CS6491 Final Report

Topic: 5) Seeing is Believing: Visualizing Convex Optimization
Name: Zhengxiang ZHOU
StudentID: 55817209

Introduction

Convex optimization has been widely used in different fields. Nevertheless, in most of the cases, we can only observe the numerical optimization process, which is not intuitive and requires domain knowledge to understand the algorithms. Motivated by this issue, I design and develop a software to visualize the optimization process for 2-dimension functions. The optimization process is represented as the moving points on the 2-D surface. For a user, he can select an initial point and observe how the point converges to the optimum.

Requirements

There are some settings for the elements in the program:

1. A point is a 2-D vector for the input of one iteration of the optimization algorithm.
2. The default error for optimization algorithms is 1e-6. Thus functions that are too sensitive to small values may not perform the best in the default settings.
3. To save computation power, some graphic elements are optimized with some decrement of elaboration.

The requirements for a visual representation of the 2-D optimization process is listed below:

1. Each iteration in the optimization algorithm is represented by a line joining the two points before and after the iteration.
2. A contour line (different colors of regions) is needed to indicate how close a region is to the optimum.
3. Multiple optimization methods should be supported for comparisons.
4. For the ease of use, analyzed functions should be able to change without modifying the codes.
5. Some numerical data should also be available to better understand the algorithms.

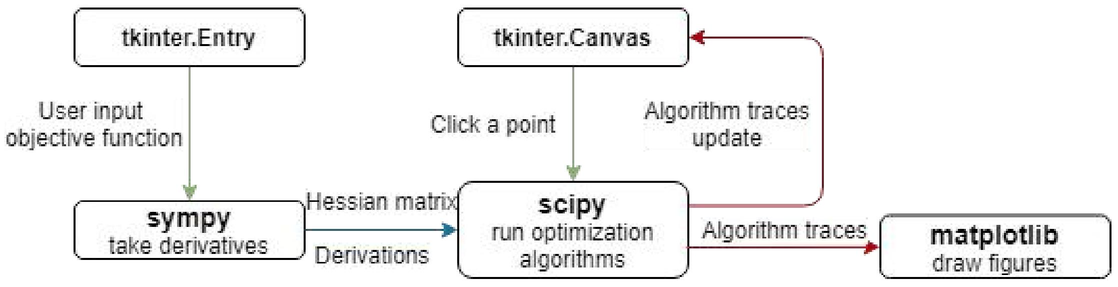
Implementation

The programming language I use is Python, which contains a large number of easy-to-use libraries. The following are the details of the implementation.

Dependencies:

I use **numpy** for number storage and simple calculations. **tkinter** for GUI support since it is light, **matplotlib** to create figures, and **sympy** and **scipy** to run the optimization algorithms.

Design:



The program receives the user-input objective function by an *Entry* widget. Then it differentiates the function to get the hessian matrix and the derivatives, which might be needed by the optimization algorithms.

At the beginning of the program, the user can modify the precision of the 2-D surface, given some functions may be sensitive to small values. Besides, he can check different algorithms so that the program only shows the traces of desired algorithms.

After the user clicks a point on the 2-D surface, the desired algorithms will be called and iterate for only 5-10 rounds. The traces will be recorded and drawn on the 2-D surface. For those algorithms that don’t terminate, the program will repeat the previous steps until all algorithms terminate and are drawn. This avoids that the program does not respond for a long time if some algorithms cannot terminate in a few rounds. Every 5-10 rounds, the program will be forced to give a response.

Finally, the errors vs iterations figure will be shown in the widget when the optimization algorithms terminate, which can better help understand the algorithms.

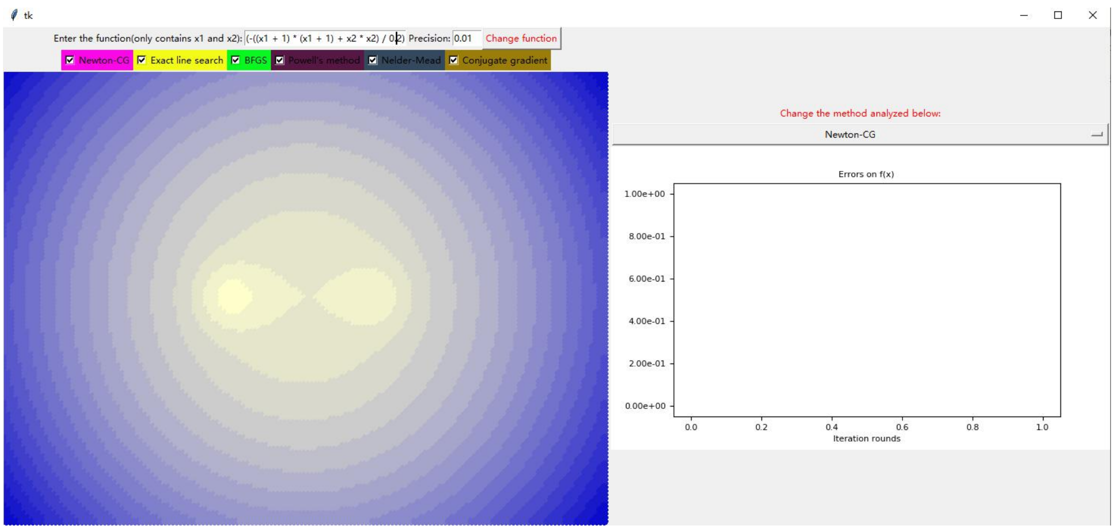
Illustrations

In the illustration, the provided objective function is

$$x_1^2 + x_2^2 - 2e^{-\frac{(x_1 - 1)^2 + x_2^2}{0.2}} - 3e^{-\frac{(x_1 + 1)^2 + x_2^2}{0.2}}$$

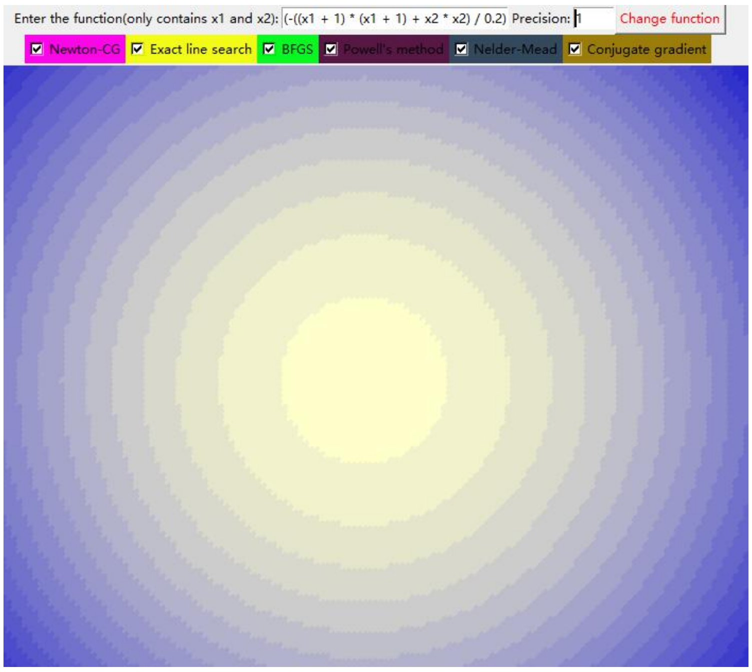
which has two local optimums and shapes like two bowls.

Overview:


Enter the function(only contains x1 and x2): (-((x1 + 1) * (x1 + 1) + x2 * x2) / 0.2) Precision: 0.01

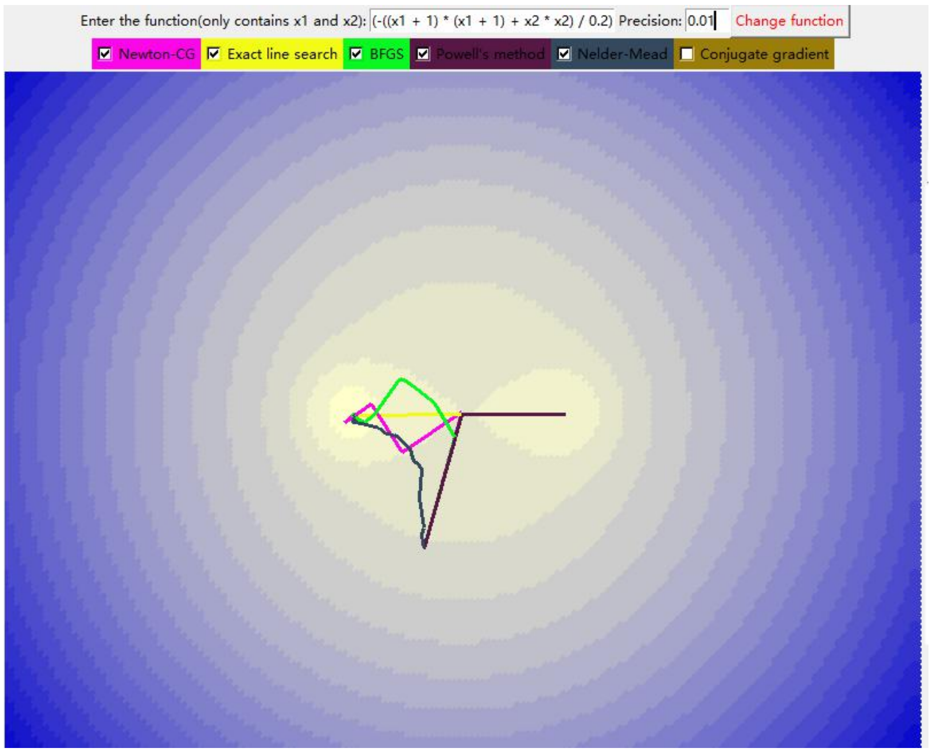
Change the method analyzed below: Newton-CG

As shown in the figure above, objective functions can be modified in the entry widgets as well as the precision. As aforementioned, precision should be set based on the function, thus I leave it for the user himself. For example, in the illustration, if I change the precision to 1, which means the program will ignore the fractional parts of points, the contour line will behave like this:

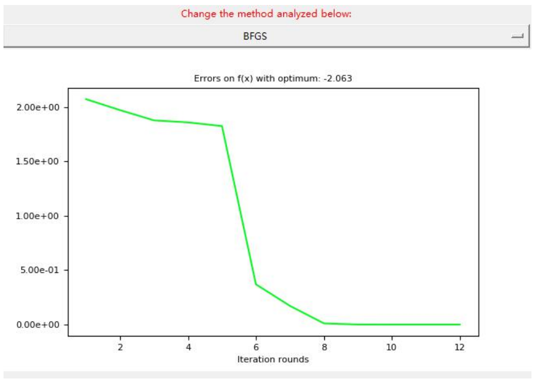

Enter the function(only contains x1 and x2): (-((x1 + 1) * (x1 + 1) + x2 * x2) / 0.2) Precision: 1

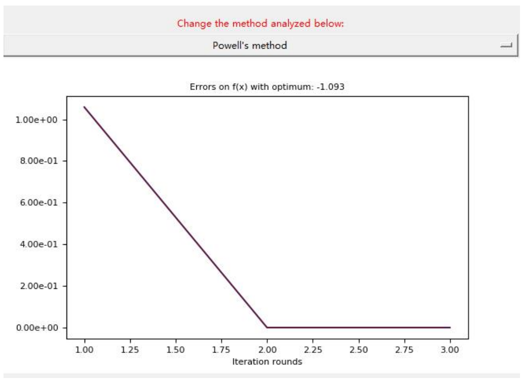
Because the two local optimums of the objective function is too close, only the difference in fractional parts can tell the difference between the two optimums. In the figure above, the difference is ignored due to a too large precision.

Currently there are 6 supported optimization algorithms: Newton method, exact line search, BFGS(Broyden–Fletcher–Goldfarb–Shanno algorithm), Powell's method, Nelder Mead, and Conjugate gradient. The user can check the option boxes to only observe the traces of their desired algorithms.


Enter the function(only contains x1 and x2): (-((x1 + 1) * (x1 + 1) + x2 * x2) / 0.2) Precision: 0.01
Change function
☑ Newton-CG ☑ Exact line search ☑ BFGS ☑ Powell's method ☑ Nelder-Mead ☐ Conjugate gradient

The corresponding error analysis can be viewed in the figures on the right.





Code analysis

To reduce the redundancy in the code and make it easy to redevelop, I use several general variables (dictionary) to write the code logic.

```
ALL_METHODS = [NEWTON, ELS, BFGS, POWELL, NELDERMEAD, CG]

for method_id in unconverged_list:
            res, TRACES[method_id] = methods[method_id](current_point[m
ethod_id], iteround = cal_round)
            terminate_flag[method_id] = res.success
            if method_id not in TOTAL_TRACES:
                TOTAL_TRACES[method_id] = []
            TOTAL_TRACES[method_id].extend(TRACES[method_id][1:])
            current_point[method_id] = TRACES[method_id][-1].copy()
```

For a developer who wants to add new algorithms to the program, he only needs to modify some parts of the config and write a function which receive the initial point and return the traces. Generally, adding a new algorithm requires about 20 lines codes, which is quite simple and scalable.

Conclusion

The visual representation of optimization process is a good perspective for people to better understand the algorithms and make comparisons between different algorithms. I have built a program which visualize the traces of 2-D optimizations. In fact, functions that have less or equal to 3 dimensions can be visualized. As the dimensions rise, the difficulty increases. 2-D is a tradeoff between difficulty and usability. The current program is just a demo, as the graphic design is not perfect and there can be more complex analysis on the optimization algorithms, such as computation time. Moreover, the design can be extended to the visualization of 3-D function optimizations.

Reference links

Interactive Visualization of Optimization Algorithms in Deep Learning:
https://emiliendupont.github.io/2018/01/24/optimization-visualization

Scipy lecture notes: 2.7. Mathematical optimization: finding minima of functions:
https://scipy-lectures.org/advanced/mathematical_optimization/auto_examples/plot_gradient_descent.html

tkinter documentation: https://docs.python.org/3/library/tk.html