

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

```
1  #ifndef __CAUCHYS__
2  #define __CAUCHYS__
3
4  #include "multivariate.h"
5  #include "multi/termination.hpp"
6
7  namespace numerical_optimization {
8
9  template<typename VectorTf>
10 class Cauchy : public Multivariate<VectorTf> {
11 public:
12     using Base = Multivariate<VectorTf>;
13     using Base::Base;
14     using Base::plot;
15     using Base::function;
```

```
16     using Base::gradient;
17     using function_t = typename Base::function_t;
18
19     // constructors
20     CauchyS(function_t f):Base(f){};
21
22     // generally works
23     VectorTf eval(const VectorTf& init=VectorTf::Random(), float e=epsilon)
24     ↪ override {
25         // 1. initialize
26         VectorTf xi = init;
27         // 2. loop
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
36             // 2. the steepest descent direction
37             VectorTf p = -1*gradient(xi)/gradient(xi).norm();
38
39             // 3. step length
40             float alpha = this->line_search_inexact(xi, p);
41
42             // 4. update gradient
43             xi = xi + alpha*p;
44         }
45         return xi;
46     };
47
48     // termination
49     template<Termination::Condition CType>
50     bool terminate(const std::vector<VectorTf>& x, float h=epsilon) const {
51         return Termination::eval<VectorTf, CType>(function, x, h);
52     }
53 };
54
55 ///////////////////////////////////////////////////////////////////
56 }/// the end of namespace numerical_optimization ///
57 ///////////////////////////////////////////////////////////////////
58 #endif //__CAUCHYS__
```

Homework 4

(b) Gradient and Hessian

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

```
1  #include <cmath>
2  #include <iostream>
3
4  #include "multivariate.h"
5
6  using namespace Eigen;
7
8  namespace numerical_optimization {
9
10 // two-variables case
11 // specialization of the vector2f case
12 template<>
13 Vector2f _gradient<Vector2f>(const std::function<float(const Vector2f&)>& f,
14   ↪ const Vector2f& x, float h) {
15     using v2 = Vector2f;
16
17     float dx = 3*f(v2(x[0]-4*h, x[1]))-32*f(v2(x[0]-3*h,
18   ↪ x[1]))+168*f(v2(x[0]-2*h, x[1]))-672*f(v2(x[0]-h, x[1]))
19     -3*f(v2(x[0]+4*h, x[1]))+32*f(v2(x[0]+3*h,
20   ↪ x[1]))-168*f(v2(x[0]+2*h, x[1]))+672*f(v2(x[0]+h, x[1]));
21     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],
22   ↪ x[1]-2*h))-672*f(v2(x[0], x[1]-h))
23     -3*f(v2(x[0], x[1]+4*h))+32*f(v2(x[0], x[1]+3*h))-168*f(v2(x[0],
24   ↪ x[1]+2*h))+672*f(v2(x[0], x[1]+h));
25
26     float inv = (1/(h*840));
27     return v2(dx, dy)*inv;
28 }
29
30 // specialization of the vector2f case
31 template<>
32 Matrix2f _hessian<Vector2f>(const std::function<float(const Vector2f&)>& f,
33   ↪ const Vector2f& x, float h) {
34     using vec2 = Vector2f;
35
36     h=0.01;
37     auto dfdx = [&](vec2 x){
38         float inv = (1/(h*840));
```

Homework 4

```
33     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]))  
34         -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]));  
35     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],  
↪ x[1]-3*h))+168*f(vec2(x[0], x[1]-2*h))-672*f(vec2(x[0], x[1]-h))  
41         -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));  
42     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 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));  
47 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));  
48 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));  
49  
50 Matrix2f m;  
51 m << dxx, dxy, dyx, dyy;  
52 float inv = 1/(4*h*h);  
53 return m*= inv;  
54 }  
55  
56 template<>  
57 Vector2d _gradient<Vector2d>(const std::function<float(const Vector2d)>& f,  
↪ const Vector2d& x, float h) {  
58     using v2 = Vector2d;  
59  
60     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]))  
61         -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]));  
62     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))  
63         -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));
```

```
64
65     double inv = (1/(h*840));
66     return v2(dx, dy)*inv;
67 }
68
69 // specialization of the vector2f case
70 template<>
71 Matrix2d _hessian<Vector2d>(const std::function<float(const Vector2d&)>& f,
72 ↪ const Vector2d& x, float h) {
73     using vec2 = Vector2d;
74     h=0.01;
75     auto dfdx = [&](vec2 x){
76         double inv = (1/(h*840));
77         double app = 3*f(vec2(x[0]-4*h, x[1]))-32*f(vec2(x[0]-3*h,
78 ↪ x[1]))+168*f(vec2(x[0]-2*h, x[1]))-672*f(vec2(x[0]-h, x[1]))
79         -3*f(vec2(x[0]+4*h, x[1]))+32*f(vec2(x[0]+3*h,
80 ↪ x[1]))-168*f(vec2(x[0]+2*h, x[1]))+672*f(vec2(x[0]+h, x[1]));
81         return app*inv;
82     };
83
84     auto dfdy = [&](vec2 x){
85         double inv = (1/(h*840));
86         double app = 3*f(vec2(x[0], x[1]-4*h))-32*f(vec2(x[0],
87 ↪ x[1]-3*h))+168*f(vec2(x[0], x[1]-2*h))-672*f(vec2(x[0], x[1]-h))
88         -3*f(vec2(x[0], x[1]+4*h))+32*f(vec2(x[0],
89 ↪ x[1]+3*h))-168*f(vec2(x[0], x[1]+2*h))+672*f(vec2(x[0], x[1]+h));
90         return app*inv;
91     };
92
93     double dxx = f(vec2(x[0]+2*h, x[1]))-2*f(vec2(x[0], x[1]))+f(vec2(x[0]-2*h,
94 ↪ x[1]));
95     double dxy = f(vec2(x[0]+h, x[1]+h))-f(vec2(x[0]-h, x[1]+h))-f(vec2(x[0]+h,
96 ↪ x[1]-h)) + f(vec2(x[0]-h, x[1]-h));
97     double dyx = f(vec2(x[0]+h, x[1]+h))-f(vec2(x[0]+h, x[1]-h))-f(vec2(x[0]-h,
98 ↪ x[1]+h)) + f(vec2(x[0]-h, x[1]-h));
99     double dyy = f(vec2(x[0], x[1]+2*h))-2*f(vec2(x[0], x[1]))+f(vec2(x[0],
100 ↪ x[1]-2*h));
101
102     Matrix2d m;
103     m << dxx, dxy, dyx, dyy;
104     double inv = 1/(4*h*h);
105     return m*= inv;
106 }
```

```
99 | ///////////////////////////////////////////  
100 | } /// the end of namespace numerical_optimization ///  
101 | ///////////////////////////////////////////
```

(c) Newton's method

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

```
1  #ifndef __NEWTONS__
2  #define __NEWTONS__
3
4  #include "multivariate.h"
5  #include "multi/termination.hpp"
6
7  namespace numerical_optimization {
8
9  template<typename VectorTf>
10 class Newtons : public Multivariate<VectorTf> {
11 public:
12     using Base = Multivariate<VectorTf>;
13     using Base::Base;
14     using Base::plot;
15     using Base::function;
16     using Base::gradient;
17     using Base::hessian;
18     using function_t = typename Base::function_t;
19
20     // constructors
21     template<Termination::Condition CType>
22     bool terminate(const std::vector<VectorTf>& x, float h=epsilon) const {
23         return Termination::eval<VectorTf, CType>(function, x, h);
24     }
25
26     // generally works
27     VectorTf eval(const VectorTf& init=VectorTf::Random(), float e=epsilon)
28     ↪ override {
29         VectorTf xi = init;
30         for(size_t i=0; i<this->iter; i++) {
31             #ifdef BUILD_WITH_PLOTTING
32                 plot.emplace_back(std::make_pair(xi, function(xi)));
33             #endif
34
35             // 1. termination
36             if(terminate<Termination::Condition::MagnitudeGradient>({xi}, e))
37             ↪ break;
38
39             // 2. gradient update
40             xi = xi - hessian(xi).inverse()*gradient(xi);
41         }
42     }
43 }
```

```
39     return xi;
40 };
41 };
42 //////////////////////////////////////
43 }/// the end of namespace numerical_optimization ///
44 //////////////////////////////////////
45 #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, α . In the case of inexact step length, depending on ρ and initial α , the step length could be computed badly. In that case, the function failed to find a optimal point.

```
1  #ifndef __QUASI_NEWTONS__
2  #define __QUASI_NEWTONS__
3
4  #include <math.h>
5  #include <cassert>
6  #include "multivariate.h"
7  #include "multi/termination.hpp"
8
9  namespace numerical_optimization {
10 namespace quasi_newtons {
11 enum Rank { SR1, BFGS, };
12 };
13
14 template<typename VectorTf, quasi_newtons::Rank RankMethod>
15 class QuasiNewtons : public Multivariate<VectorTf> {
16 public:
17     using Base = Multivariate<VectorTf>;
18     using Base::Base;
19     using Base::plot;
20     using Base::iter;
21     using Base::function;
22     using Base::gradient;
23     using function_t = typename Base::function_t;
24     using MatrixTf = Eigen::Matrix<typename VectorTf::Scalar,
    ↪ VectorTf::RowsAtCompileTime, VectorTf::RowsAtCompileTime>;
25
26     template<Termination::Condition CType>
27     bool terminate(const std::vector<VectorTf>& x, float h, float eps=epsilon) {
28         return Termination::eval<VectorTf, CType>(function, x, h, eps);
29     }
30     VectorTf eval(const VectorTf& init=VectorTf::Random(), float e=epsilon)
    ↪ override {
31
32         VectorTf xi = init;
```

Homework 4

```
33     MatrixTf Hk = MatrixTf::Identity();
34
35     size_t iteration = 0;
36     for(size_t i=0; i<this->iter; i++) {
37
38         #ifdef BUILD_WITH_PLOTTING
39             plot.emplace_back(std::make_pair(xi, function(xi)));
40         #endif
41
42         // Compute a Search Direction
43         VectorTf p = (-1*Hk*gradient(xi)).normalized();
44
45         // Compute a step length Wolfe Condition
46         double alpha = 0;
47         if constexpr (RankMethod==quasi_newtons::Rank::SR1)
48             alpha = this->line_search_inexact(xi, p, 0.99, 0.5, 3);
49         else if constexpr (RankMethod==quasi_newtons::Rank::BFGS)
50             alpha = this->line_search_inexact(xi, p, 0.8, 0.5, 3);
51
52         // Define sk and yk
53         VectorTf Sk = alpha*p;
54         VectorTf yk = gradient(xi+Sk) - gradient(xi);
55
56         // Compute Hk+1
57         if constexpr (RankMethod==quasi_newtons::Rank::SR1)
58             Hk = SR1(Hk, Sk, yk);
59         else if constexpr (RankMethod==quasi_newtons::Rank::BFGS)
60             Hk = BFGS(Hk, Sk, yk);
61
62         xi = xi - Hk*gradient(xi);
63
64         if constexpr (RankMethod==quasi_newtons::Rank::SR1) {
65             if(terminate<Termination::Condition::MagnitudeGradient>({xi},
66 ↪ 0.01)) break;
67         }
68         else if constexpr (RankMethod==quasi_newtons::Rank::BFGS) {
69             if(terminate<Termination::Condition::MagnitudeGradient>({xi},
70 ↪ 1e-8)) break;
71         }
72         return xi;
73     };
74
75     inline MatrixTf SR1(const MatrixTf& Hk, const VectorTf& Sk, const VectorTf&
76 ↪ yk) {
```

Homework 4

```
74     auto frac = 1/((Sk - Hk*yk).transpose()*yk);
75     return Hk + ((Sk-Hk*yk) * (Sk-Hk*yk).transpose())*frac;
76 }
77 inline MatrixTf BFGS(const MatrixTf& Hk, const VectorTf& Sk, const VectorTf&
↪ yk) {
78     auto pk = 1/(yk.transpose() * Sk);
79     return
↪ (MatrixTf::Identity()-pk*Sk*yk.transpose())*Hk*(MatrixTf::Identity()-pk*yk*Sk.transpose())
80 }
81 };
82 //////////////////////////////////////
83 }/// the end of namespace numerical_optimization ///
84 //////////////////////////////////////
85 #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 usually 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 dependency 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

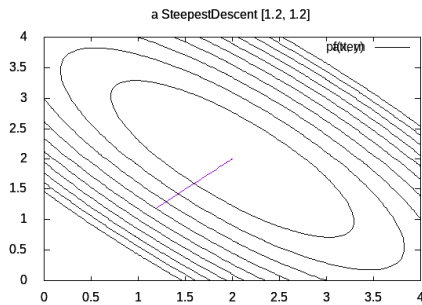
- The method of steepest descent has the slowest convergence speed
- As before said, I expected the backtracking line search gives more faster speed to search a step length.
- But it is not. Because the Quasi-Newton's methods are depending on searching a step length, when the method failed to find an adequate 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

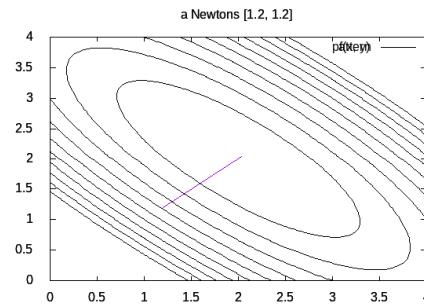
Plotting

1. Discuss for moving the approximate points

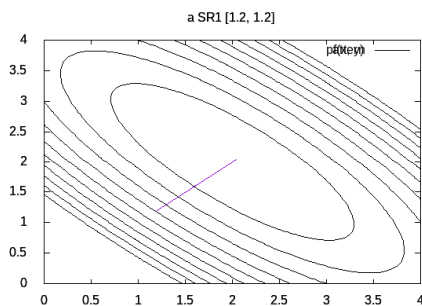
- I choose the linear function, which always converges, to observe the movement of approximate points.
- Excepted the method of steepest descent, we can observe that Newton's method and Quasi-Newton's method takes a large step length.
- Therefore, the plottings draw the sharp lines in the kind of Newton's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



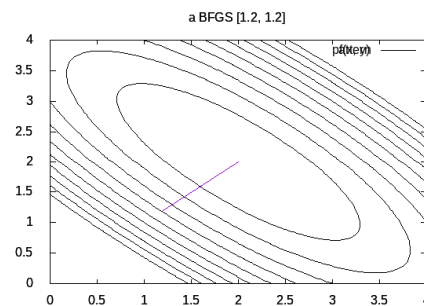
(a) SteepestDescent



(b) Newtons

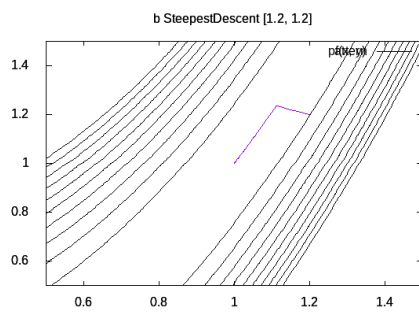


(c) SR1

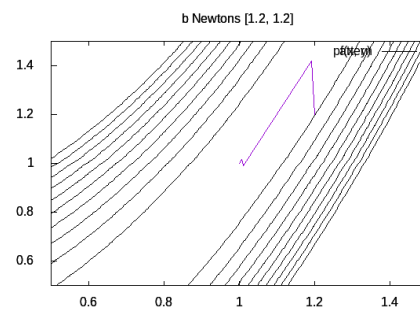


(d) BFGS

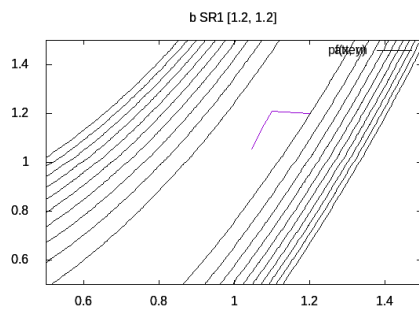
Figure 1: $f(x, y) = (x + 2y - 6)^2 + (2x + y - 6)^2$



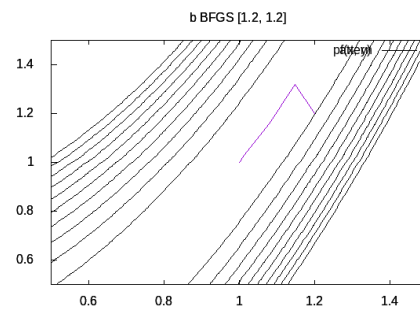
(a) SteepestDescent



(b) Newtons

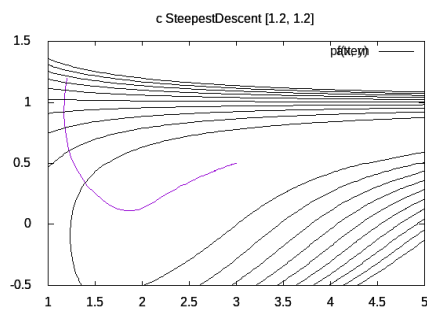


(c) SR1

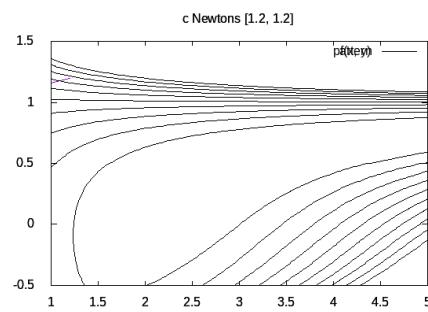


(d) BFGS

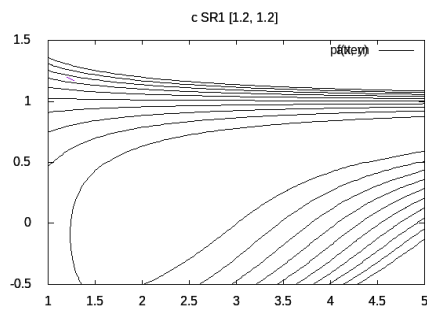
Figure 2: $(x, y) = 50 * (y - x^2)^2 + (1 - x)^2$



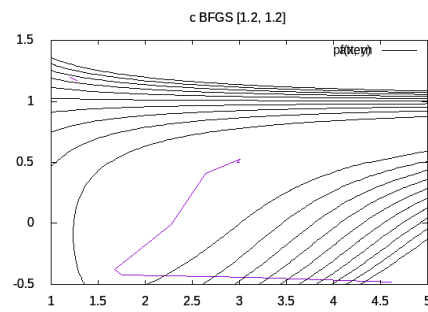
(a) SteepestDescent



(b) Newtons

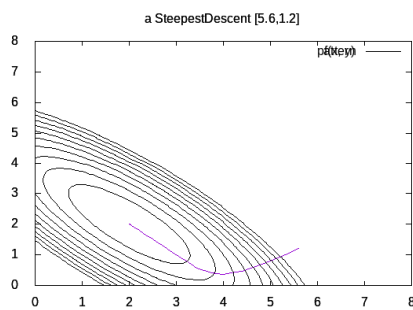


(c) SR1

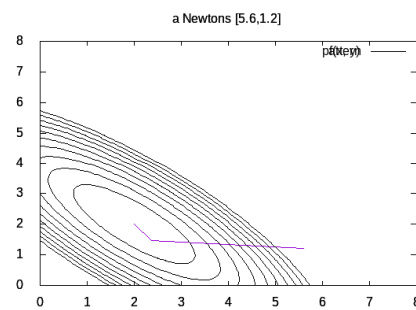


(d) BFGS

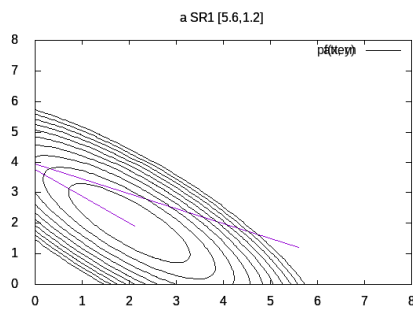
Figure 3: $f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$



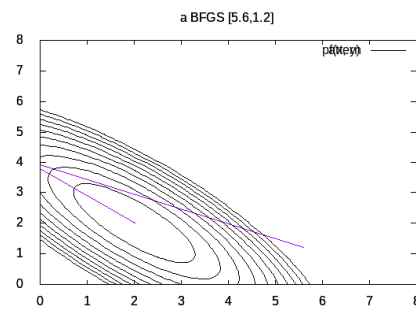
(a) SteepestDescent



(b) Newtons

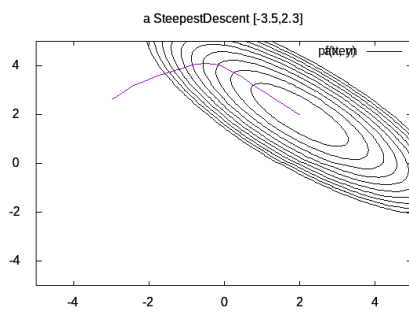


(c) SR1

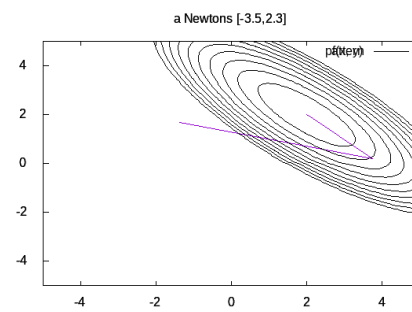


(d) BFGS

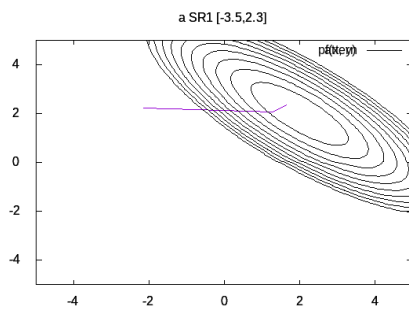
Figure 4: Moving comparison[5.6,1.2]: $f(x,y) = (x + 2y - 6)^2 + (2x + y - 6)^2$



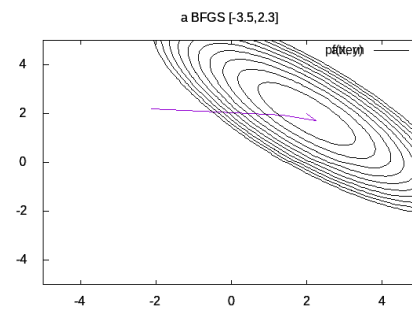
(a) SteepestDescent



(b) Newtons

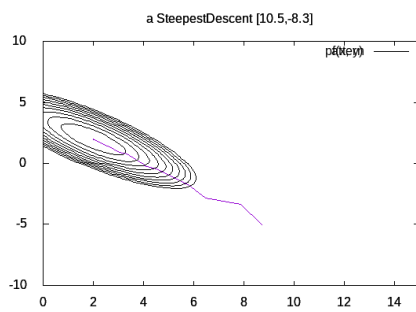


(c) SR1

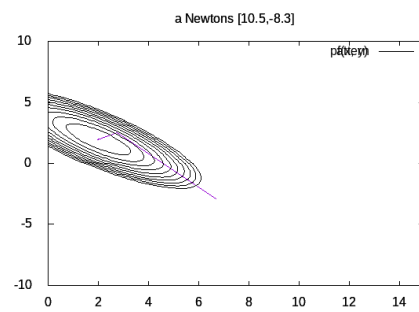


(d) BFGS

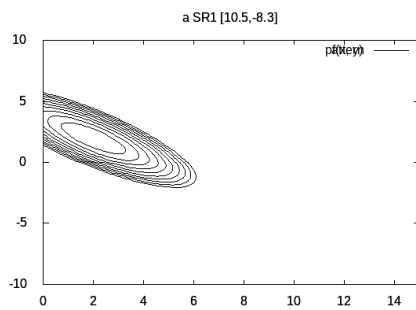
Figure 5: Moving comparison $[-3.5,2.3]$ $f(x,y) = (x+2y-6)^2 + (2x+y-6)^2$



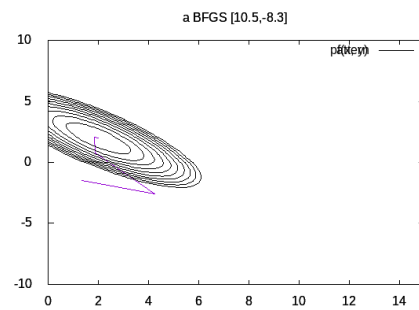
(a) SteepestDescent



(b) Newtons



(c) SR1



(d) BFGS

Figure 6: Moving comparison[10.5,-8.3] $f(x, y) = (x + 2y - 6)^2 + (2x + y - 6)^2$