CS2601 Linear and Convex Optimization Project Report – Implementing the Water-filling Algorithm

Xuankun Yang 523030910248

December 28, 2024

Contents

1 Introduction **1**

2 Obectives of the Experiment **2**

3 Experimental Process **3**

3.1 Binary Search 4

3.2 Gradient Descent 5

3.3 Newton Method 6

4 Results and Analysis **7**

4.1 Horizontal Comparision 7

4.1.1 Fix Dimension $n = 10$ 8

4.1.2 Discussion on Dimension n 10

4.2 Vertical Comparision 12

4.2.1 Binary Search 12

4.2.2 Gradient Descent Method 13

4.2.3 Newton Method 14

4.3 Visualization of Iteration Process 14

5 Conclusion **16**

1 Introduction

In this project, we concentrate on *The Water-filling Problem.* I implemented 3 different methods in my experiments.

In this report, we have 3 parts: *Obectives of the Experiment*, *Experimental Process* and *Results and Analysis.*

2 Obectives of the Experiment

The convex optimization problem:

$$\text{minimize } -\sum_{i=1}^{n} \log(\alpha_i + x_i)$$

$$\text{subject to } x \succeq 0, \mathbf{1}^T x = 1, \alpha_i > 0$$

Introducing Lagrange multipliers $\lambda^\star \in \mathbf{R}^n$ for the inequality constraints $x^\star \succeq 0$, and a multiplier $\nu^\star \in \mathbf{R}$ for the equality constraint $\mathbf{1}^T x = 1$, we obtain the *KKT* conditions:

$$x^\star \succeq 0, \mathbf{1}^T x^\star = 1, \lambda^\star \succeq 0, \lambda_i^\star x_i^\star = 0, i = 1, \ldots, n,$$

$$-\frac{1}{\alpha_i + x_i^\star} - \lambda_i^\star + \nu^\star = 0, i = 1, \ldots, n.$$

We can directly solve these equations to find $x^\star, \lambda^\star$ and $\nu^\star$.
Noting that $\lambda^\star$ acts as a slack variable in the last equation, so it can be eliminated, leaving

$$x^\star \succeq 0, \mathbf{1}^T x^\star = 1, (\nu^\star - \frac{1}{\alpha_i + x_i^\star}) x_i^\star = 0, i = 1, \ldots, n,$$

$$\nu^\star \geqslant \frac{1}{\alpha_i + x_i^\star}, i = 1, \ldots, n.$$

Next, we make a discussion on $\nu^\star$

1. If $\nu^\star < \frac{1}{\alpha_i}$, the last condition can hold only if $x_i^\star > 0$, according to the third condition, $\nu^\star = \frac{1}{\alpha_i + x_i^\star}$
   Solving for $x_i^\star$, we conclude that $x_i^\star = \frac{1}{\nu^\star} - \alpha_i$, if $\nu^\star < \frac{1}{\alpha_i}$
2. If $\nu^\star \geqslant \frac{1}{\alpha_i}$, according to the third condition, $x_i^\star = 0$ must hold
   Therefore, $x_i^\star = 0$, if $\nu^\star \geqslant \frac{1}{\alpha_i}$

Thus, we have

$$x_i^\star = \begin{cases} \frac{1}{\nu^\star} - \alpha_i, & \nu^\star < \frac{1}{\alpha_i} \\ 0, & \nu^\star \geqslant \frac{1}{\alpha_i} \end{cases}$$

or, put more simply, $x_i^\star = \max\{0, \frac{1}{\nu^\star} - \alpha_i\}$.
Substituting this expression for $x_i^\star$ into the condition $\mathbf{1}^T x^\star = 1$, we obtain

$$\sum_{i=1}^{n} \max\{0, \frac{1}{\nu^\star} - \alpha_i\} = 1$$

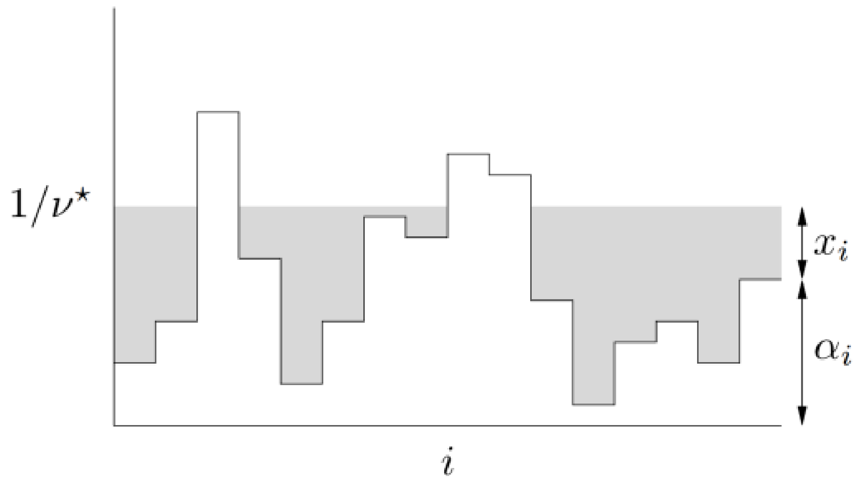

Figure 1: Visualization

The lefthand side is a piecewise-linear increasing function of $\frac{1}{\nu^\star}$, with breakpoint at α_i, so the equation has a unique solution which is readily determined.

This solution method is called water-filling for the following reason. We think of i as the ground level above patch i, and then flood the region with water to a depth $\frac{1}{\nu}$, as illustrated in figure 1. The total amount of water used is then $\sum_{i=1}^{n} \max\{0, \frac{1}{\nu^\star} - \alpha_i\}$. We then increase the flood level until we have used a total amount of water equal to one. The depth of water above patch i is then the optimal value $x_i^\star$.

Therefore, the objective of our experiments is to implemente some algorithm and find the $\nu^\star$, and keep $\sum_{i=1}^{n} \max\{0, \frac{1}{\nu^\star} - \alpha_i\} = 1$, so that we can make our target $-\sum_{i=1}^{n} \log{(\alpha_i + x_i)}$ minimized.

3 Experimental Process

In this experiment, I implemented 3 different methodology.

3.1 Binary Search

From the analysis above, $\sum_{i=1}^{n} \max\{0, \frac{1}{\nu^\star} - \alpha_i\}$ is a piecewise-linear increasing function of $\frac{1}{\nu^\star}$, with breakpoint at α_i, so the equation $\sum_{i=1}^{n} \max\{0, \frac{1}{\nu^\star} - \alpha_i\} = 1$ has a unique solution which is readily determined.
Naturally, I come up with the *Binary Search*.
The python code of the key function *fill water* is as follows:

```
def fill_water(alpha, total_water, precision):
    lower_bound = 1/(total_water + max(alpha))
    upper_bound = 1/min(alpha)
    iteration = 0

    while upper_bound - lower_bound > precision:
        nu = (lower_bound + upper_bound)/2
        x = np.maximum(0, 1/nu - alpha)
        water_sum = np.sum(x)
        iteration += 1

        if water_sum > total_water:
            lower_bound = nu
        else:
            upper_bound = nu

    nu_opt = (upper_bound + lower_bound)/2
    x_opt = np.maximum(0, 1/nu_opt - alpha)
    print(f"Iterations : {iteration}")

    return np.array(x_opt), iteration
```

1. Firstly, consider the upper bound of ν, *i.e.* consider the lower bound of the level $\frac{1}{\nu}$
$$\frac{1}{\nu} > \min \alpha_i, i = 1, \ldots, n$$
must holds.
Thus the upper bound of ν could be $\frac{1}{\min \alpha_i}$.

2. As for the lower bound of ν, consider the upper bound of the level $\frac{1}{\nu}$
$$\frac{1}{\nu} < 1 + \max \alpha_i, i = 1, \ldots, n$$
must holds.
Thus the lower bound of ν could be $\frac{1}{1+\max \alpha_i}$.

3. Next, we set a precision gap, if upper bound minus lower bound is greater than our precision gap, then update ν using the middle of upper bound and upper bound and update x according to $x_i = \max\{0, \frac{1}{\nu} - \alpha_i\}$.

4. To ensure $\mathbf{1}^T x = 1$,

   - If $\sum_{i=1}^{n} x_i >$ total water (here we take 1), which means the level is too high, *i.e.* ν is too small, then we increase lower bound to ν
   - if $\sum_{i=1}^{n} x_i <$ total water, which means the level is too low, *i.e.* ν is too big, then we decrease upper bound to ν

5. Continue doing this until the precision gap is satisfied, and we got the optimized x and optimized ν.

3.2 Gradient Descent

The python code of the key function *fill water* and *backtracking line search* are as follows:

```
def fill_water(alpha, total_water, precision, max_iter,
                                    use_line_search = True, lr = 1e-
                                    3):
    n = len(alpha)
    x = np.ones((n, 1))*(total_water / n)

    for t in track(range(max_iter), description = " ..."):
        nu = 1 / np.min(alpha + x)
        x = np.maximum(0, 1 / nu - alpha)
        grad = -1 / (alpha + x)
        if use_line_search:
            step = backtracking_line_search(x, -grad,
                                                total_water)
        else:
            step = lr
        x = x - step * grad

    return np.array(x)
```

```
def backtracking_line_search(x, direction ,total_water, beta = 0
                                    .5, max_iter = 1000):
    step = 1

    for i in range(max_iter):
        if np.sum(x + step * direction) > total_water:
            step = step * beta
        else:
```

```
            break

    return step
```

1. Firstly, initialize x averagely.
2. In each iteration, we update the level with the minimum of $\alpha_i + x_i$, *i.e.* update ν with $\frac{1}{\min\ \alpha_i + x_i}$.
3. According to $x_i = \max\ \{0, \frac{1}{\nu} - alpha_i\}$, we update x by $x =$ *np.maximum(0, 1 / nu - alpha)*.
4. Calculate the gradient of our target function $f(x) = -\sum_{i=1}^{n} \log\left(\alpha_i + x_i\right)$, we get
$$\frac{\partial f}{\partial x_i} = -\frac{1}{\alpha_i + x_i}$$
$\therefore$
$$\nabla f(x) = -\frac{1}{\alpha + x}$$
5. And we obtain the direction(the negative gradient), then we try to determine the step size. And I offerd two ways of determine the step:
   - Set a learning rate and fix the step with learning rate (however I find this method always leads to bad output up to the water sum is out of total water, I will discuss this later)
   - To ensure $\sum_{i=1}^{n} x_i = 1$, I set a big step first and a ratio beta, and gradually make new x(*i.e.* $x + step * direction$) get close to 1.
6. Next, we update x with $x - step * grad$.

3.3 Newton Method

The python code of the key function *fill water* is as follows:

```
def fill_water(alpha, total_water, precision, max_iter):
    n = len(alpha)
    x = np.ones((n, 1))*(total_water / n)

    for t in track(range(max_iter), description = " ..."):
        nu = 1 / np.min(alpha + x)
        x = np.maximum(0, 1 / nu - alpha)
        grad = -1 / (alpha + x)
        nt = alpha + x
```

```
        step = backtracking_line_search(x, nt, total_water)
        x = x + step * nt

    return np.array(x)
```

1. Like the strategy in *Gradient Descent*, initialize x averagely.
2. In each iteration, we update the level with the minimum of $\alpha_i + x_i$, *i.e.* update ν with $\frac{1}{\min\ \alpha_i + x_i}$.
3. According to $x_i = \max\ \{0, \frac{1}{\nu} - alpha_i\}$, we update x by $x =$ *np.maximum(0, 1 / nu - alpha)*.
4. We have know that $\nabla f(x) = -\frac{1}{\alpha + x}$, then we calculate $\nabla^2 f(x)$

$$\frac{\partial}{\partial x_j}\left(-\frac{1}{\alpha_i + x_i}\right) = \begin{cases} \frac{1}{(\alpha_i + x_i)^2}, & i = j \\ 0, & i \neq j \end{cases}$$

   Then we have $\nabla^2 f(x) = diag(\frac{1}{(\alpha_1 + x_1)^2}, \frac{1}{(\alpha_2 + x_2)^2}, \ldots, \frac{1}{(\alpha_n + x_n)^2})$
   $\nabla^2 f(x)^{-1} = diag((\alpha_1 + x_1)^2, (\alpha_2 + x_2)^2, \ldots, (\alpha_n + x_n)^2)$
   $\therefore$ the *Newton step*

$$\Delta x_{nt} = -\nabla^2 f(x)^{-1} \nabla f(x) = (\alpha_1 + x_1, \alpha_2 + x_2, \ldots, \alpha_n + x_n)^T$$

5. And we obtain the direction(Δx_{nt}), then we try to determine the step size:
   - Set a big step first and a ratio beta, and gradually make $x + step * direction$ get close to 1.
6. Next, update x with $x + step * \Delta x_{nt}$

4 Results and Analysis

4.1 Horizontal Comparision

In this comparision, I fixed the random seed(generate α) with 123456, and analize three methods above.

4.1.1 Fix Dimension $n = 10$

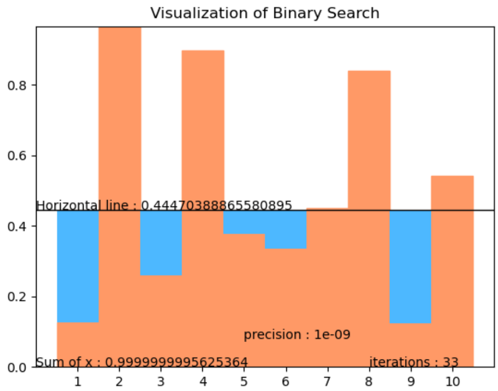

Figure 2: *Binary Search*

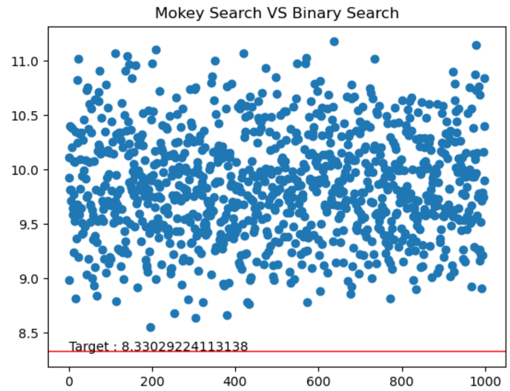

Figure 3: *Binary Search*

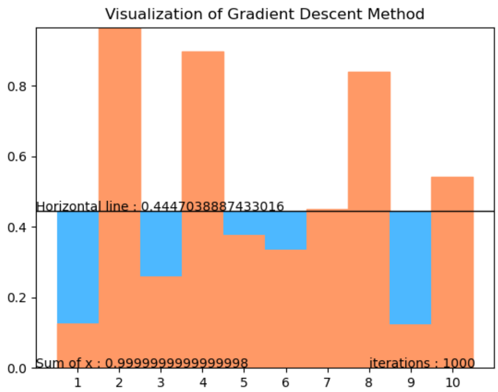

Figure 4: *Gradient Descent Method*

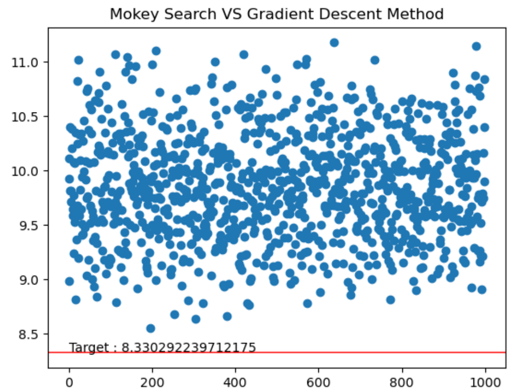

Figure 5: *Gradient Descent Method*

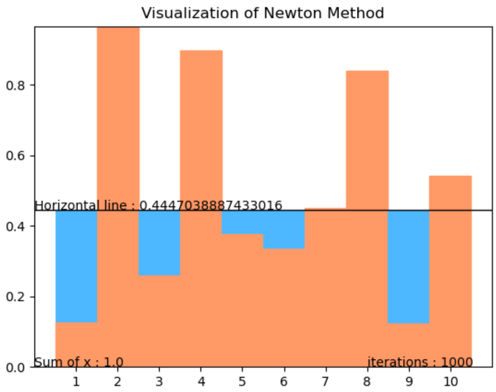

Figure 6: *Newton Method*

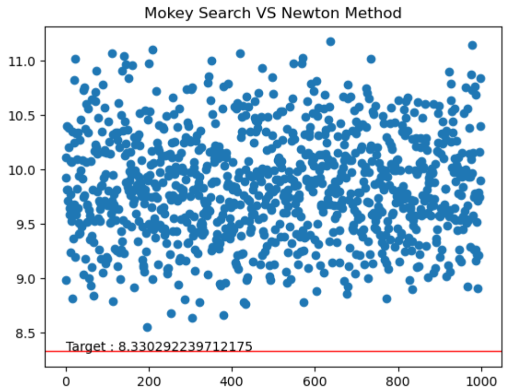

Figure 7: *Newton Method*

- In *Binary Search*, I set precision with $1e-9$, and we can see that it only needs 33 iterations, which is very efficient.
  It turned out that sum of x is 0.9999999995625364 which is very close to 1 and satisfies the precision and target decreases to a small number 8.33029224113138.

- In *Gradient Descent Method*, I set iterations with 1000 and used the *Backtracking Line Search* with 1000 max iteration.
  It turned out that the sum of x is 0.9999999999999998 which is very close to 1 and target decreases to 8.330292239712175 which is a little better than *Binary Search*.

- In *Newton Method*, I set iterations with 1000 and used the *Backtracking Line Search* with 1000 max iteration.
  It turned out that the sum of x is 1.0 which is very close to 1 and target decreases to 8.330292239712175.

4.1.2 Discussion on Dimension n

Table 1: Binary Search

dimension n	**iterations**	**sum of** x	**target**
10	30	1.000000001753952	8.330292234022055
20	30	1.0000000009110206	20.652378045251726
50	32	1.0000000009443624	59.84541793154762
100	33	0.9999999989148151	127.35888574561403
200	33	0.9999999994600418	249.72533875828353
500	35	0.9999999997656048	651.7772009927137
1000	35	0.9999999996562974	1319.1718511257977
10000	42	0.9999999999154736	14023.175128139863

In *Binary Search*, I set precision with $1e-8$.

Table 2: Gradient Descent Method

dimension n	**sum of** x	**target**
10	0.9999999999999998	8.330292239712175
20	**0.9999999999999997**	20.652378049182065
50	**1.0**	59.84541793759993
100	**1.0**	127.35888573539113
200	**1.0**	249.7253387515363
500	**0.9999999999999998**	651.7772009878523
1000	**1.0**	1319.171851116217
10000	**0.9999999999999996**	14023.175128131234

In *Gradient Descent Method*, I set max iterations with 1000.

Table 3: Newton Method

dimension n	**sum of** x	**target**
10	**1.0**	8.330292239712175
20	0.9999999999999991	20.65237804918207
50	0.9999999999999986	59.84541793759995
100	0.9999999999999957	127.35888573539121
200	0.9999941080478836	249.72553013269402
500	0.9297244906311094	658.2840629609998
1000	0.5782796887567309	1353.8930686187957
10000	0.615221710576339	14221.123470723647

In *Newton Method*, I set max iterations with 1000 as the same.

- From table 1, we find out that in *Binary Search* with parameter *precision* fixed, as the dimension grows, the iterations grows slowly, but is always efficient.

- From table 2 and the results in monkey search, we were surprised to find that the performence of *Gradient Descent Method* is almost unaffected by the dimension.

- From table 3, we can see that as the dimension grows, the sum of x slowly deviates from 1, which is a bad phenomenon.
  I guess this is because the number of iterations is too small, and when I set the dimension $n = 10000$ and max iterations with 100000, the sum of x turned out to become 0.9999999926158202 with target 14023.175130704825, which looks normal.
  The reason I guess is that **As the *Newton Method* proceeds, the step size should decrease, otherwise we may choosed a big step and made the sum of x decreases a lot**.
  In order to confirm my guess, I set the max iteration with 1000, dimension $n = 10000$ and discuss on the initial step in *Backtracking Line Search*.

Table 4: Discussion on Initial Step Size in *Backtracking Line Search*

initial step size	sum of x	target
1.0	0.615221710576339	14221.123470723647
0.8	**0.9842580219785946**	**14219.855889521325**
0.5	0.615221710576339	14221.123470723647
0.1	**0.9842580219785946**	**14219.855889521325**
0.01	0.7874384545583886	14220.531921277568

From the table, interestingly, different initial step size may obtian the same result, and it not that the smaller initial step sieze the better performence.

4.2 Vertical Comparision

In this comparision, I fixed $n = 10$, then set a series of parameters of of each method and compare their impact. In this part, the random seed is still 123456.

4.2.1 Binary Search

Table 5: Binary Search

precision	iterations	sum of x	target
1e-1	7	0.9854959700589823	8.37749991528356
1e-2	10	0.9964362155179058	8.341863037752896
1e-3	13	0.9996475623041371	8.331435698156694
1e-4	17	1.0000209098077328	**8.330224405058946**
1e-5	20	1.000002957539133	8.330282644958652
1e-6	23	0.9999998159219318	8.330292836892756
1e-7	27	1.0000000122727468	8.330292199897297
1e-8	30	1.000000001753952	8.330292234022055
1e-9	33	0.9999999995625364	8.33029224113138
1e-10	**37**	**0.9999999999734269**	8.33029223979838

- From the table above, it is easy to find that the number of iterations and the precision of difference between sum of x and 1 is directly related to the parameter *precision*.

- And performence of *target* may be rarely associated with parameter *precision*.

4.2.2 Gradient Descent Method

In analysis below, I made two comparisions: the max iterations and using backtracking line search or not. I do not consider the iterations in backtracking line search, because it will always ends in a small number of iterations.

Table 6: Gradient Descent Method using Backtracking Line Search

iterations	**sum of x**	**target**
1	0.5996978112837809	10.046418174429736
5	0.9866644124857917	8.430947441250991
10	0.9997781218498675	8.334272506259687
20	0.9999845516531557	8.33034869077645
50	0.9999999999994104	8.330292239714463
100	**0.9999999999999998**	**8.330292239712175**
500	**0.9999999999999998**	**8.330292239712175**
1000	**0.9999999999999998**	**8.330292239712175**
10000	**0.9999999999999998**	**8.330292239712175**

From the table above, it is easy to find out that the convergence speed is impressive, and saturation is reached after almost 100 iterations.

Table 7: Gradient Descent Method with Fixed Stepsize

stepsize	**sum of x**	**target**
1e-1	136.68263799843982	-38.237979542758495
1e-2	39.8889399957294	-21.63855225777158
1e-3	9.401411836686217	-5.1838843477010155
1e-4	**1.324753676483748**	**7.338759167059794**
1e-5	0.2820381305500916	11.226990829633314
1e-6	0.205025479153595	11.67804173573701
1e-7	0.19702955018123156	11.729147183266553

In the experiments above, I set max iterations with 1000, and the performence looks bad.

4.2.3 Newton Method

In analysis below, I made comparisions among different max iterations and do not consider the iterations in backtracking line search.

Table 8: Newton Method using Backtracking Line Search

iterations	sum of x	target
1	0.8359152954500617	10.035688630176235
5	0.8423697957445522	9.286513603368162
10	0.9562616963351661	8.55558788241405
20	0.9984104159780414	8.337795046131635
50	0.9999999098836335	8.330292674682592
100	0.9999999999999952	8.330292239712206
500	**1.0**	**8.330292239712175**
1000	**1.0**	**8.330292239712175**
10000	**1.0**	**8.330292239712175**

From the table above, it is easy to see that the convergence speed is impressive, and saturation is reached after almost 100 iterations.

4.3 Visualization of Iteration Process

In this part, I select two random seed 123456 and 12345, and compare the process of these methods.

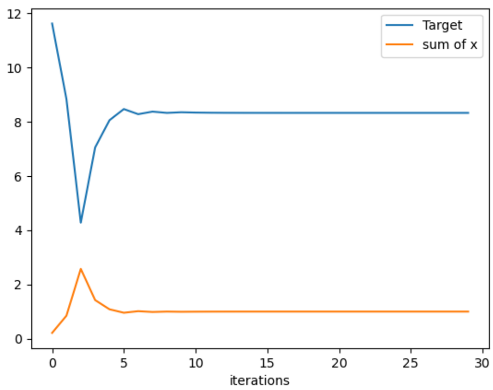

Figure 8: *Binary Search 1*

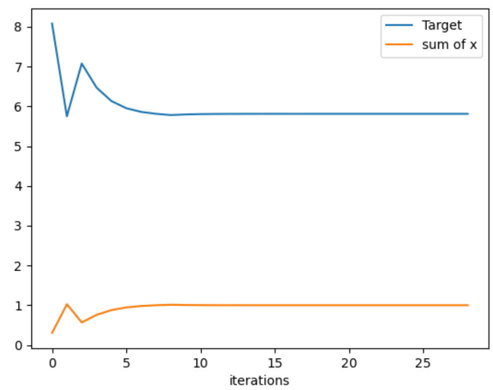

Figure 9: *Binary Search 2*

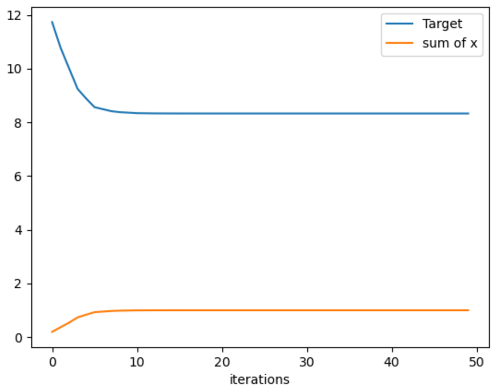

Figure 10: *Gradient Descent Method 1*

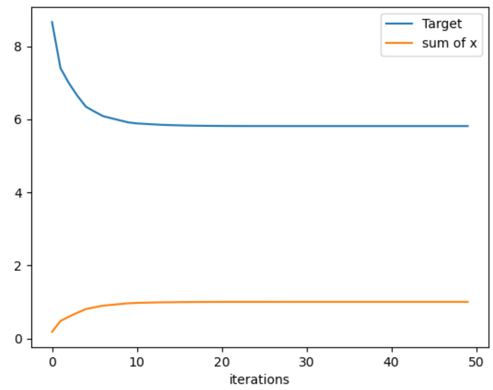

Figure 11: *Gradient Descent Method 2*

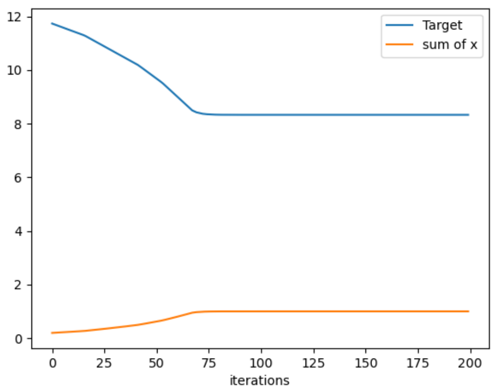

Figure 12: *Newton Method 1*

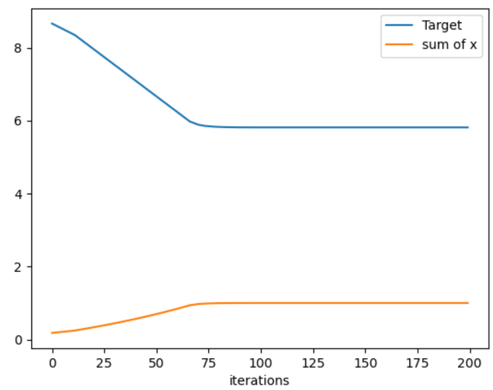

Figure 13: *Newton Method 2*

In the graph above, 1 means we used random seed 123456 and 2 is 12345.

- It is apparent that the *Binary Search* moves fast, but is not stable in the beginning.
- And the *Gradient Descent Method* converge faster than *Newton Method*, the reason I guess is: **Since *Newton Method* uses the *Taylor* second-order term,which is quadratic, the rate of descent is significant while x is far from x_{opt}, and it slows down while x is getting close to x_{opt}**. And in this problem, x is always *near* x_{opt}.

5 Conclusion

In this project, we made a deep analysis of the original problem first and use strategies learned from class converting it into a more straightward problem – the water filling problem. Next, we used three different methods to solve it, and made some clear and intuitive comparisions and analysis.

From this project, I gained a deeper understanding of handling optimization problems, it helped me learn to see optimization problems from a different perspective, I hold a firm belief that this will be of great benefit to my future study and life.