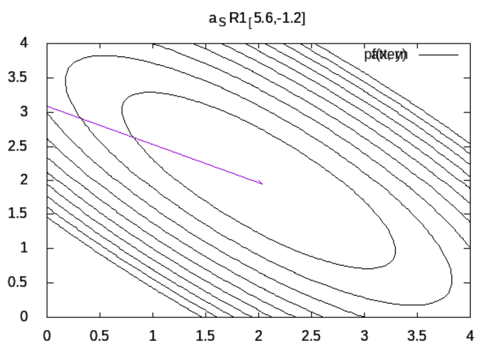
Figure 3: $f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$



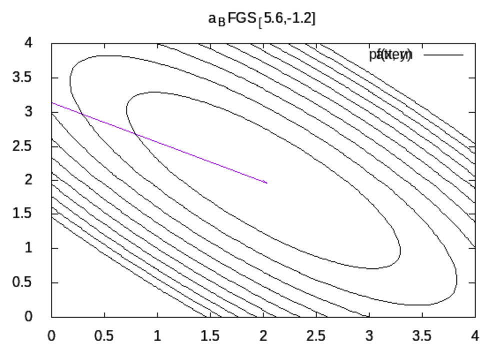
(a) SteepestDescent



(b) Newtons

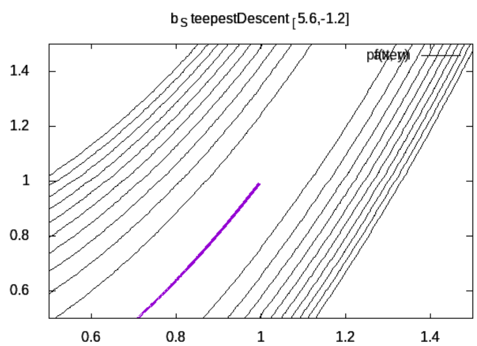


(c) SR1

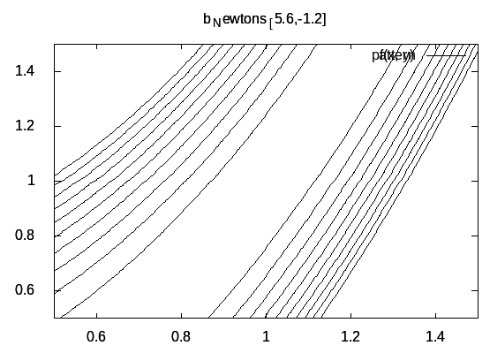


(d) BFGS

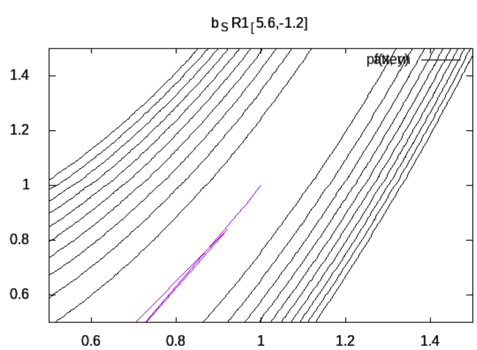
Figure 4: $f(x, y) = (x + 2y - 6)^2 + (2x + y - 6)^2$



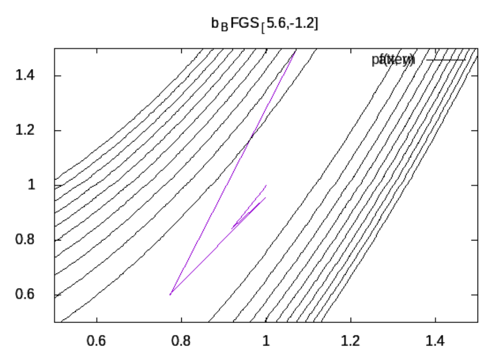
(a) SteepestDescent



(b) Newtons

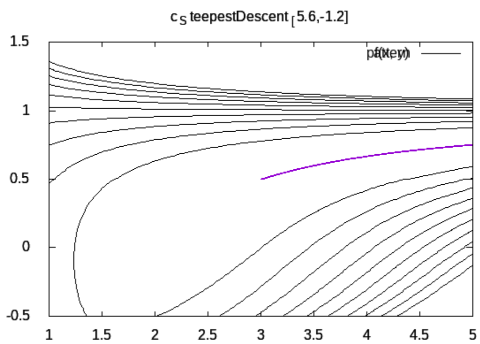


(c) SR1

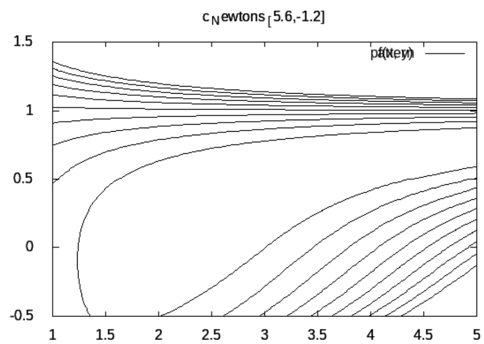


(d) BFGS

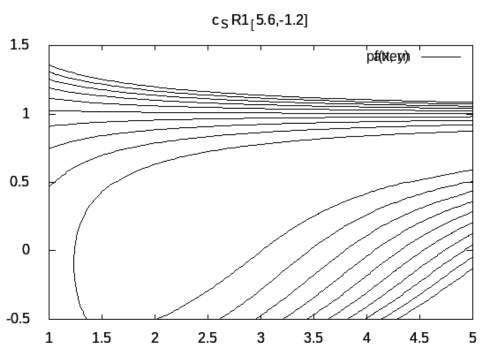
Figure 5: $(x, y) = 50 * (y - x^2)^2 + (1 - x)^2$



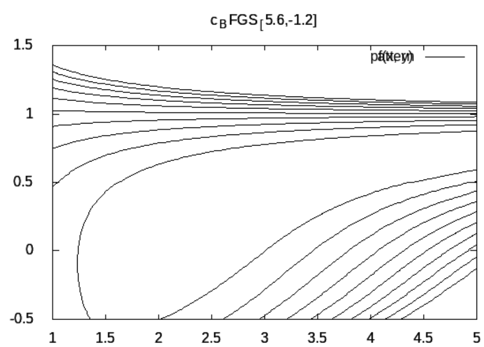
(a) SteepestDescent



(b) Newtons

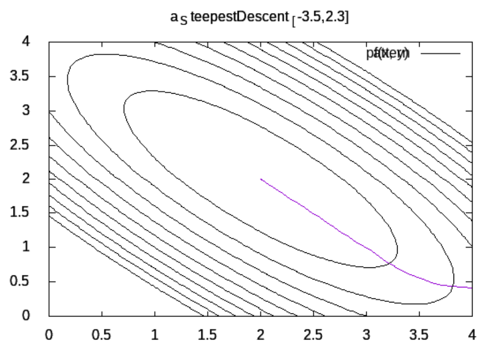


(c) SR1

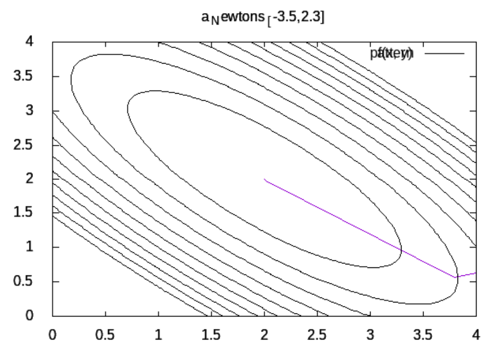


(d) BFGS

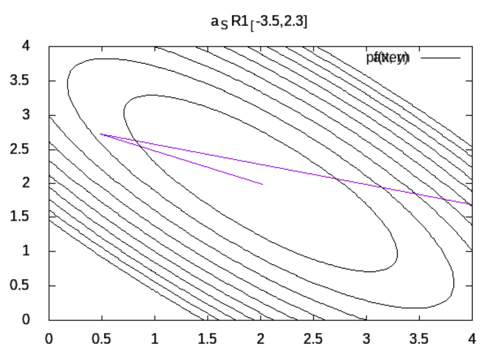
Figure 6: $f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$



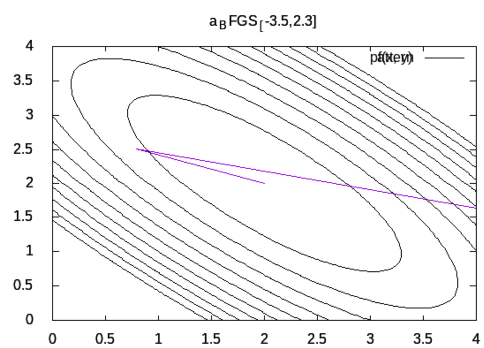
(a) SteepestDescent



(b) Newtons

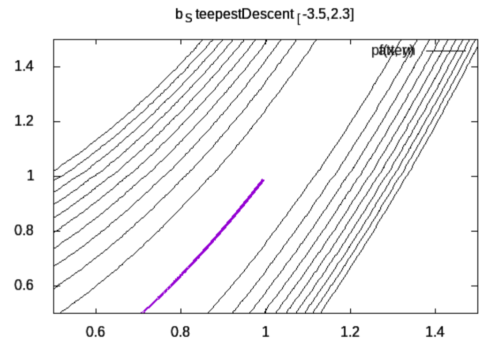


(c) SR1

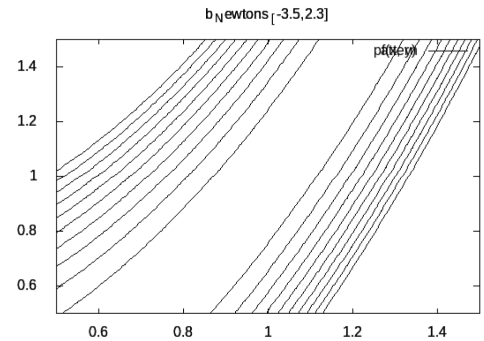


(d) BFGS

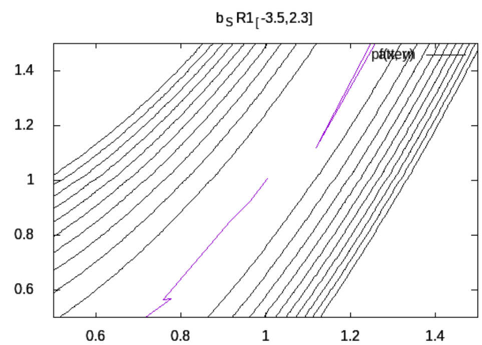
Figure 7: $f(x, y) = (x + 2y - 6)^2 + (2x + y - 6)^2$



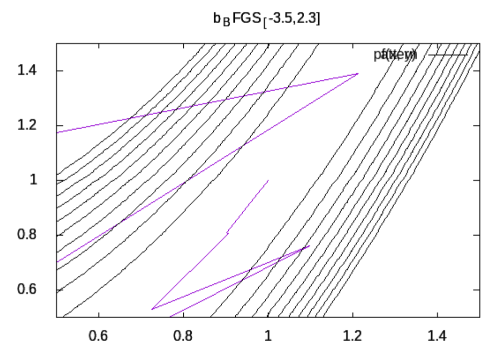
(a) SteepestDescent



(b) Newtons



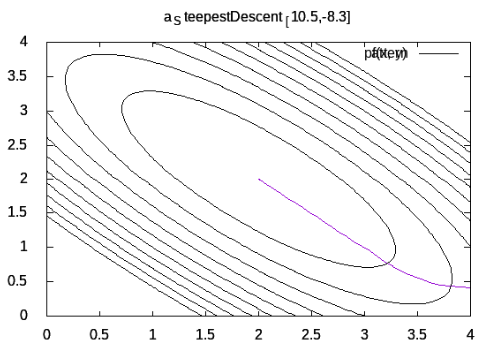
(c) SR1



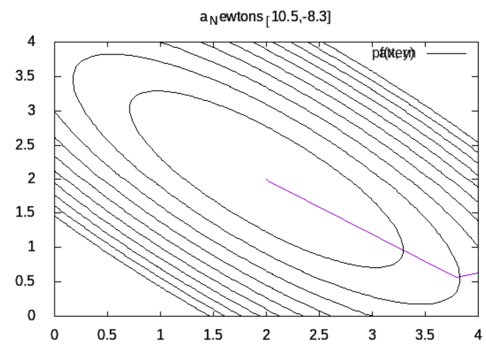
(d) BFGS

Figure 8: $(x, y) = 50 * (y - x^2)^2 + (1 - x)^2$

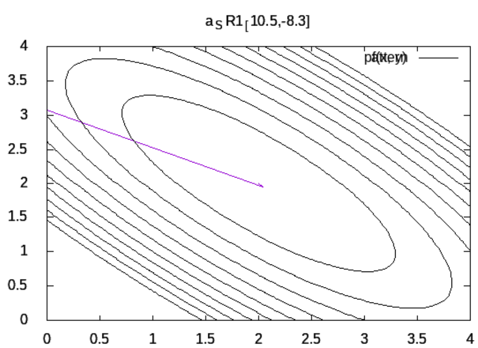
Homework 3



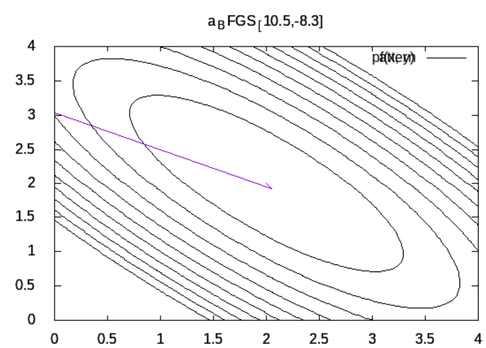
(a) SteepestDescent



(b) Newtons



(c) SR1



(d) BFGS

Figure 9: $f(x, y) = (x + 2y - 6)^2 + (2x + y - 6)^2$