Linear and Convex Optimization: Project

Yu Junjie

December 27, 2024

Contents

1 Obejective of the Experiment

1.1 Representation of the Problem

The goal of the project is to develop a program to solve the *Water-filling* problem. And the *Water-filling* problem can be represent as a convex optimization problem as below:

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

$$\text{subject to} \qquad x \succeq 0, \mathbf{1}^T x = 1 \tag{2}$$

where $\alpha_i > 0$. And use the KKT condition, we obtain the following relations:

$$x^* \succeq 0, \quad \mathbf{1}^T x^* = 1 \tag{3}$$

$$x_i^*(\nu^* - 1/(\alpha_i + x_i^*)) = 0, \qquad \text{where } i = 1, \ldots, n \tag{4}$$

$$-1/(\alpha_i + x_i^*) - \lambda_i^* + \nu^* = 0, \qquad \text{where } i = 1, \ldots, n \tag{5}$$

After solving the equations that the KKT condition gives, we can finally change the *Water-filling* problem to solve the equation that:

$$x_i^* = \max(0, 1/\nu^* - \alpha_i) \tag{6}$$

where ν^* is determined by solving:

$$\sum_{i=1}^{n} x_i^* = 1 \tag{7}$$

The value of ν^* is found by adjusting it until the total power used is 1.

1.2 Obejective of the Experiment

Our goal is to solve the *Water-filling* problem by coding. And finally find the optimal x^* and the minimum $-\sum_{i=1}^{n} \log(\alpha_i + x_i)$ under the randomly generated α.

2 Experimental Process

2.1 Process of the Experiment

For the reason that we have already knew the *Water-filling* problem can

transform into the problem of adjusting the value of ν^* to satisfy the constraints: $\sum_{i=1}^{n} x_i^* = 1$, where $x_i^* = \max(0, 1/\nu^* - \alpha_i)$. Therefore, we only have to do the iterations to find the optimal ν^*, and correspondingly obtain x_i^* and the optimal solution. So, we write the code to do the iterations and calculate the target.

2.2 Inplement Details of My Code

As for the code, we define the following functions:

- *calculate_target*: we use this function to calculate $-\sum_{i=1}^{n} \log(\alpha_i + x_i)$ after we generate the random α and get the optimal x_i^*.
- $fill\_water$: I inplement the convex optimization algorithm in this function to do the iterations to find the optimal ν^*, x^* and the optimal solution. This function will be introduced specificly in the next subsection.
- *visualize_water*: For the reason that the optimization problem has the reality backgrounds as water-filling. So, we intuitively show the final result by plot in the form of filling water.
- *visualize_targets_and_error*: this function help us intuitively see the changing of the targets we get and the corresponding error in the period of iterations.
- *monkey_search* and $visualize_monkey_search$: we use these two functions to see that our algorithm are better than the solutions generated randomly.

And for some parameters, we choose:
$0.0 \leq \alpha \leq 1.0, total\_water = 1.0, dimension = 10, precision = 10^{-6}$.

As for the most inportant part: the $fill\_water$ function, I have tried three algorithms introduced in our lecture, and a more Bisection Method which will be explained in the following five part.

Note that: The detailed comments of my code, please turn to corresponding jupyter notebook. In the following part, I will only choose the important $fill\_water$ function to show and the explaination is copied from part of the content in corresponding jupyter notebooks. I use GPT to translate Chinese into English, so, if you want to see it more clearly and efficiently, I suggegest to turn to the relevant notebook.

2.2.1 The Simple Method

Initially, I want to try the most simple way to test whether or not I can get an acceptable solution, even if not, I can have a relatively better start point of ν^*. And the code is showed below:
note that: the comments for the code I provide is removed to get a good look. To see the complete version, turn to **project_simple.ipynb**.

```
1  def fill_water(alpha, total_water, precision, track=False):
2      nu = 2.0
3      error = 1.0
4      iteration = 0
5      targets = []
6      errors = []
7
8      while error > precision:
9          x_star = np.maximum(0, 1/nu - alpha)
10         total_power = x_star.sum()
11         error = np.abs(total_power - total_water)
12
13         if track:
14             targets.append(calculate_target(alpha, x_star))
15             errors.append(error)
16
17         if total_power > total_water:
18             nu *= 1.01
19         else:
20             nu *= 0.99
21
22         iteration += 1
23
24     if track:
25         return x_star, targets, errors
26     return x_star
```

In the simple way for the *fill_water* function, I initialize $\nu = 2.0$ and $error = 1.0$. The reason why I choose $\nu = 2.0$ rather than other value is that after I carry out several experiment I find the final optimal ν^* is always close to 2.0. And then we keep to update the value of ν thus updating the value of $error$ until $error > precision$(the precision we want to achieve). And the rule of updating is the simple $\nu* = 1.01$ or $\nu* = 0.99$ based on the value relations between $total\_power$ and $total\_water$. And put the updating process to the reality, it is just the process of changing the water level. And

due to the fact that the function: $\sum_{i=1}^{n} \max(0, 1/\nu^* - \alpha_i)$ is a piecewise-linear function of $1/\nu^*$, therefore, we will certainly get to achieve the precision we set after several times of iterations.

However, this simple method seemed not to work so well which will be introduced in the third part: Results and Analysis.

Having done the work of the simple way for the *fill_water* function and get the solution that is not quite well. I started to think of the algorithms that I learned in the lectures: the General Descent, the Gradient Descent, Line Search, Newton's Method. And after some discussion with roommates, the idea of Bisection Method.

All of these methods except for Line Search (because it's quite difficult for me to achieve) will be explained in the following subsections.

2.2.2 the General Descent

```
1  def fill_water(alpha, total_water, precision, track=False):
2      step_size = 0.01
3      nu = 2.0
4      error = 1.0
5      iteration = 0
6      targets = []
7      errors = []
8
9      while np.abs(error) > precision:
10         x = np.maximum(0, 1/nu - alpha)
11         total_power = x.sum()
12         error = total_power - total_water
13
14         if track:
15             targets.append(calculate_target(alpha, x))
16             errors.append(error)
17
18         nu += step_size * error
19
20         iteration += 1
21
22     if track:
23         return x, targets, errors
24     return x
```

Define the *fill_water* function.

First, initialize the value of ν to 2.0 (this ν value is based on empirical

results after several experiments), and set the initial value of error to 1.0. Start the iteration from 0, and create arrays targets[] and errors[] to record the solution and corresponding errors after each iteration.

Initialize the $step\_size = 0.01$, which corresponds to the step size t in the algorithm.

Iterate until the absolute value of the error is less than the desired precision. The error represents the difference between total_water (which is set to 1) and total_power (which is the sum of all x_i values in the current state under the current ν value). Unlike the simple method, where the absolute value is taken, the error here also indicates the direction of descent Δx.

The iteration method is based on General Descent, where Δx determines the direction of descent and $step\_size * \Delta x$ is the length. Besides this, I also heve done some optimization to this method. In the case of traditional descent methods, if we choose $|\Delta x| = 0.1$ and use its sign for the direction, oscillations near the target value might occur. But I observe that as the error decreases, the step size should also reduce in order to approach the optimal value. However, since this is General Descent and not Gradient Descent, we choose to update ν using the formula:

$$\nu \mathrel{+}= \text{step\_size} \times \text{error}$$

This approach is more efficient for this problem, as it converges faster and the step size is appropriately adjusted. This will be further explained in the Gradient Descent part.

2.2.3 the Gradient Descent

```
1  def fill_water(alpha, total_water, precision, track=False):
2      step_size = 0.1
3      x = np.zeros_like(alpha)
4      nu = 2.0
5      error = 1.0
6      iteration = 0
7      targets = []
8      errors = []
9
10     while error > precision:
11         x = np.maximum(0, 1/nu - alpha)
12         total_power = x.sum()
13         error = np.abs(total_power - total_water)
14
15         if track:
```

```
            targets.append(calculate_target(alpha, x))
            errors.append(error)

        gradient = np.sum(1/(nu**2))
        nu -= step_size * gradient

        iteration += 1

    if track:
        return x, targets, errors
    return x
```

Note that: In fact, the Gradient Descent algorithm we use is exactly General Descent with $\Delta x = -\nabla f(x)$.

Therefore, similar to the General Descent algorithm, we iterate until the value of the error is smaller than the desired precision. Since $\Delta x = -\nabla f(x)$, this Δx already has directional meaning, and the changing of its value also continues to optimize the *step_size* during each iteration. Thus, the error here refers to the absolute difference between total_water and total_power. And this error is only used as the condition to determine when the iteration should stop.

As for the iteration method: we choose step_size = 0.1 (corresponding to t in the algorithm), and $\Delta x = -\nabla f(x)$.

In code, this corresponds to:

$$\text{gradient} = \sum \left(\frac{1}{\nu^2} \right)$$

$$\nu - = \text{step\_size} \times \text{gradient}$$

When we designed the fill_water function using the General Descent method before, we applied an optimization with step_size $\times$ error. The experimental results showed that using Gradient Descent was really slow, because α is randomly generated, and we couldn't find a good starting point. Thus, we had to choose a more empirical value of 2.0 for ν. Meanwhile, I increased the step_size from 0.01 to 0.1 and reduced the precision to 0.01. Even under these conditions, the running time was mostly in the minute range. On the other hand, the optimized General Descent can provide results in under 1 second, with very good performance. Therefore, I concluded that the optimized General Descent algorithm is better than Gradient Descent.

2.2.4 Bisection Method

```python
def fill_water(alpha, total_water, precision, track=False):
    lower = 0
    upper = np.max(1 / alpha)
    x = np.zeros_like(alpha)
    error = 1.0
    iteration = 0
    targets = []
    errors = []

    while error > precision:
        nu = (lower + upper) / 2
        x = np.maximum(0, 1/nu - alpha)
        total_power = x.sum()
        error = np.abs(total_power - total_water)

        if track:
            targets.append(calculate_target(alpha, x))
            errors.append(error)

        if total_power < total_water:
            upper = nu
        else:
            lower = nu

        iteration += 1

    if track:
        return x, targets, errors
    return x
```

After a discussion with my roommate, I suddenly realized that since $\sum_{i=1}^{n} \max\left(0, \frac{1}{\nu^*} - \alpha_i\right)$ is a piecewise-linear increasing function of $1/\nu^*$, a more efficient method for solving such problems is to perform continuous bisection to quickly approximate the final optimal x^*.

In the fill_water function, we first determine the initial values for the lower and upper bounds. Since $x = \max(0, \frac{1}{\nu} - \alpha)$, it follows that $\nu < \frac{1}{\alpha_i}$ for all i. Therefore, we choose the initial upper bound as $\max\left(\frac{1}{\alpha_i}\right)$, while the lower bound is set to 0.

We then begin the iteration. The iteration steps are the same as before, except for the method of updating the parameters. Unlike the descent method that updates ν, here we use bisection to update either the lower or

upper bound. However, essentially, this still updates ν, as in each iteration, ν is the average of the two bounds.

2.2.5 Newton's Method

```
1  def fill_water(alpha, total_water, precision, track=False):
2      nu = 2.0
3      error = 1.0
4      iteration = 0
5      targets = []
6      errors = []
7      epsilon = 1e-8
8
9      while error > precision:
10         x = np.maximum(0, 1/nu - alpha)
11         total_power = x.sum()
12         error = np.abs(total_power - total_water)
13
14         if track:
15             targets.append(calculate_target(alpha, x))
16             errors.append(error)
17
18         dF_dnu = np.sum(1 / (nu**2))  # First derivative
19         d2F_dnu2 = np.sum(-2  / (nu**3))  # Second
               derivative
20
21         if abs(d2F_dnu2) < epsilon:
22             d2F_dnu2 = epsilon
23
24         nu = nu - dF_dnu / d2F_dnu2
25
26         iteration += 1
27
28     if track:
29         return x, targets, errors
30     return x
```

Following Newton's method, we choose $\Delta x_{nt} = -\nabla^2 f(x)^{-1} \nabla f(x)$. Therefore, we first compute the first and second derivatives and use them as indicators for the update.

And the reason for the following code:

```
    if abs(d2F_dnu2) < epsilon:
    d2F_dnu2 = epsilon
```

is that when $d2F\_dnu2$ becomes very small, a runtime error warning occurs, it's just a way to avoid the denominator to be zero. Hence, we set a minimum value $\varepsilon = 10^{-8}$.

However, unfortunately, since the code already runs quite slowly during the Gradient Descent step (when calculating the first derivative), calculating the second derivative here increases the running time significantly. As a result, the outputs of each code block in the jupyter notebook are not provided and I will not do the analysis of result produced by this method.

3 Results and Analysis

3.1 The Simple Method

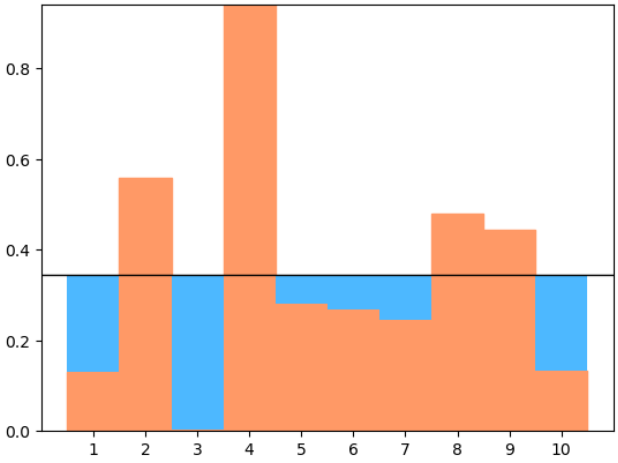


Figure 1: visualization of water filling

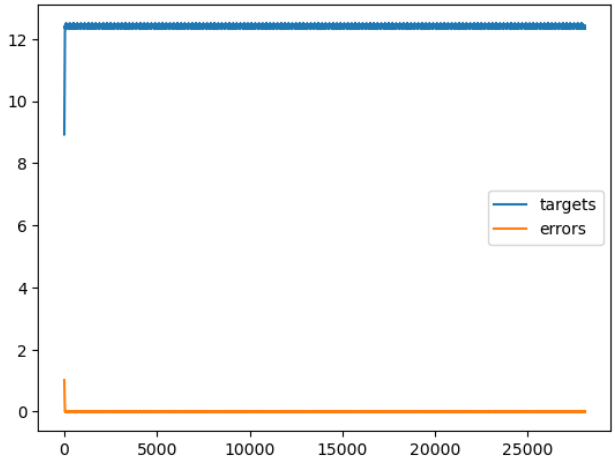


Figure 2: targets and errors during iterations

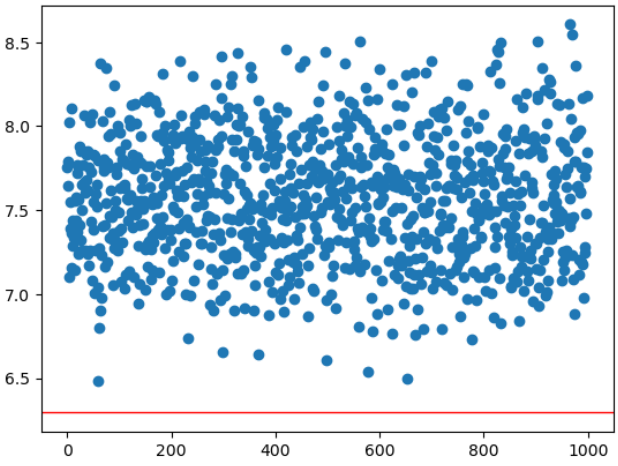


Figure 3: compared with “monkey”

As you can see, in fig[2], the line seems to be quite wide, its because the x-label is to large and the target keeps vibrate even it comes to 25000 times iterations.

To see it more clearly, I make it stop when iterations comes to 100, and the figure is as below:

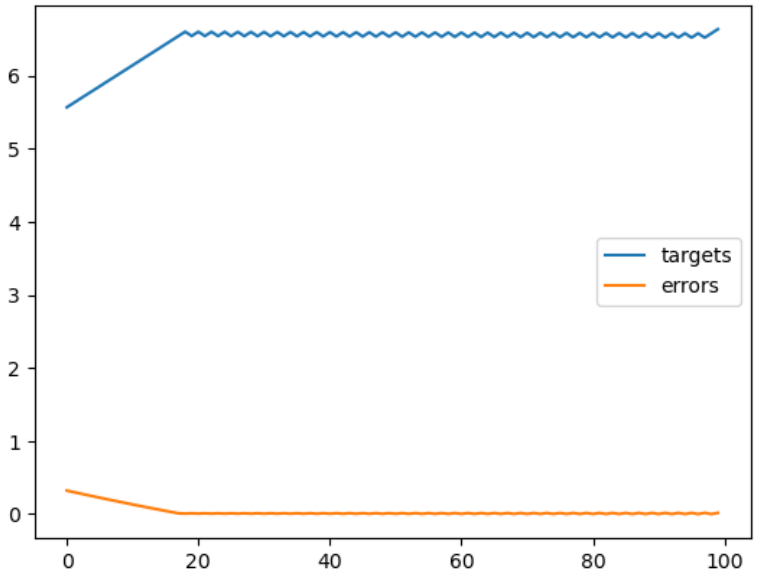


Figure 4: targets and errors under 100 times iterations

This time, it's quite clear to see the vibration, and correspondingly $\sum_{i=1}^{n} x_i = 0.9834414024368723$, which can't achieve our precision.

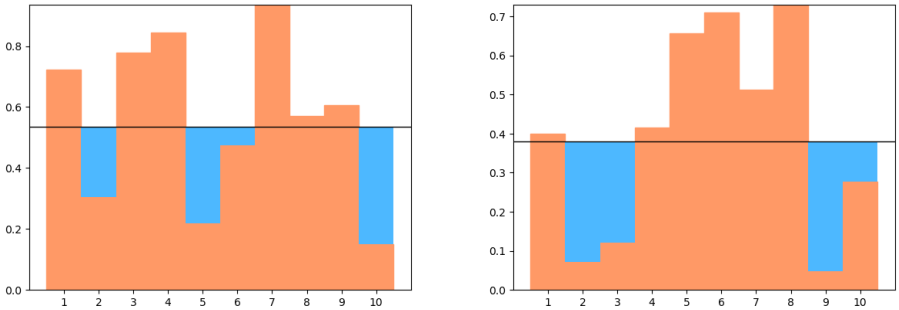
Therefore, the simple way can't achieve our standard, so let's turn to the descent method!

3.2 the General Descent

Here I provide the result of two independent experiments, and you can see, by using the General Descent, we can obtain an ideal result.

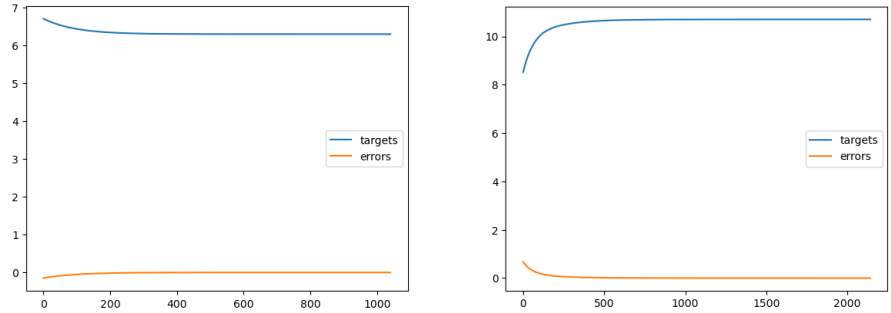
And the detailed data for the experiment is:

- Experiment 1:
  - $\sum_{i=1}^{n} x_i = 0.9999990075794267$
  - $horizontal_line = 0.5364982106984345$
  - $iterations = 1041$
- Experiment 2:
  - $\sum_{i=1}^{n} x_i = 1.0000009971989834$
  - $horizontal_line = 0.3793815661631934$
  - $iterations = 2143$



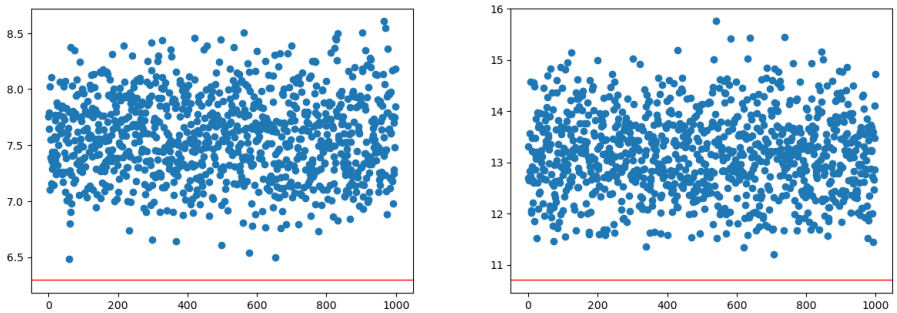
(a) experiment 1 (b) experiment 2

Figure 5: visualization of water filling



(a) experiment 1 (b) experiment 2

Figure 6: targets and errors during iterations



(a) experiment 1 (b) experiment 2

Figure 7: compared with “monkey”

As you can see, in Figure 6, the graphic is very smooth under the elaborate adjust by error, and the vibration won’t occur because the step_size we

choose can achieve approach in one direction. That is to say, using the way as $\nu+ = step\_size * error$ will not lead our solution after each iteration to the other side of final optimal solution.

There's no doubt that it do does a good job!

3.3 the Gradient Descent

At first, as I said in the Second Part, the Gradient Descent Method runs very slow, thus I change the precision to 0.01.

And here is the data for it:

- $\sum_{i=1}^{n} x_i = 0.9995361774638688$, which only achieves precision of 0.001.
- $horizontal_line = 0.5948296449808184$
- $iterations = 12$

And the visualization graph is provided below:

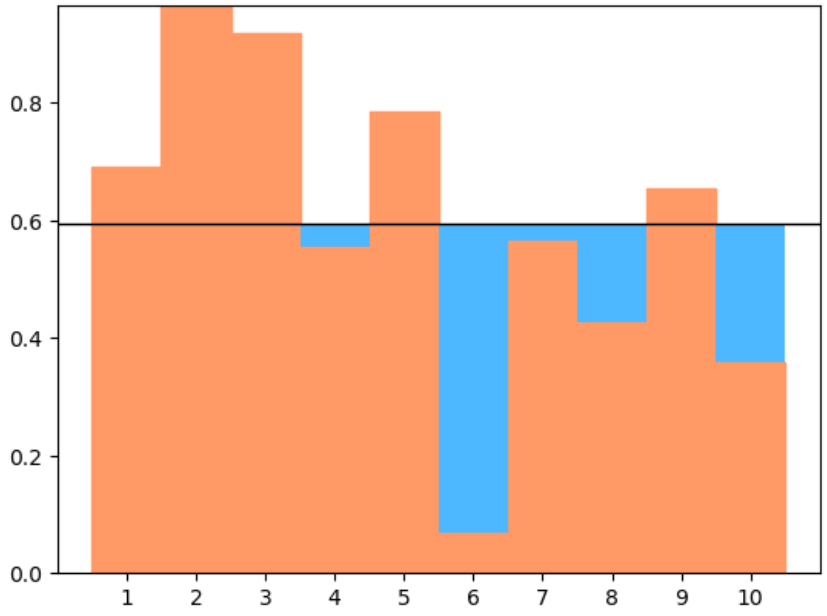


Figure 8: visualization of water filling

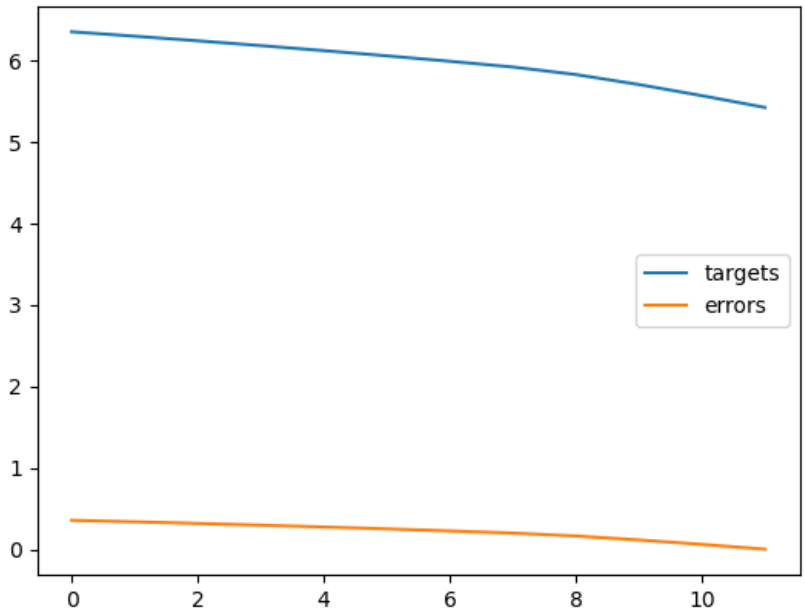


Figure 9: targets and errors during iterations

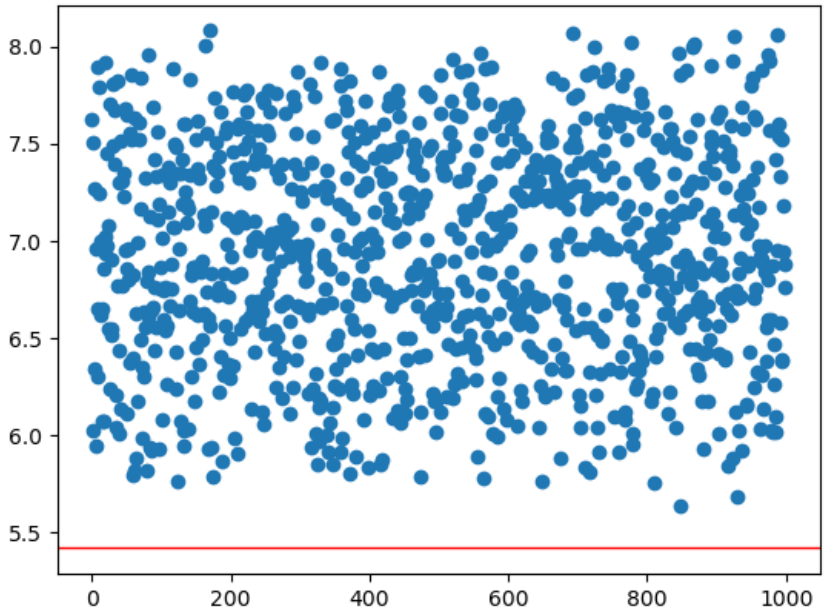


Figure 10: compared with “monkey”

As we can see, after 12 iterations, the algorithm has already had the ability to provide a result with very small bias. If we have enough time and

sufficient computility, we can believe it will return a fantastic result even though the cost weighs a lot. So, it's not that perfect even though we might obtain a good result.

3.4 Bisection Method

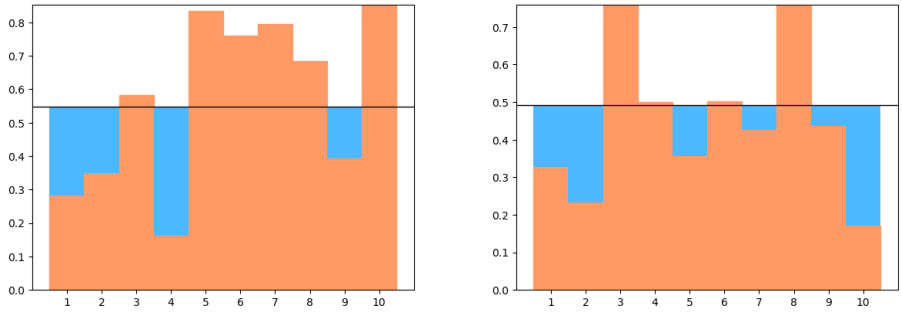
To conclude the above three algorithms, the result that the simple method returns are not that good, the Gradient Descent costs too much time and computility to achieve our goal (so do the Newton's Method whose results I don't show because I only get the result when the precision is 0.1, and $n = 3$. And result with higher precision and more channels can't be obtained even though I have waited for nearly 10min).

So, till now, under some comparison. We find it seems that only the optimized General Descent can efficiently return a good result. Here, fortunately, the Bisection Method also makes it! And it is even better for fewer iterations.

And here is the result data, I also provide two seperate experiments:

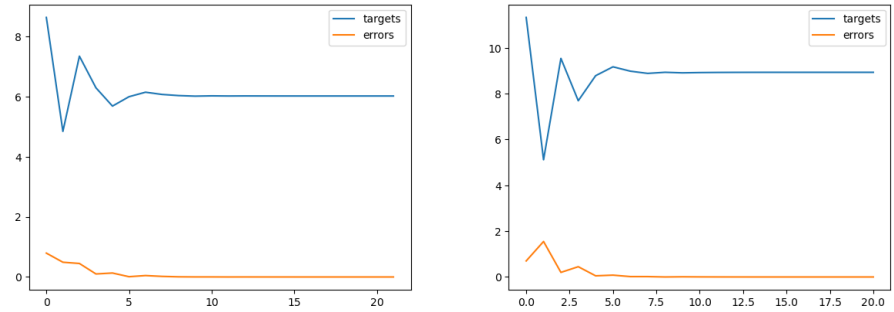
- Experiment 1:
  - $\sum_{i=1}^{n} x_i = 0.9999996766480913$
  - $horizontal_line = 0.5474824842815678$
  - $iterations = 22$
- Experiment 2:
  - $\sum_{i=1}^{n} x_i = 1.0000005952842315$
  - $horizontal_line = 0.49176061581498987$
  - $iterations = 21$

And here are the corresponding visualized graph:



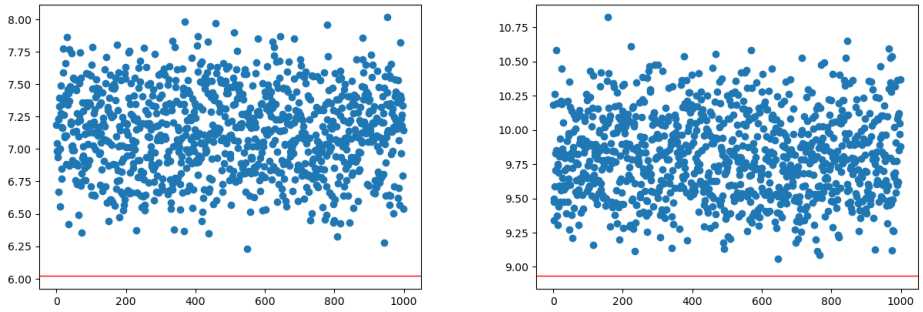
(a) experiment 1 (b) experiment 2

Figure 11: visualization of water filling



(a) experiment 1 (b) experiment 2

Figure 12: targets and errors during iterations



(a) experiment 1 (b) experiment 2

Figure 13: compared with “monkey”

It should be explained that we can see the graphic vibrate severely at start, that is because when changing the *upper* and *lower* at begining, its

value changes greatly because of the large value difference between *upper* and *lower* at start.

We can see from both the data and the graph, the Bisection Method works perfect! And due to its much smaller iterations than the optimized General Descent, we can say it's the best among all the method I provide.

Done! Thanks for reading!