Homework #4

Implement Gradient based Methods & Disscusion

GIST AI

20231126 황산하

*코드는 Python으로 작성했고 Report와 함께 Zip으로 압축하여 제출하였습니다.

INDEX

1. Comparison and Result

이번 과제에서도 과제 3에서 주어졌던 함수와 같은 함수들이 주어졌다. 이번 과제는 다변수 함수들에 대해서 Gradient based method인 Steepest Descent, Newton, Quasi Newton (SR1, BFGS)를 구현하여 Local minimum을 찾고, 비교하는게 목표이다.

$$F_1(x,y) = (x+3y-5)^2 + (3x+y-7)^2$$

$$F_2(x,y) = 50(x-y^2)^2 + (1-y)^2$$

$$F_3(x,y) = (1.5-x+xy)^2 + (2.25-x+xy^2)^2 + (2.625-x+xy^3)^2$$

이번 과제도 엄밀한 비교를 위해서 최대한 하이퍼파라미터는 고정을 하고 실험을 진행했다. 과제에서 주어진 (1.2, 1.2)에서 시작했을 때 어떤 차이가 있는지 비교를 했고 스텝사이즈 α를 찾는 방법을 각각 Bisection method, Inexact Search with Wolfe Condition 을 구현하여 비교를 진행했다. 결과표들은 (1.2, 1.2)에서 시작됐을 경우에만 정리했다.

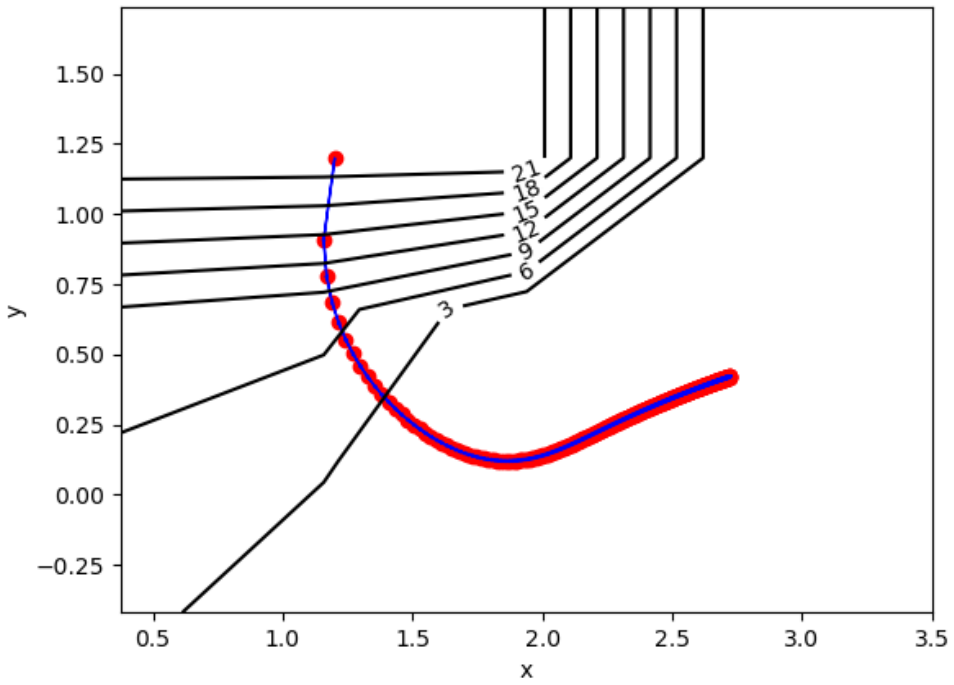
[표 1 Steepest Descent 결과표]

Steepest Descent	(x, y)	Iter	Convergence	f(x,y)
F1	(1.988, 1.012)	89 iter (0.17 sec)	Yes	0.00111
F2	(0.741, 0.861)	268 iter (0.52 sec)	Yes	0.0191
F3	(2.720, 0.421)	402 iter (0.87 sec)	Yes	0.017

1) Steepest Descent

. Steepest Descent 방법은 주어진 방법 중에서 가장 느리지만 한 번의 예외도 없이 수렴 지점을 찾을 수 있었던 방법이다. 현재 딥러닝에서 가장 잘 알려진 경사하강법의 시초격이라고 볼 수 있을 거 같은데, -gradient 방향으로만 방향이 진행되기 때문에 수렴을 보장해주어 좋았다. 이 부분이 수업시간에 배웠었던 왜 수렴을 보장하는지 수식적으로 증명했던 내용과 일치하는 부분이었다. 다른 방법들과 달리 초기 점을 어디로 잡는지에 상관없이 대부분 local minimum으로 수렴했고, 하지만 다른 방법들에 비해서 꽤 많은 iteration을 필요로 했기 때문에 결과를 받아 보기 위해서는 기다리는 시간이 조금 길었다는 것이 단점이었다. 특히 결과표에서 볼 수 있듯이 찾은 함수 값도 거의 0에 수렴했기 때문에 다른 방법들에 비해서 함수가 간단한 경우에는 매우 강력한 방법이라는 생각이 들었다.

하지만 이 방법의 단점을 하나 뽑자면, Step size (α)에 매우 민감하다는 점이었다. (BFGS, SRQ에서도 해당 됨) 이 부분은 2. discussion에서 간단하게 생각을 정리해서 말해볼 예정이지만 어떤 step size를 고르느냐에 따라서 처음 구현했을 때는 아예 학습을 진행하지못하고 초기점에 머무는 현상도 보였다. 아래 그림 1은 f3함수에서 어떻게 수렴하는지 그린 contour plot이다



[그림 1] F3 함수에 대한 SD 방법의 Contour Plot,

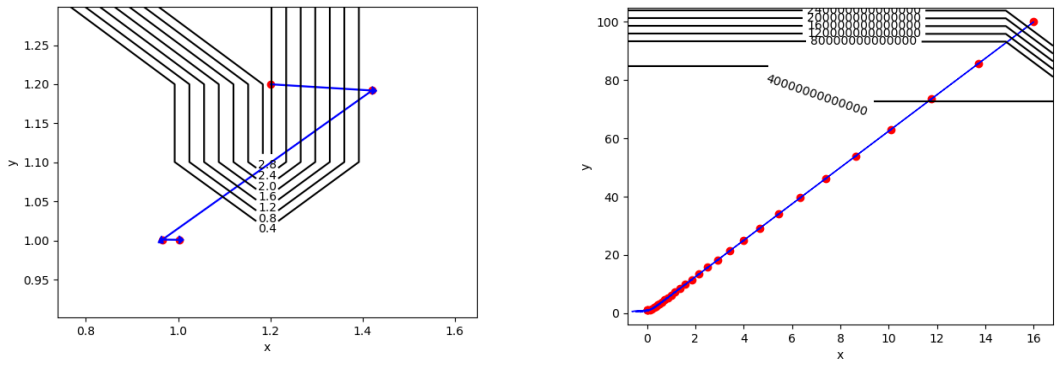
[표 2 Newton Method 결과표]

Newton Method	(x, y)	Iter (Time)	Convergence	f(x,y)
F1	(2.0, 0.999)	4 iter (0.01sec)	Yes	7.8886e-31
F2	(0.9999, 0.9999)	4 iter (0.007sec)	Yes	4.208e-11
F3	(1.37e-17, 1.000)	4 iter (0.01sec)	Yes	14.2031

2) Newton Method

Newton 방법은 이전 과제에서도 구현했듯이 수렴속도가 매우 빠른 알고리즘이다. 이전 결과와 마찬가지로 Newton 방법은 빠르게 local minimum으로 수렴했는데 주어진 1.2, 1.2에서는 다행히 발산하지는 않았다. 발산하는 경우를 찾기 위해 여러가지 초기 x,y를 바꿔봤지만 찾지는 못했던 것 같다. 아무리 x,y를 크게 줘도 빠르게 local minimum을 찾는 걸 보고 괜히 newton method가 여러 알고리즘에 이용되는게 아니구나하는 생각도 들었다.

Newton method를 구현할 때 주의해야할 점은 Second Derivative가 너무 작을 경우 발산할 수 있다는 점이다. 그래서 내가 구현할 때는 너무 작은 Second Derivative가 나올경우를 대비해서 적당히 작은 epsilon 값을 추가해서 구현을 진행해서 nan값이 나오는 것을 방지했다.



[그림 2] F1 함수에 대한 newton 방법의 Contour Plot, (우)는 x,y를 좀 다르게 줘본 결과이다.

[표 3 SR1 Method 결과표]

SR1	(x, y)	Iter (Time)	Convergence	f(x,y)
F1	(1.2448, 1.2448)	6 iter (0.01sec)	Yes	4.0829
F2	발산	4 iter (0.007sec)	No	발산
F3	(0.2718, 0.2718)	31 iter (0.07sec)	Yes	11.2516

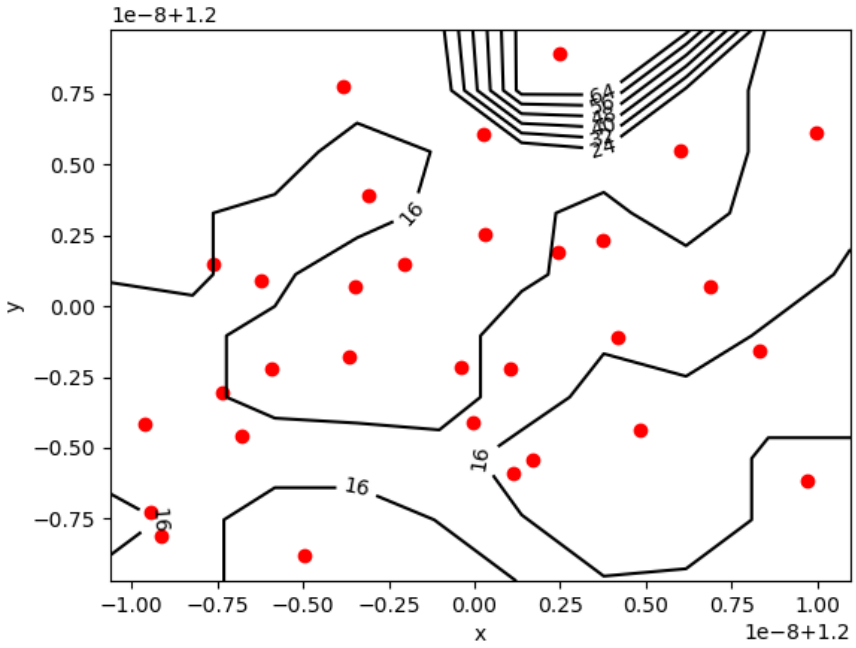
3) Quasi Newton – SR1

Quasi Newton 방법은 Hessian을 추정해서 값을 업데이트하는 방법인데, 그 중에서도 SR1은 rank1 update를 하는 방법이다. 구현해보기 전에는 역행렬 구하는 비용이 많이 들기 때문에 이를 줄이고자 Quasi newton을 하는 줄 알았는데, Hessian을 구하고 value를 업데이트 하려면 추정된 행렬을 역행렬을 구해서 업데이트를 진행해야한다는 점을 알았다. 하지만 이렇게 진행할 경우 추정된 행렬이 singular여서 역행렬을 구하기 어려울 때가 발생하는데, 이를 방지하기 위해서 아예 역행렬 자체를 추정해버리는 방법을 사용했다. 이를 통해 역행렬을 초반에 Identity 행렬로 준 행렬의 역행렬만 구하고 나머지는 추정값으로 이용할 수 있었다.

SR1은 결과에서 볼 수 있듯 수렴되면 아주 빠르게 수렴할 수 있다. 이는 Newton 방법이랑 비슷하다고 볼 수 있고 Newton 방법에 비해서 hessian을 구하는 비용이 들지 않아 조금은 computation 부담이 줄 수 있다는 장점이 있다.

그럼에도 불구하고 이 방법 역시도 step size에 영향을 크게 받는 다는 걸 인지했다. F3 함수의 경우 stepsize에 따라서 업데이트를 아예 안하거나 발산하거나 하는 경우가 비일비재했다. 차라리 exact line search를 이용해서 정확한 stepsize를 구한다면 발산하는 경우를 줄일 수 있지 않을지 궁금했는데 구현 문제일 수도 있지만 일단 내가 구현한 코드에서는 알고리즘이 step size에 민감

한 모습을 계속 보였다.



[그림 3] F3 함수에 대한 SR1 방법의 Contour Plot, 결과값이 작게 나와서 contour plot 을 그릴 때 arrow가 상대적으로 너무 컸기 때문에 점만 찍어 보았다.

[표 4 BFGS Method 결과표]

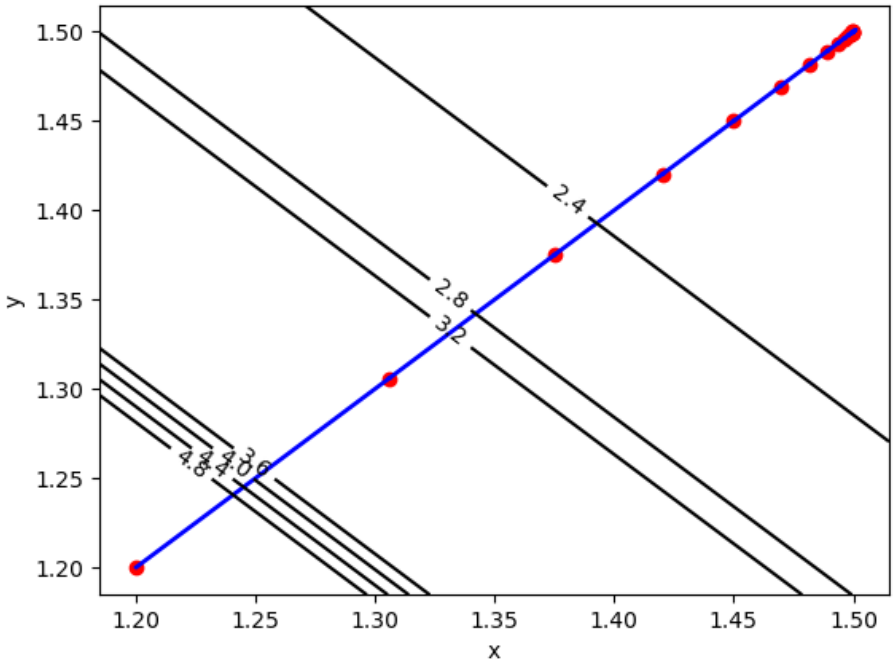
SR1	(x, y)	Iter (Time)	Convergence	f(x,y)
F1	(1.499, 1.499)	18 iter (0.01sec)	Yes	2
F2	(1, 1)	13 iter (0.03sec)	Yes	1.13e-06
F3	(0.2718, 0.2718)	10 iter (0.07sec)	Yes	9.71763

4) Quasi Newton - BFGS

BFGS도 역시 Quasi Newon의 한 방법으로 SR1방법과 다르게 rank2업데이트를 하는 방식이다. Hessian 행렬을 추정하는 방식만 조금 다르기 때문에 나머지 알고리즘은 똑같이 구현할 수 있었다. 이번에도 역시 역행렬을 직접 추정하는 방식을 사용했고 SR1에 비해서 좀 더 안정적이라는 느낌을 받았다. 결과표에서도 볼 수 있듯이, iteration은 다소 길어졌지만 안정적으로 수렴을 했고, 찾은 값들 역시 0에 가깝거나 크게 크지 않은 값들로 수렴을 했다.

다만 주목할만한 점은 구현 문제일 가능성이 크지만 업데이트된 x, y를 보면 값이 정확하게 일치

하는 것을 알 수 있다. 확인해보니 추정된 alpha값 역시 놀랍게 똑같았는데 line search 문제라기에는 Steepest 방법에서도 똑 같은 line search를 이용했고, value를 업데이트하는 과정에서 문제가 발생한 거 같은데, x와 y를 나눠서 각각 업데이트하게 되면 발산하게 되어 과제물에는 이렇게 제출하게 되었다. 그렇다고 해서 지금 추정된 local minimum의 값이 합리적이지 않은 것도 아니라서 어떤 이유로 같게 나오는지 궁금했지만 답을 찾지는 못했다.



[그림 4] F1 함수에 대한 BFGS 방법의 Contour Plot,

2. Disccusion

1) Step Size

Step size를 이용하지 않는 Newton 방법을 제외하고 나머지 방법에서는 step size에 매우 민감한 결과를 보였다. 물론 민감한게 당연하지만 그래서 어떻게 step size를 찾느냐? 하는 문제는 이번 과제를 진행하면서 지속 되었다. 수업시간에 배운 내용으로는 inexact search와 exact search를 비교하며 굳이 exact search를 할 필요없이 그 비용을 줄여 local minimum을 찾으려고 했다. 그래서 이번 과제를 진행할 때 처음엔 단순하게 Bisection이 구현하기 쉬우니 Bisection으로 step size를 찾겠다는 생각을 했으나, 의외로 bisection에서는 step size가 오른쪽 boundary로만 update가 되어 값이 오히려 발산하는 결과를 얻었다.

두번째는 교재 pdf에 나와있는 inexact search를 이용했으나, 이번에도 좋은 결과를 얻지 못하였다. 이번에는 반대로 step size가 너무 작아지는 문제가 생겨 값이 update가 되지 않아 종료조건에 걸리는 바람에 1iteration에 종료되거나 종료조건을 해제하니 max iteration에 갈때까지 업데이트 없이 알고리즘이 진행되었다.

마지막으로는 Wolfe condition을 넣어서 해당되는 값 안에 있으면 그 값을 step size로 이용하려고 했다. 이 방법이 가장 유효했는데, 너무 작지도 크지도 않은 step size를 발견함으로써 대부분의 알고리즘과 함수에서 수렴하는 결과를 이끌었다. 하지만 그럼에도 고민되는 부분이 알고리즘이 너무 민감해서 조금이라도 다른 값을 주거나, wolfe condition을 만족하더라도 발산하는 경우 (SR1 –f2함수)에는 step size를 어떻게 찾아줘야할지 가장 일반적인 상황에서도 적용할 수 있는 방법이 뭐일지 고민해볼만한 지점이라고 생각했다.

2) Terminal Condition

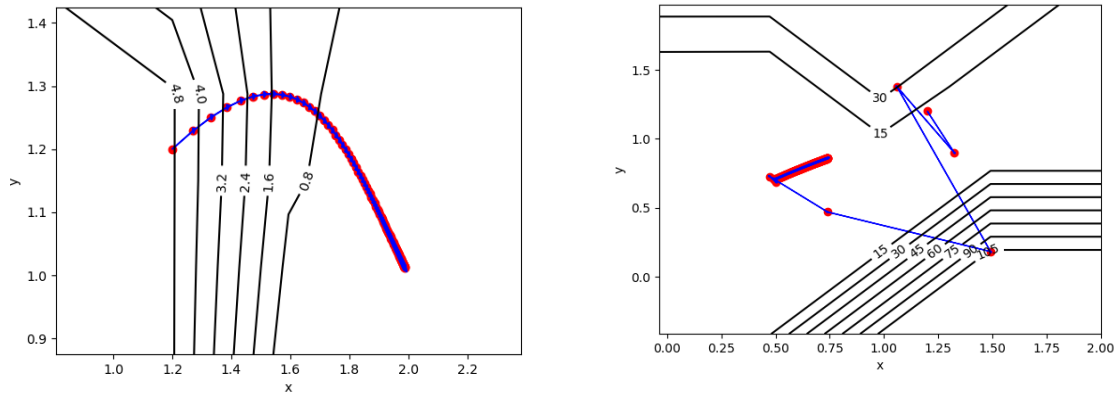
종료조건으로는 크게 두가지를 사용했는데 초반에 사용했던 것은 함수의 gradient를 구해서 이것의 norm을 이용하는 방식이었다. Norm이 작으면 gradient가 0에 가까워서 더 이상 업데이트가 일어나지 않는다고 판단하고 종료하는게 좋겠다 하는 생각에 제안한 종료조건인데, 생각보다는 잘 되지 않았다. 오히려 함수가 더 복잡한 경우에는 굳이 norm을 구하는 cost때문에 병목이 걸릴 수도 있다는 생각이 들었다.

두번째는 이전 과제에서도 계속 이용했던 이전값과 현재값을 비교하는 방식이었다. 이 방법은 개인적으로 이용하기가 싫었는데 왜냐하면 search하다보면 값이 작아졌다가 커졌다가 하기 때문에 그런 과정에서 커졌다가도 다시 작아지는 방향으로 간다면 오히려 global minimum에 근접하는 local minimum을 찾을 수도 있다고 생각해서 어떻게 하면 크게 발산하는 경우 말고 종료조건을 잘 줄 수 있을지에 대한 고민을 했다. 크게 좋은 방법이 떠오르지 않아서 처음 방법인 norm을 이용하는 방법과 이전보다 지금 값이 크고, 차이가 1 이상나면 종료하는 조건을 통해 나름 완벽하진 않지만 안정적인 종료조건을 만들 수 있었던 것 같다.

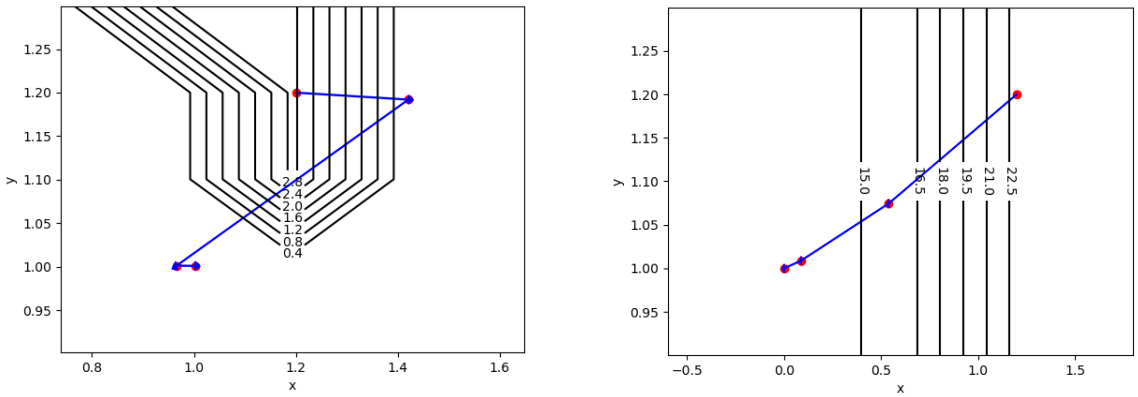
3. Appendix A : Code

Code는 Python으로 작성했으며 제출한 폴더를 열어보면, mothod.py, result.py, ultils.py로 구성되어 있다. 가독성을 위해서 이전과제에서는 utils.py를 작성하지 않았지만 보조적으로 이용할 step size search 방법이라던지 hessian matrix와 gradient를 계산하는 함수를 작성해서 utils.py를 만들었다. 따라서 method.py에는 이번 과제의 핵심 알고리즘인 Steepest method, newton 등이 작성되어있다. Result.py에서는 과제에서 주어진 함수의 local minimum을 찾은 결과를 확인할 수 있는데 쉽게 python result.py의 스크립트를 입력해서 확인할 수 있다. Main 함수에서 수정할 수 있는 파라미터는 epsilon, max iteration, 함수 종류, 초기 값이다.

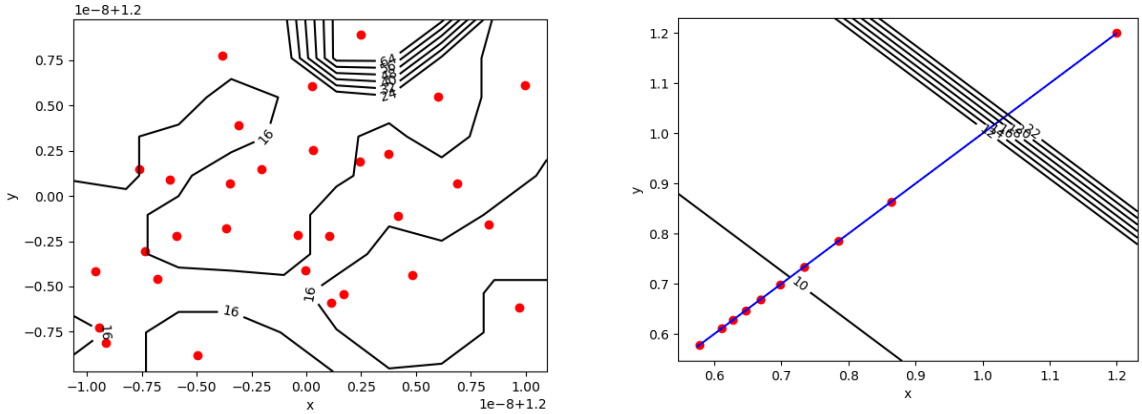
4. Appendix B : Some Result



[그림 5,6] SD 방법으로 search한 그림, (좌)f1 (우)f2



[그림 7,8] Newton 방법으로 search한 그림, (좌)f2 (우)f3, f1 함수는 1 iteration만에 최적점을 찾아서 그래프를 그릴 수 있을 정도로 많이 값이 주어지지 않았다.



[그림 9,10] Quasi-Newton 방법으로 search한 그림, (좌)f3 with SR1 (우)f3 with BFGS