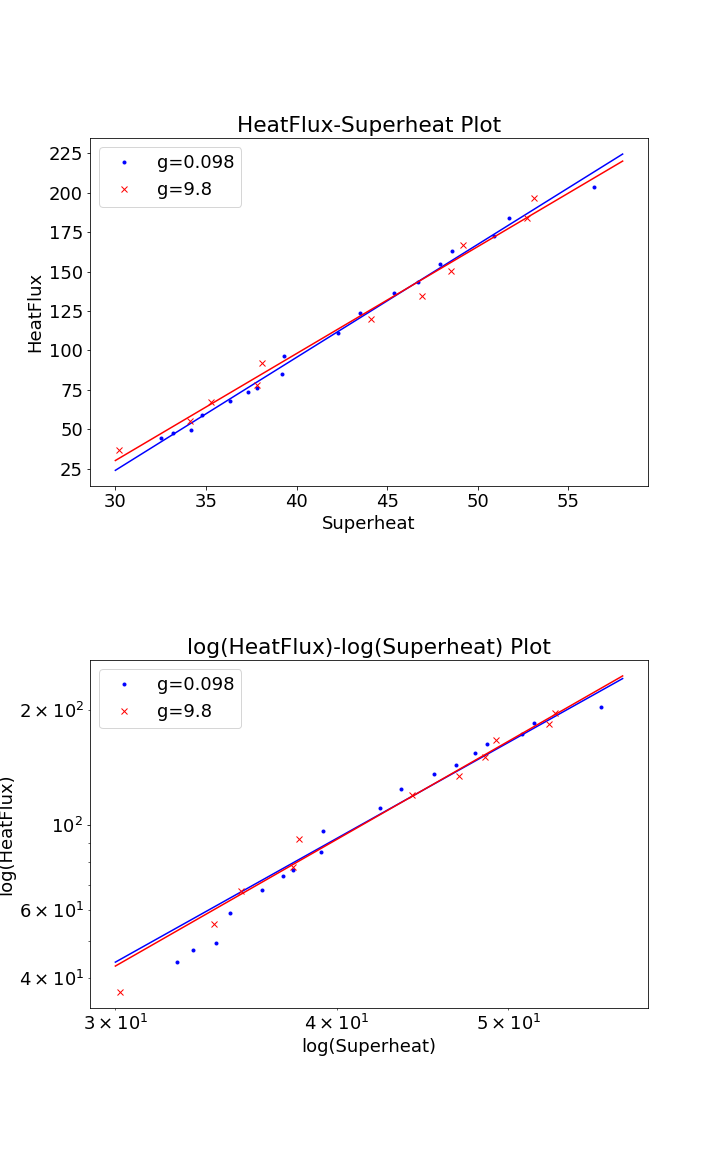
**Report for Project 1**

**By Jiahao Huang, Sept. 14th**

**Task 1**



The figure above includes a HeatFlux-Superheat plot and a HeatFlux-Superheat log-log plot. Blue dots and lines represent the data and regression under the condition of micro gravity (0.098), while the red ones represent the condition of regular gravity (9.8).

As shown in the figure, heat flux is positively correlated with superheat, and the relationship varied a little in different gravity conditions.

It seems that there could be a linear relationship between heat flux and superheat, so linear regression is applied to the data. (As for the log-log plot, linear relationship should be transferred into exponential relationship according to simple math calculations.) The result is shown in the table below. Mean Square Error (MSE) is used to decide whether the regression is good or not.

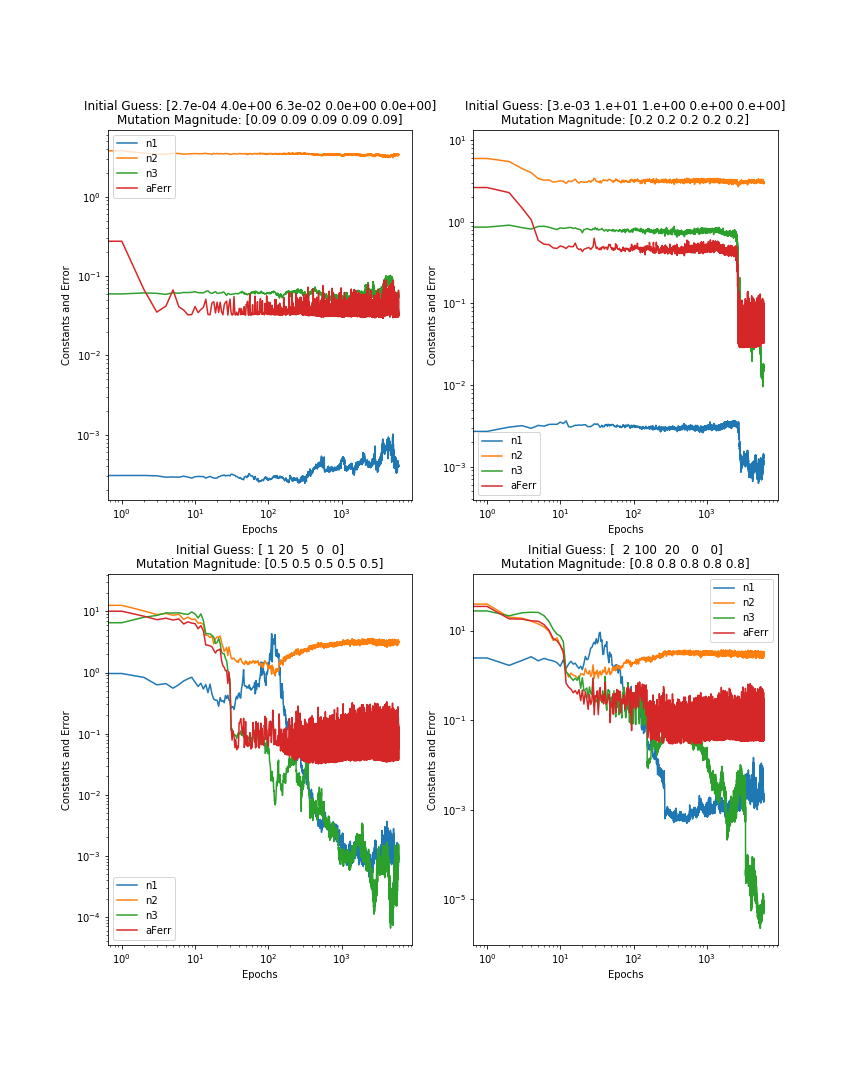
|  |  |  |
| --- | --- | --- |
| g | Lin. Regression | Exp. Regression |
| 0.098 | y = 7.167503 \* x + -191.125966  MSE = 15.158677 | y = 0.006760 \* x ^(2.581364)  MSE = 72.352377 |
| 9.8 | y = 6.785368 \* x + -173.429351  MSE = 38.131234 | y = 0.005384 \* x ^(2.641377)  MSE = 32.514733 |

Therefore, linear regression has a better performance when gravity is low, while exponential regression works better under a regular-gravity condition. However, the conclusion isn’t accurate enough because of inadequacy of data.

Code Modified:

* Save the data as a csv file, so that the reading process will look concise.
* The data type is changed to pandas.DataFrame, also to make the code concise.
* Linear and exponential regression added.
* Plot the scatter data and the regression graph and save it.

**Task 2**



The figure above indicates how constants and errors change (within totally 6000 generations/epochs) under different initial guesses and mutation magnitude. Mutation magnitude are adjusted to make the genetic algorithm come to ideal results. With a more dramatic initial guess (farther from the optimal parameters), the mutation magnitude has to be adjusted larger for a broader range of trial. Parameters and some of the results are listed below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Experiment | Parameters | | | |
| n1 | n2 | n3 | Mutation  Magnitude |
| 1 (Original) | 2.7e-4 | 4 | 6.3e-2 | 9% |
| 1 (Optimal) | 7.640e-4 | 3.194 | 6.064e-2 |
| 2 (Original) | 3.0e-3 | 10 | 1 | 20% |
| 2 (Optimal) | 1.290e-3 | 3.055 | 5.000e-2 |
| 3 (Original) | 1 | 20 | 5 | 50% |
| 3 (Optimal) | 3.682e-3 | 2.755 | 1.098e-2 |
| 4 (Original) | 2 | 100 | 20 | 80% |
| 4 (Optimal) | 9.000e-4 | 3.150 | 4.700e-2 |

|  |  |  |
| --- | --- | --- |
| Experiment | Results | |
| Relative Absolute Error | Best Epoch |
| 1 | 2.958% | 3679 |
| 2 | 2.872% | 3233 |
| 3 | 3.186% | 485 |
| 4 | 2.935% | 682 |

Some more details can be found out from the tables. Though different solutions are reached when initial guesses vary, all of the solutions have the same order of magnitudes, which means these solutions are closed to the exact optimal solution. All of the relative absolute errors are near 3%, and it also proves genetic algorithm works well in this case.

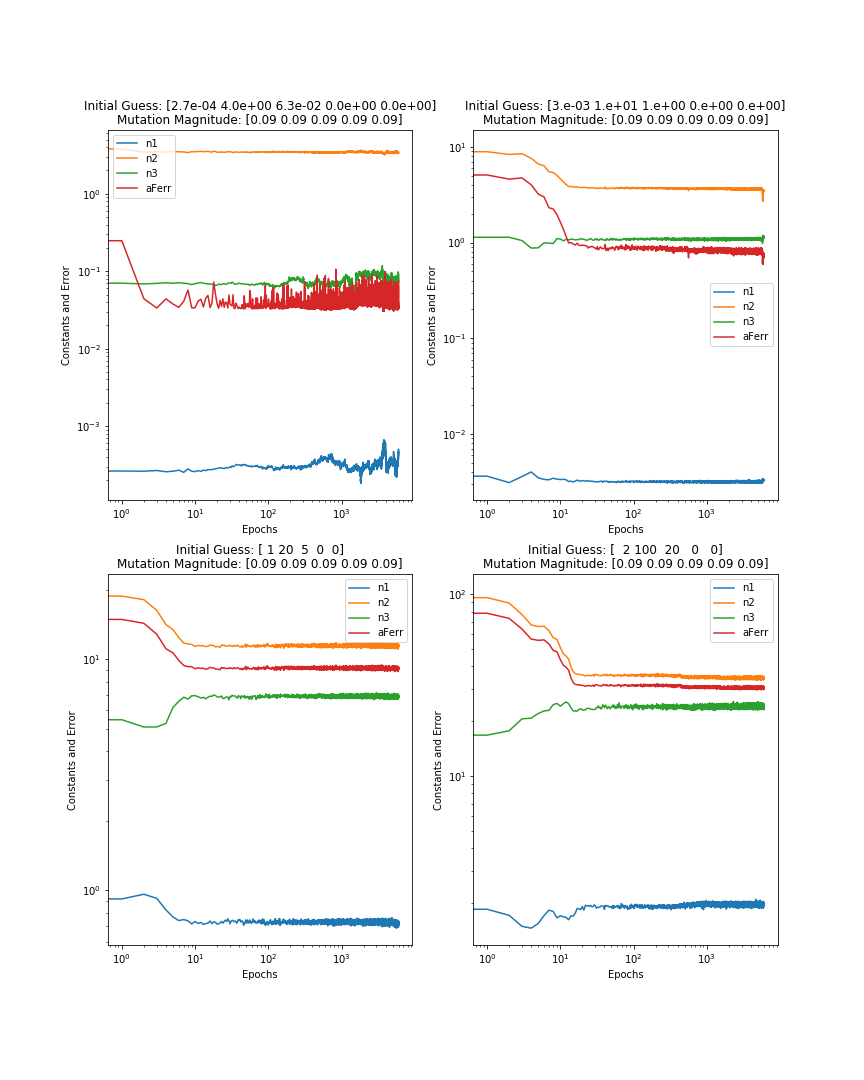
As this problem can also be considered as a linear regression problem, therefore we could create a baseline with linear regression algorithm (See Task2\_LR.py). Listed below is the result of linear regression.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameters | | | Results |
| n1 | n2 | n3 | Relative Absolute Error |
| 3.933e-3 | 2.746 | 4.943e-2 | 2.995% |

It is amazing that the solutions from Experiment 1, 2, 4 perform better than linear regression when the loss function is Relative Absolute Error (no doubt linear regression has the minimum mean square loss). All of the results above prove genetic algorithm to be potential when the model is non-linear or the loss function is partly non-differentiable.

The result of task 2 also aligns with that of task 1. One of the conclusions from task 1 is that the relationship between heat flow and superheat varied a little under different conditions of gravity. According to task 2, there are two orders of magnitude difference between n2 and n3, correspondingly represents the weight of superheat and gravity. It is a quantitative proof for the conclusion of task 1.

As mentioned before, the mutation magnitudes and initial parameters were carefully chosen. If we choose 0.09 as the mutation magnitude for all 4 experiments, some of them will fail to reach an optimal solution. Here are the results.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Experiment | Parameters | | | |
| n1 | n2 | n3 | Mutation  Magnitude |
| 1 (Original) | 2.7e-4 | 4 | 6.3e-2 | 9% |
| 1 (Optimal) | 6.300e-4 | 3.244 | 6.247e-2 |
| 2 (Original) | 3.0e-3 | 10 | 1 | 9% |
| 2 (Optimal) | 3.236e-3 | 2.803 | 1.102 |
| 3 (Original) | 1 | 20 | 5 | 9% |
| 3 (Optimal) | 0.7247 | 11.043 | 6.874 |
| 4 (Original) | 2 | 100 | 20 | 9% |
| 4 (Optimal) | 1.9748 | 33.5636 | 23.49 |

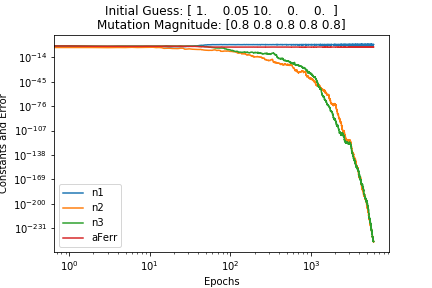
|  |  |  |
| --- | --- | --- |
| Experiment | Results | |
| Relative Absolute Error | Best Epoch |
| 1 | 0.02994 | 3161 |
| 2 | 0.5907 | 5825 |
| 3 | 8.830 | 5902 |
| 4 | 29.625 | 4881 |

As the result show, when initial guesses are far from the optimal solution, a small mutation magnitude could make the algorithm reach a bad solution. Experiment 4 has also been done for another 50,000 generations, but the relative absolute error is still around 30, which proves the algorithm converges to a partial minimum. Therefore, according to the experiments above, when initial guess is far away from the optimal solution, increasing the mutation magnitude could help achieve a good result in some conditions.

However, adjusting the mutation magnitude doesn’t always promise good solutions. Furthermore, if initial values are extremely badly chosen, the algorithm will hardly approach the optimal solution, no matter how mutation magnitude is chosen. Here’s an example.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Experiment | Parameters | | | |
| n1 | n2 | n3 | Mutation  Magnitude |
| 5 (Original) | 1 | 0.05 | 10 | 80% |
| 5 (Optimal) | 96.70 | 1.7e-5 | 2.072e-2 |
| LR sol. (Comparison) | 3.933e-3 | 2.746 | 4.943e-2 | / |

|  |  |  |
| --- | --- | --- |
| Experiment | Results | |
| Relative Absolute Error | Best Epoch |
| 5 | 9.576% | 61 |



As shown in the graph, n2 and n3 rapidly slopes to zero. However, it’s not the single case. Similar trends could also be observed in experiment 2, 3, 4 (the n3/green line).

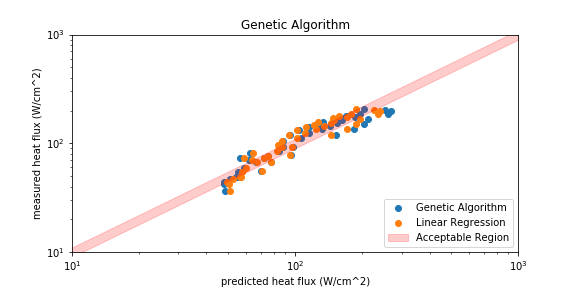
Here is my explanation to this phenomenon: genes naturally have a higher probability to become less. Each gene has a uniform chance to mutate by [-m, m], where m is the magnitude of mutation. However, according to the property of uniform distribution, after several generations, the new genes are more possibly to be less than the original ones (even though the expectation doesn’t change). This phenomenon is especially observable when m is large.

It can explain why parameter n3 also slopes to zero in experiment 3 and 4, since the original parameter is chosen large enough, genetic algorithm passes by (but not converges into) the optimal solution. Consequently, many of generations are wasted, as the best generations for these two experiments are 485 and 682 (far less than 6000). As for experiment 5, the original n2 plays a trivial role in the equation, and therefore it will follow its natural trend to decline into 0.

So, here’s some conclusions from these experiments.

1. If the right values are unknown, large original parameters and mutation magnitude can be chosen.
2. Use small mutation magnitude to fine-tune the solution.
3. Uniform distribution may not be good enough for mutation rate. Normal distribution can be tried.

Also we can use the optimal parameters generated from genetic algorithm and linear regression to see if the predicted heat flux aligns with true heat flux.



The blue prints are results from genetic algorithm, while the orange points are those from linear regression, and the red region is the error region of measurement. The statistical results are listed below.

|  |  |  |
| --- | --- | --- |
| Methodology | RMS Error | RMS Deviation |
| Genetic Algorithm | 0.02679 | 0.1752 |
| Linear Regression | 0.02433 | 0.1627 |

We can decide whether the model is overfitting or underfitting through RMS deviation. With a large deviation, the model is underfitting, usually the RMS error will also be large. However, a too little deviation means the model is possibly facing the problem of overfitting, which means the model has learned the noise as feature. Either overfitting models or underfitting model will perform poorly in the prediction tasks. Therefore, a good deviation should be around the measurement error.

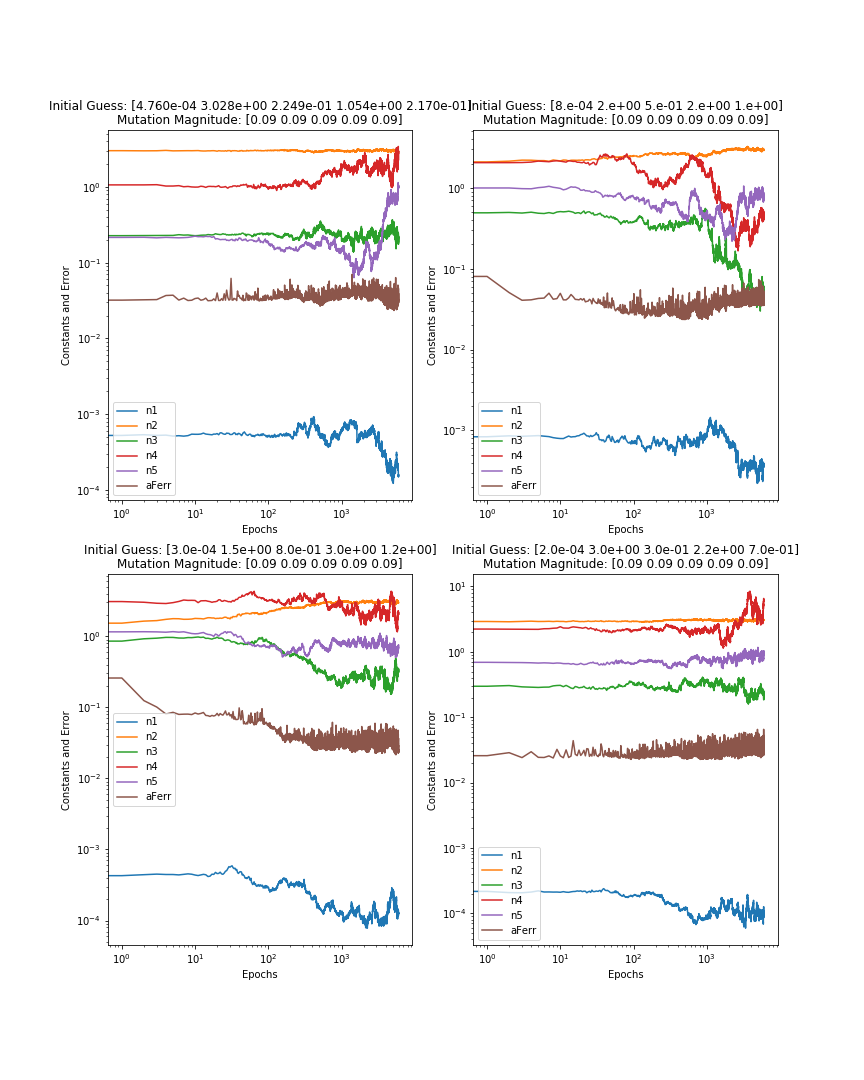
In this case, since the measurement error is 10%, the deviation is somehow larger than the error. Therefore, overfitting is not happening, and the model isn’t misled by the noise. With an RMS error is less than 0.03, we can claim it could be a good model for prediction.

Code Modified:

* Reconstruct the code to align with my coding habits. Please refer to appendix for details.
* Use functions and packages (numpy, pandas) to make the codes concise.
* Include a linear regression algorithm as a baseline.
* Make the graphs easier to read.

**Task 3**

Now, a 5-parameter model is applied to this problem. Since the model can no longer be transformed into a linear regression problem, only genetic algorithm is feasible. With some simple modification to the codes, we can obtain a constants-and-error plot for this new model.



Experiments are conducted 4 times. However, according to the conclusion from Task 2, all of the initial parameters are chosen closed to the optimal solution to promise and magnitude of mutation is chosen as 0.09, so as to promise the convergence and accuracy of the algorithm.

Some of the detailed parameters and results are listed below.

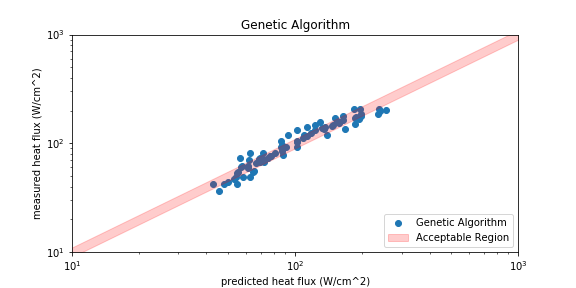
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Experiment | Parameters | | | | | |
| n1 | n2 | n3 | n4 | n5 | Mutation  Magnitude |
| 1 (Original) | 4.76e-4 | 3.028 | 0.2249 | 1.054 | 0.217 | 9% |
| 1 (Optimal) | 1.546e-4 | 2.976 | 0.3454 | 1.798 | 0.6690 |
| 2 (Original) | 8e-4 | 2 | 0.5 | 2 | 1 | 9% |
| 2 (Optimal) | 6.619e-4 | 2.675 | 0.3712 | 1.716 | 0.4266 |
| 3 (Original) | 3e-4 | 1.5 | 0.8 | 3 | 1.2 | 9% |
| 3 (Optimal) | 1.669e-4 | 2.946 | 0.4143 | 1.572 | 0.5742 |
| 4 (Original) | 2e-4 | 3 | 0.3 | 2.2 | 0.7 | 9% |
| 4 (Optimal) | 1.065e-4 | 3.004 | 0.3637 | 2.018 | 0.7475 |

|  |  |  |
| --- | --- | --- |
| Experiment | Results | |
| Relative Absolute Error | Best Epoch |
| 1 | 2.336% | 5196 |
| 2 | 2.347% | 505 |
| 3 | 2.151% | 5524 |
| 4 | 2.255% | 1496 |

Despite different initial guesses, all of the experiments have similar optimal solution to this problem. All of the relative absolute errors are around 2.2%, which is more likely to be a precise prediction model compared with the one in Task 2. As experiment 3 has a minimum error among all the experiments, solution from experiment 3 is chosen for further use in the following tasks.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | n1 | n2 | n3 | n4 | n5 |
| Chosen Parameters | 1.669e-4 | 2.946 | 0.4143 | 1.572 | 0.5742 |

Deviation can also be calculated in the same way as Task 2. Results are shown below.

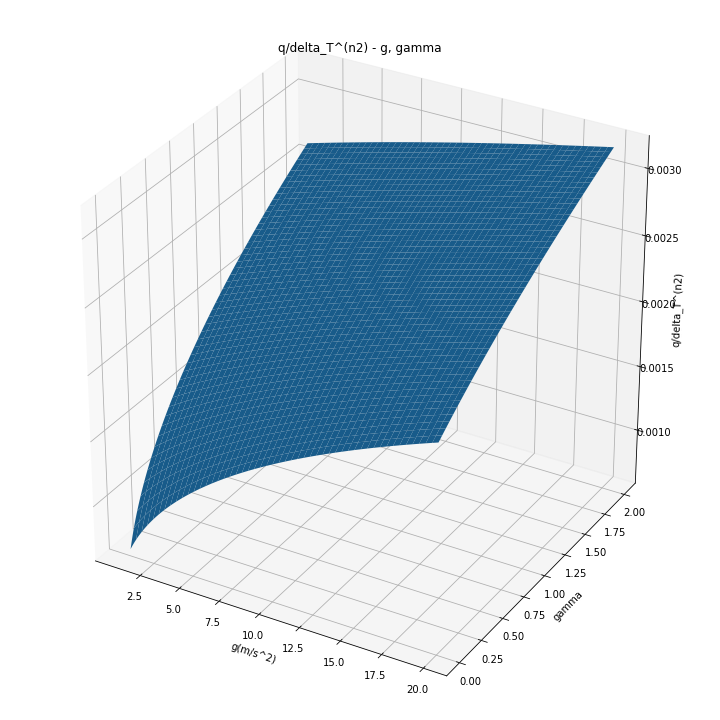


|  |  |  |
| --- | --- | --- |
| Methodology | RMS Error | RMS Deviation |
| Genetic Algorithm | 0.01484 | 0.1289 |

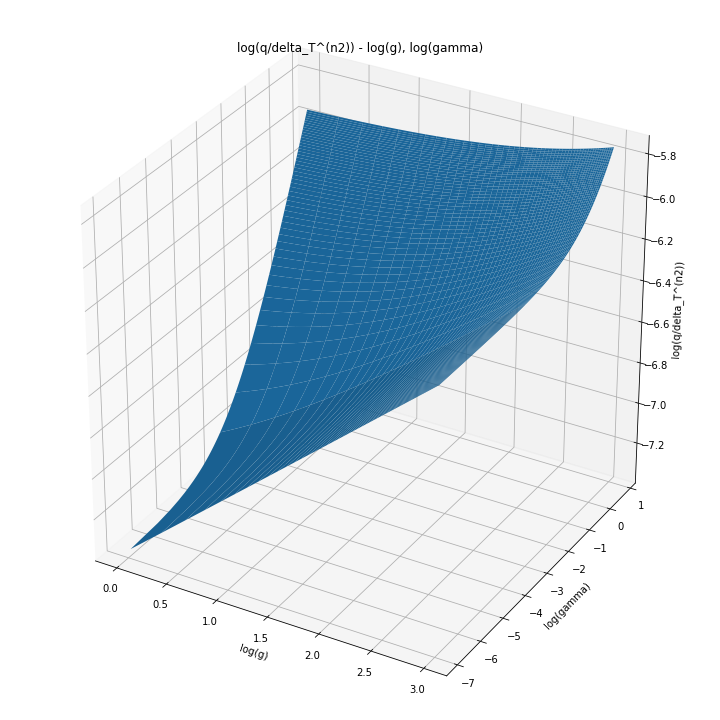
All of the criterions are having a better result compared with task 2, which means the 5-parameter model is more precise than the 3-parameter model. Also, the deviation is still larger than 10%, which means overfitting is not happening.

If we use the optimal parameters obtained from genetic algorithm, we can get the followed relationship:

With the condition of , we can create a surface plot of versus and as followed.



Also, we can manually transform the coordinates into log form.



Code Modified:

* Loss Function is changed into:

# Relative Absolute Error

def AFERR(n):

Ferr = -lydata["HeatFlux"] + np.log(n[:, 0]) + n[:, 1] \* lydata["Superheat"] + n[:, 2] \* np.log(ydata["g"] + n[:, 3]\*gen\*ydata["SurfaceTension"]) + n[:, 4]\*lydata["p"]

aFerr = np.abs(Ferr) / np.abs(lydata["HeatFlux"])

return aFerr

where ydata and lydata are type of DataFrame and n is a 77x5 matrix which stores 77 groups of parameters (n1 to n5). In my program, the first column of -1 is removed because I don’t think it’s necessary.

For more details, please refer to the README.txt file and the codes in the appendix.

**Task 4**

Since we have the model (the positions of and have been exchanged to align with the former problem):

We can take the natural log of both sides and obtain:

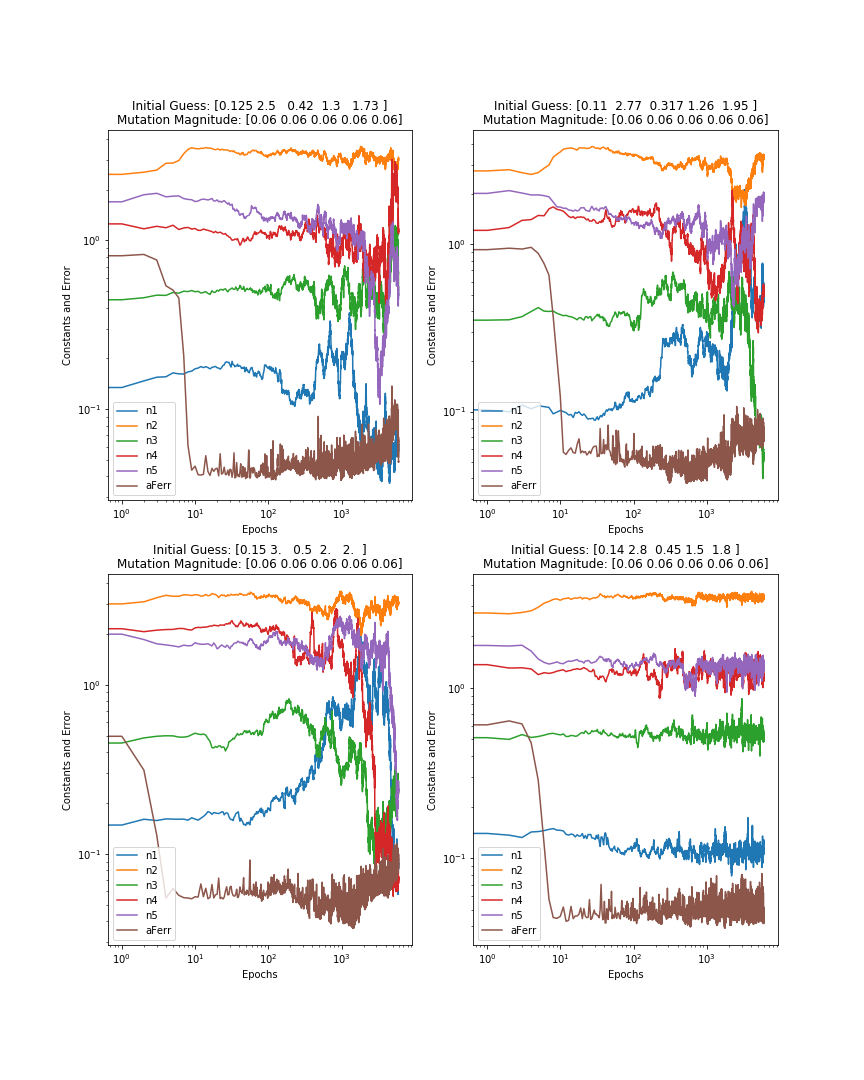
Therefore, should be:

And should be the sum of all relative absolute .

**Task 5**

In non-dimension form, similar genetic algorithm could also be applied to this problem. Since parameters have been transformed into non-dimensional form, the initial guesses should also be altered accordingly. Furthermore, since is the value to be predicted, relative absolute error from previous tasks are no longer of referential importance. Even if the log error is large, the actual error could also be small when is not so large.

Here is the constants and error plots for 4 different initial guesses.



Several failure guesses have been tried before these 4 experiments. These 4 experiments are to fine-tune the parameters for an optimal solution, also MFRAC is chosen as 0.3 to obtain a more precise model. Some of the detailed results are listed below.

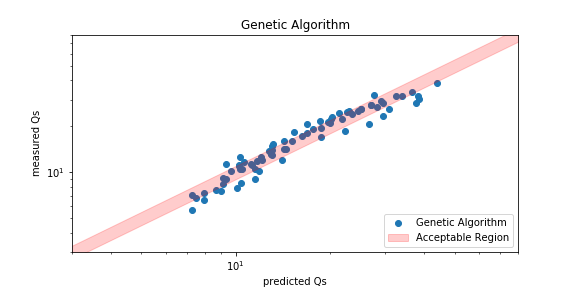
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Experiment | Parameters | | | | | |
| n1 | n2 | n3 | n4 | n5 | Mutation  Magnitude |
| 1 (Original) | 0.125 | 2.5 | 0.42 | 1.3 | 1.73 | 6% |
| 1 (Optimal) | 0.2028 | 3.0769 | 0.4884 | 0.9616 | 1.2637 |
| 2 (Original) | 0.11 | 2.77 | 0.317 | 1.26 | 1.95 | 6% |
| 2 (Optimal) | 0.2523 | 3.0733 | 0.4301 | 1.0458 | 1.3931 |
| 3 (Original) | 0.15 | 3 | 0.5 | 2 | 2 | 6% |
| 3 (Optimal) | 0.6107 | 2.9642 | 0.4532 | 1.0588 | 1.8455 |
| 4 (Original) | 0.14 | 2.8 | 0.45 | 1.5 | 1.8 | 6% |
| 4 (Optimal) | 0.1517 | 3.3467 | 0.4782 | 0.9635 | 1.4136 |

|  |  |  |
| --- | --- | --- |
| Experiment | Results | |
| Relative Absolute Error | Best Epoch |
| 1 | 0.03792 | 984 |
| 2 | 0.03717 | 533 |
| 3 | 0.03636 | 1429 |
| 4 | 0.03903 | 1774 |

All of the solutions has a relative absolute log error around 0.04, which is slightly large compared with those in the previous tasks. However, it is due to little target (). In fact, the model is performing as well as the previous one, which will be shown later. Solution from experiment 3 seems to minimize the error, and therefore is chosen as the optimal model for followed problems.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | n1 | n2 | n3 | n4 | n5 |
| Chosen Parameters | 0.6107 | 2.9642 | 0.4532 | 1.0588 | 1.8455 |

We can also obtain statistic features in a similar way as we did before. The measurement error rate is still chosen as 10%. Actually, since the non-dimensional parameters are processed by multiplication and division, the error rate could be chosen properly higher. Results are shown below:

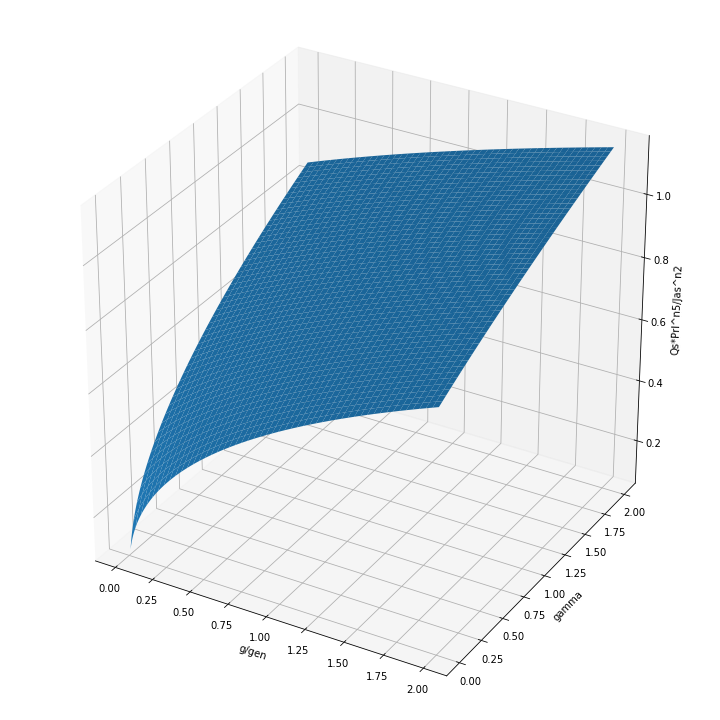


|  |  |  |
| --- | --- | --- |
| Methodology | RMS Error | RMS Deviation |
| Genetic Algorithm | 0.014746 | 0.12779 |

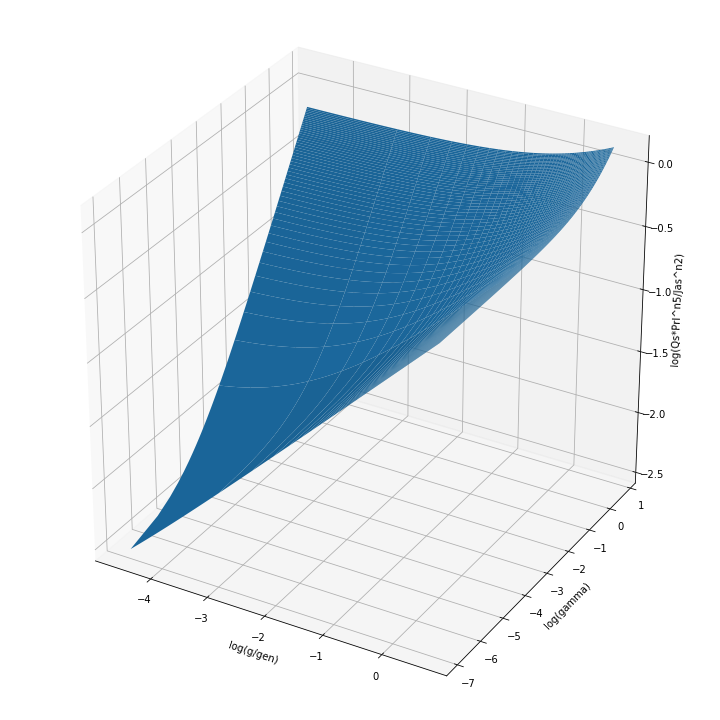
RMS error and RMS deviation are slightly higher than the result of task 3. But the RMS error is still low and overfitting does not happen in this case. Therefore, the non-dimensional, 5-parameter model is also performing well in this problem.

From the equation, we can obtain that:

The surface plot follows like:

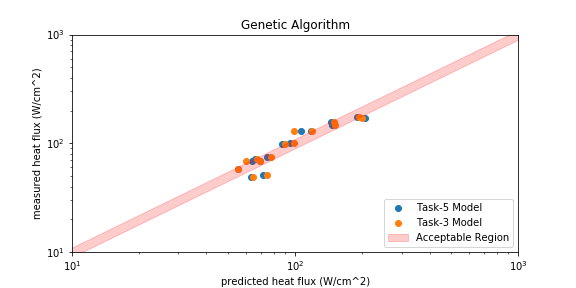


Also, we can manually transform the coordinates into log form.



These 2 graphs look quite similar to those of Task 3, which proves the correctness. In fact, they are almost the same equation except for the scale.

We can validate the accuracy of models with the validation dataset. Here, task-3 (orange points) model is a raw data model while task-5 model (blue points) is a dimensionless one. Here are the statistical results for these 2 models.



|  |  |  |
| --- | --- | --- |
| Model | RMS Error | RMS Deviation |
| Dimensionless | 0.04098 | 0.1577 |
| Raw data | 0.04445 | 0.1699 |

Both of the models are performing well on the validation dataset, and dimensionless model has a better performance considering RMS error and RMS deviation.

To sum up, both of the methodologies can be applied to this data prediction problem and have good performances on training data and validation data. The advantage of raw-data model is that researchers can create a model quite easily, without considering any physical principals. The dimensionless model requires some calculation, but it is more interpretable as well as precise and aligns with the scientific habit. Also, more details could be included in a dimensionless model. Physical parameters regarding p are used in a dimensionless model instead of a direct p in the raw data model.

After all, both of the models work, and any model can be chosen to meet the needs of researchers.

**Appendix**

**Note:** The codes pasted below in the appendix might be not runnable if your local file structure is wrong. If you need a runnable version, please contact me through [jiahao\_huang@berkeley.edu](mailto:jiahao_huang@berkeley.edu). I can submit a github link or a zip package of this whole project.

**Appendix 1 - File Structure Tree**

│ curve\_fit.ipynb

│ data1.ipynb

│ data2.ipynb

│ data3.ipynb

│ demo.ipynb

│ README.txt

│ Report.docx

│ Task1.ipynb

│ Task2.ipynb

│ Task2\_LR.ipynb

│ Task3.ipynb

│ Task4.ipynb

│ Task5.ipynb

│

├─.ipynb\_checkpoints

│ …

│

├─data

│ data1.csv

│ data2.csv

│ data3.csv

│ data4.csv

│

└─result

…

Some of the important files will be documented in the following appendixes.

**Appendix 2 – README.txt**

This file is going to give an introduction and explanation to each of the files and codes.

- curve\_fit.ipynb

A baseline program to validate the effect of the solutions obtained from genetic algorithms.

scipy.optimize.curve\_fit is used to obtain a baseline solution within very short codes.

The result just plays a role of reference and is not documented in the report.

- data1.ipynb, data2.ipynb, data3.ipynb

Data processing program for train data (3-parameter), train data (5-parameter) and valid data which saves the data to data/data1.csv, data/data2.csv, data/data4.csv.

As a result, data can be loaded from local files in the following tasks and codes can be shortened.

Data has a type of pandas.DataFrame, and every column are named by its physical name to make the codes more readable.

- demo.ipynb

Original codes given. Not used in the tasks.

- Task1.ipynb

Gives a scatter plot and regression result according to the data.

- Task2.ipynb, Task3.ipynb, Task5.ipynb

These 3 programs share a similar structure, and therefore they are put together. Most of the matrixes have the type of ndarray or DataFrame, so that matrix calculation can be applied and "for" loops are avoided.

Several functions are created for programming convenience.

initialize(): to initialize the matrix n. In my program, n is a 45(77)x5 matrix, and the first column of -1 is removed.

AFERR(): a function to calculate the error. It will change with the data.

selection(): work together with AFERR() to find out nkeep.

mating(): the mating process of genetic algorithm.

train(): the main process of training. In each generation, selection() and mating() will be called.

const\_error(), pred\_true(): 2 plotting programs.

stat(): to get the statistic results of the parameters n\_best.

- Task2\_LR.ipynb

Use linear regression algorithm to solve the Task 2.

- Task4.ipynb

A data process program to obtain the dimensionless parameters from the data. Saved in data/data3.csv.

- data/\*

Processed data is saved in this directory.

- result/\*

The plotting results obtained from the tasks. Also, best parameters are stored in this document.

**Appendix 3 – data3.csv (Non-dimensional data)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Qs | Jas | Prl | g/gen | SurfaceTension |
| 6.829017 | 5.61828 | 4.83 | 0.01 | 1.79 |
| 7.340032 | 5.739289 | 4.83 | 0.01 | 1.79 |
| 7.649738 | 5.912159 | 4.83 | 0.01 | 1.79 |
| 9.167297 | 6.015881 | 4.83 | 0.01 | 1.79 |
| 10.49903 | 6.275186 | 4.83 | 0.01 | 1.79 |
| 11.39718 | 6.448056 | 4.83 | 0.01 | 1.79 |
| 11.81528 | 6.534491 | 4.83 | 0.01 | 1.79 |
| 13.20896 | 6.77651 | 4.83 | 0.01 | 1.79 |
| 14.94331 | 6.793797 | 4.83 | 0.01 | 1.79 |
| 17.18868 | 7.312407 | 4.83 | 0.01 | 1.79 |
| 19.20177 | 7.519851 | 4.83 | 0.01 | 1.79 |
| 21.09098 | 7.848304 | 4.83 | 0.01 | 1.79 |
| 22.2214 | 8.073036 | 4.83 | 0.01 | 1.79 |
| 23.94027 | 8.28048 | 4.83 | 0.01 | 1.79 |
| 25.25652 | 8.401489 | 4.83 | 0.01 | 1.79 |
| 26.7586 | 8.79909 | 4.83 | 0.01 | 1.79 |
| 28.52392 | 8.937386 | 4.83 | 0.01 | 1.79 |
| 31.54355 | 9.749876 | 4.83 | 0.01 | 1.79 |
| 5.683105 | 5.220678 | 4.83 | 1 | 1.79 |
| 8.5324 | 5.894872 | 4.83 | 1 | 1.79 |
| 10.45258 | 6.102316 | 4.83 | 1 | 1.79 |
| 12.07853 | 6.534491 | 4.83 | 1 | 1.79 |
| 14.24648 | 6.586352 | 4.83 | 1 | 1.79 |
| 18.58236 | 7.623573 | 4.83 | 1 | 1.79 |
| 20.79676 | 8.10761 | 4.83 | 1 | 1.79 |
| 23.2744 | 8.384202 | 4.83 | 1 | 1.79 |
| 25.86045 | 8.505211 | 4.83 | 1 | 1.79 |
| 28.49295 | 9.110256 | 4.83 | 1 | 1.79 |
| 30.42861 | 9.179404 | 4.83 | 1 | 1.79 |
| 7.914053 | 4.888889 | 3.91 | 2 | 1.79 |
| 9.089962 | 5.115873 | 3.91 | 2 | 1.79 |
| 10.17254 | 5.168254 | 3.91 | 2 | 1.79 |
| 11.5911 | 4.97619 | 3.91 | 2 | 1.79 |
| 13.21498 | 5.325397 | 3.91 | 2 | 1.79 |
| 13.75627 | 5.290476 | 3.91 | 2 | 1.79 |
| 15.26815 | 5.342857 | 3.91 | 2 | 1.79 |
| 17.15334 | 6.02381 | 3.91 | 2 | 1.79 |
| 19.39316 | 6.02381 | 3.91 | 2 | 1.79 |
| 22.23028 | 6.180952 | 3.91 | 2 | 1.79 |
| 24.9554 | 6.425397 | 3.91 | 2 | 1.79 |
| 26.11264 | 6.652381 | 3.91 | 2 | 1.79 |
| 27.68052 | 6.826984 | 3.91 | 2 | 1.79 |
| 29.30439 | 6.984127 | 3.91 | 2 | 1.79 |
| 31.56289 | 7.368254 | 3.91 | 2 | 1.79 |
| 33.44807 | 7.542857 | 3.91 | 2 | 1.79 |
| 38.2637 | 8.031746 | 3.91 | 2 | 1.79 |
| 6.565767 | 5.134243 | 4.83 | 2 | 1.79 |
| 7.541341 | 5.358974 | 4.83 | 2 | 1.79 |
| 8.439488 | 5.393548 | 4.83 | 2 | 1.79 |
| 10.96359 | 5.600993 | 4.83 | 2 | 1.79 |
| 11.41267 | 5.428122 | 4.83 | 2 | 1.79 |
| 12.66697 | 5.61828 | 4.83 | 2 | 1.79 |
| 14.23099 | 6.275186 | 4.83 | 2 | 1.79 |
| 16.08923 | 6.275186 | 4.83 | 2 | 1.79 |
| 18.44299 | 6.430769 | 4.83 | 2 | 1.79 |
| 20.70385 | 6.638213 | 4.83 | 2 | 1.79 |
| 21.66393 | 6.862945 | 4.83 | 2 | 1.79 |
| 22.9647 | 7.070389 | 4.83 | 2 | 1.79 |
| 24.31192 | 7.191398 | 4.83 | 2 | 1.79 |
| 26.18564 | 7.588999 | 4.83 | 2 | 1.79 |
| 27.74966 | 7.779156 | 4.83 | 2 | 1.79 |
| 31.74486 | 8.28048 | 4.83 | 2 | 1.79 |
| 13.06946 | 7.209892 | 4.54 | 1 | 0 |
| 12.05106 | 7.03616 | 4.54 | 1 | 0 |
| 11.2024 | 6.862427 | 4.54 | 1 | 0 |
| 10.52346 | 6.688695 | 4.54 | 1 | 0 |
| 7.128798 | 5.906899 | 4.54 | 1 | 0 |
| 10.184 | 6.514963 | 4.54 | 1 | 0 |
| 8.995864 | 6.428096 | 4.54 | 1 | 0 |
| 11.10296 | 6.292473 | 4.83 | 0.01 | 1.71 |
| 12.62052 | 6.6555 | 4.83 | 0.01 | 1.71 |
| 14.04517 | 6.828371 | 4.83 | 0.01 | 1.71 |
| 15.99631 | 7.191398 | 4.83 | 0.01 | 1.71 |
| 18.1178 | 7.450703 | 4.83 | 0.01 | 1.71 |
| 21.46262 | 7.848304 | 4.83 | 0.01 | 1.71 |
| 25.03973 | 8.28048 | 4.83 | 0.01 | 1.71 |
| 32.132 | 8.79909 | 4.83 | 0.01 | 1.71 |

**Appendix 4 – curve\_fit.ipynb**

import pandas as pd

import numpy as np

ydata = pd.read\_csv("./data/data3.csv")

lydata = ydata.copy()

lydata = np.log(lydata + 1e-12)

gen=9.8

y = ydata["Qs"]

x = ydata[["Jas", "g/gen", "SurfaceTension", "Prl"]]

def fun(x, n1, n2, n3, n4, n5):

y = n1 \* np.power(x["Jas"], n2) \* np.power((ydata["g/gen"]+n4\*ydata["SurfaceTension"]), n3) \* np.power(ydata["Prl"], -n5)

return y

# Regression

from scipy import optimize

n = optimize.curve\_fit(fun, x, y)[0]

def pred\_true(n, label):

# Calculating y\_pred

n = n.reshape(1,-1)

y\_pred = np.log(n[:, 0]) + n[:, 1] \* lydata["Jas"] + n[:, 2] \* np.log(ydata["g/gen"] + n[:, 3]\*ydata["SurfaceTension"]) - n[:, 4]\*lydata["Prl"]

y\_pred = np.exp(y\_pred)

y\_true = ydata["Qs"]

# Plotting

plt.scatter(y\_pred, y\_true, label=label)

# Deviation and other statistic results

def stat(n):

n = n.reshape(1,-1)

y\_pred = np.log(n[:, 0]) + n[:, 1] \* lydata["Jas"] + n[:, 2] \* np.log(ydata["g/gen"] + n[:, 3]\*ydata["SurfaceTension"]) - n[:, 4]\*lydata["Prl"]

y\_pred = np.exp(y\_pred)

y\_true = ydata["Qs"]

relErr = (y\_pred-y\_true) / y\_true

mean = np.sqrt((np.mean(relErr \* relErr) / relErr.shape[0]))

dev = np.std(relErr)

unacc\_rate = np.where((relErr > 0.1)| (relErr < -0.1))[0].shape[0] / relErr.shape[0]

print("RMSError: %f, RMSDeviation: %f, Unacceptable Rate: %f"%(mean, dev, unacc\_rate))

return mean, dev, unacc\_rate

print(n)

stat(n)

**Appendix 5 – data1.ipynb**

ydata = [[44.1, 32.5, 0.098, 1.79, 5.5]]

ydata.append([47.4, 33.2, 0.098, 1.79, 5.5])

ydata.append([49.4, 34.2, 0.098, 1.79, 5.5])

ydata.append([59.2, 34.8, 0.098, 1.79, 5.5])

ydata.append([67.8, 36.3, 0.098, 1.79, 5.5])

ydata.append([73.6, 37.3, 0.098, 1.79, 5.5])

ydata.append([76.3, 37.8, 0.098, 1.79, 5.5])

ydata.append([85.3, 39.2, 0.098, 1.79, 5.5])

ydata.append([96.5, 39.3, 0.098, 1.79, 5.5])

ydata.append([111., 42.3, 0.098, 1.79, 5.5])

ydata.append([124., 43.5, 0.098, 1.79, 5.5])

ydata.append([136.2, 45.4, 0.098, 1.79, 5.5])

ydata.append([143.5, 46.7, 0.098, 1.79, 5.5])

ydata.append([154.6, 47.9, 0.098, 1.79, 5.5])

ydata.append([163.1, 48.6, 0.098, 1.79, 5.5])

ydata.append([172.8, 50.9, 0.098, 1.79, 5.5])

ydata.append([184.2, 51.7, 0.098, 1.79, 5.5])

ydata.append([203.7, 56.4, 0.098, 1.79, 5.5])

ydata.append([36.7, 30.2, 9.8, 1.79, 5.5])

ydata.append([55.1, 34.1, 9.8, 1.79, 5.5])

ydata.append([67.5, 35.3, 9.8, 1.79, 5.5])

ydata.append([78.0, 37.8, 9.8, 1.79, 5.5])

ydata.append([92.0, 38.1, 9.8, 1.79, 5.5])

ydata.append([120., 44.1, 9.8, 1.79, 5.5])

ydata.append([134.3, 46.9, 9.8, 1.79, 5.5])

ydata.append([150.3, 48.5, 9.8, 1.79, 5.5])

ydata.append([167., 49.2, 9.8, 1.79, 5.5])

ydata.append([184., 52.7, 9.8, 1.79, 5.5])

ydata.append([196.5, 53.1, 9.8, 1.79, 5.5])

'''

ydata.append([42.4, 28.0, 19.6, 1.79, 9.5])

ydata.append([48.7, 29.3, 19.6, 1.79, 9.5])

ydata.append([54.5, 29.6, 19.6, 1.79, 9.5])

ydata.append([62.1, 28.5, 19.6, 1.79, 9.5])

ydata.append([70.8, 30.5, 19.6, 1.79, 9.5])

ydata.append([73.7, 30.3, 19.6, 1.79, 9.5])

ydata.append([81.8, 30.6, 19.6, 1.79, 9.5])

ydata.append([91.9, 34.5, 19.6, 1.79, 9.5])

ydata.append([103.9, 34.5, 19.6, 1.79, 9.5])

ydata.append([119.1, 35.4, 19.6, 1.79, 9.5])

ydata.append([133.7, 36.8, 19.6, 1.79, 9.5])

ydata.append([139.9, 38.1, 19.6, 1.79, 9.5])

ydata.append([148.3, 39.1, 19.6, 1.79, 9.5])

ydata.append([157.0, 40.0, 19.6, 1.79, 9.5])

ydata.append([169.1, 42.2, 19.6, 1.79, 9.5])

ydata.append([179.2, 43.2, 19.6, 1.79, 9.5])

ydata.append([205.0, 46.0, 19.6, 1.79, 9.5])

'''

ydata.append([42.4, 29.7, 19.6, 1.79, 5.5])

ydata.append([48.7, 31.0, 19.6, 1.79, 5.5])

ydata.append([54.5, 31.2, 19.6, 1.79, 5.5])

ydata.append([70.8, 32.4, 19.6, 1.79, 5.5])

ydata.append([73.7, 31.4, 19.6, 1.79, 5.5])

ydata.append([81.8, 32.5, 19.6, 1.79, 5.5])

ydata.append([91.9, 36.3, 19.6, 1.79, 5.5])

ydata.append([103.9, 36.3, 19.6, 1.79, 5.5])

ydata.append([119.1, 37.2, 19.6, 1.79, 5.5])

ydata.append([133.7, 38.4, 19.6, 1.79, 5.5])

ydata.append([139.9, 39.7, 19.6, 1.79, 5.5])

ydata.append([148.3, 40.9, 19.6, 1.79, 5.5])

ydata.append([157.0, 41.6, 19.6, 1.79, 5.5])

ydata.append([169.1, 43.9, 19.6, 1.79, 5.5])

ydata.append([179.2, 45.0, 19.6, 1.79, 5.5])

ydata.append([205.0, 47.9, 19.6, 1.79, 5.5])

'''

ydata.append([77.0, 41.5, 9.8, 0.00, 7.0])

ydata.append([71.0, 40.5, 9.8, 0.00, 7.0])

ydata.append([66.0, 39.5, 9.8, 0.00, 7.0])

ydata.append([62.0, 38.5, 9.8, 0.00, 7.0])

ydata.append([42.0, 34.0, 9.8, 0.00, 7.0])

ydata.append([60.0, 37.5, 9.8, 0.00, 7.0])

ydata.append([53.0, 37.0, 9.8, 0.00, 7.0])

ydata.append([71.7, 36.4, 0.098, 1.71, 5.5])

ydata.append([81.5, 38.5, 0.098, 1.71, 5.5])

ydata.append([90.7, 39.5, 0.098, 1.71, 5.5])

ydata.append([103.3, 41.6, 0.098, 1.71, 5.5])

ydata.append([117.0, 43.1, 0.098, 1.71, 5.5])

ydata.append([138.6, 45.4, 0.098, 1.71, 5.5])

ydata.append([161.7, 47.9, 0.098, 1.71, 5.5])

ydata.append([207.5, 50.9, 0.098, 1.71, 5.5])

'''

import pandas as pd

data1 = pd.DataFrame(ydata, columns=["HeatFlux", "Superheat", "g", "SurfaceTension", "p"])

data1.to\_csv("./data/data1.csv", index=False, sep=",")

print(data1.shape)

**Appendix 6 – data2.ipynb**

ydata = [[44.1, 32.5, 0.098, 1.79, 5.5]]

ydata.append([47.4, 33.2, 0.098, 1.79, 5.5])

ydata.append([49.4, 34.2, 0.098, 1.79, 5.5])

ydata.append([59.2, 34.8, 0.098, 1.79, 5.5])

ydata.append([67.8, 36.3, 0.098, 1.79, 5.5])

ydata.append([73.6, 37.3, 0.098, 1.79, 5.5])

ydata.append([76.3, 37.8, 0.098, 1.79, 5.5])

ydata.append([85.3, 39.2, 0.098, 1.79, 5.5])

ydata.append([96.5, 39.3, 0.098, 1.79, 5.5])

ydata.append([111., 42.3, 0.098, 1.79, 5.5])

ydata.append([124., 43.5, 0.098, 1.79, 5.5])

ydata.append([136.2, 45.4, 0.098, 1.79, 5.5])

ydata.append([143.5, 46.7, 0.098, 1.79, 5.5])

ydata.append([154.6, 47.9, 0.098, 1.79, 5.5])

ydata.append([163.1, 48.6, 0.098, 1.79, 5.5])

ydata.append([172.8, 50.9, 0.098, 1.79, 5.5])

ydata.append([184.2, 51.7, 0.098, 1.79, 5.5])

ydata.append([203.7, 56.4, 0.098, 1.79, 5.5])

ydata.append([36.7, 30.2, 9.8, 1.79, 5.5])

ydata.append([55.1, 34.1, 9.8, 1.79, 5.5])

ydata.append([67.5, 35.3, 9.8, 1.79, 5.5])

ydata.append([78.0, 37.8, 9.8, 1.79, 5.5])

ydata.append([92.0, 38.1, 9.8, 1.79, 5.5])

ydata.append([120., 44.1, 9.8, 1.79, 5.5])

ydata.append([134.3, 46.9, 9.8, 1.79, 5.5])

ydata.append([150.3, 48.5, 9.8, 1.79, 5.5])

ydata.append([167., 49.2, 9.8, 1.79, 5.5])

ydata.append([184., 52.7, 9.8, 1.79, 5.5])

ydata.append([196.5, 53.1, 9.8, 1.79, 5.5])

ydata.append([42.4, 28.0, 19.6, 1.79, 9.5])

ydata.append([48.7, 29.3, 19.6, 1.79, 9.5])

ydata.append([54.5, 29.6, 19.6, 1.79, 9.5])

ydata.append([62.1, 28.5, 19.6, 1.79, 9.5])

ydata.append([70.8, 30.5, 19.6, 1.79, 9.5])

ydata.append([73.7, 30.3, 19.6, 1.79, 9.5])

ydata.append([81.8, 30.6, 19.6, 1.79, 9.5])

ydata.append([91.9, 34.5, 19.6, 1.79, 9.5])

ydata.append([103.9, 34.5, 19.6, 1.79, 9.5])

ydata.append([119.1, 35.4, 19.6, 1.79, 9.5])

ydata.append([133.7, 36.8, 19.6, 1.79, 9.5])

ydata.append([139.9, 38.1, 19.6, 1.79, 9.5])

ydata.append([148.3, 39.1, 19.6, 1.79, 9.5])

ydata.append([157.0, 40.0, 19.6, 1.79, 9.5])

ydata.append([169.1, 42.2, 19.6, 1.79, 9.5])

ydata.append([179.2, 43.2, 19.6, 1.79, 9.5])

ydata.append([205.0, 46.0, 19.6, 1.79, 9.5])

ydata.append([42.4, 29.7, 19.6, 1.79, 5.5])

ydata.append([48.7, 31.0, 19.6, 1.79, 5.5])

ydata.append([54.5, 31.2, 19.6, 1.79, 5.5])

ydata.append([70.8, 32.4, 19.6, 1.79, 5.5])

ydata.append([73.7, 31.4, 19.6, 1.79, 5.5])

ydata.append([81.8, 32.5, 19.6, 1.79, 5.5])

ydata.append([91.9, 36.3, 19.6, 1.79, 5.5])

ydata.append([103.9, 36.3, 19.6, 1.79, 5.5])

ydata.append([119.1, 37.2, 19.6, 1.79, 5.5])

ydata.append([133.7, 38.4, 19.6, 1.79, 5.5])

ydata.append([139.9, 39.7, 19.6, 1.79, 5.5])

ydata.append([148.3, 40.9, 19.6, 1.79, 5.5])

ydata.append([157.0, 41.6, 19.6, 1.79, 5.5])

ydata.append([169.1, 43.9, 19.6, 1.79, 5.5])

ydata.append([179.2, 45.0, 19.6, 1.79, 5.5])

ydata.append([205.0, 47.9, 19.6, 1.79, 5.5])

ydata.append([77.0, 41.5, 9.8, 0.00, 7.0])

ydata.append([71.0, 40.5, 9.8, 0.00, 7.0])

ydata.append([66.0, 39.5, 9.8, 0.00, 7.0])

ydata.append([62.0, 38.5, 9.8, 0.00, 7.0])

ydata.append([42.0, 34.0, 9.8, 0.00, 7.0])

ydata.append([60.0, 37.5, 9.8, 0.00, 7.0])

ydata.append([53.0, 37.0, 9.8, 0.00, 7.0])

ydata.append([71.7, 36.4, 0.098, 1.71, 5.5])

ydata.append([81.5, 38.5, 0.098, 1.71, 5.5])

ydata.append([90.7, 39.5, 0.098, 1.71, 5.5])

ydata.append([103.3, 41.6, 0.098, 1.71, 5.5])

ydata.append([117.0, 43.1, 0.098, 1.71, 5.5])

ydata.append([138.6, 45.4, 0.098, 1.71, 5.5])

ydata.append([161.7, 47.9, 0.098, 1.71, 5.5])

ydata.append([207.5, 50.9, 0.098, 1.71, 5.5])

import pandas as pd

data2 = pd.DataFrame(ydata, columns=["HeatFlux", "Superheat", "g", "SurfaceTension", "p"])

data2.to\_csv("./data/data2.csv", index=False, sep=",")

print(data2.shape)

**Appendix 7 – data3.ipynb**

import pandas as pd

import numpy as np

ydata = [[50.9, 37.2, 0.098, 1.79, 5.5], [75.1, 37.7, 0.098, 1.79, 5.5], [99.2, 39.7, 0.098, 1.79, 5.5], [147.7, 47.2, 0.098, 1.79, 5.5], [172.1, 49.7, 9.8, 1.79, 5.5], [49.2, 29.6, 19.6, 1.79, 9.5], [71.6, 30.0, 19.6, 1.79, 9.5], [129.8, 36.4, 19.6, 1.79, 9.5],[173.8, 42.8, 19.6, 1.79, 9.5], [68.7, 32.1, 19.6, 1.79, 5.5], [128.5, 38.0, 19.6, 1.79, 5.5], [69.0, 40.1, 9.8, 0.0, 7.0], [58.2, 37.1, 9.8, 0.0, 7.0], [100.3, 41.2, 0.098, 1.71, 5.5], [156.9, 47.4, 0.098, 1.71, 5.5]]

ydata = pd.DataFrame(ydata, columns=["HeatFlux", "Superheat", "g", "SurfaceTension", "p"])

# Constants

gen=9.8

const\_names = ["Tsat", "cpl", "hlv", "ul", "Prl", "rhol", "rhov", "sigma"]

consts = {

"5.5": [34.9, 4.18, 2418, 7.19e-4, 4.83, 994, 0.0397, 0.0706],

"7.0": [38.0, 4.18, 2406, 6.53e-4, 4.54, 993, 0.0476, 0.0692],

"9.5": [45.0, 4.18, 2394, 5.96e-4, 3.91, 990, 0.182, 0.0688]

}

const5\_5 = np.array([34.9, 4.18, 2418, 7.19e-4, 4.83, 994, 0.0397, 0.0706])

const7\_0 = np.array([38.0, 4.18, 2406, 6.53e-4, 4.54, 993, 0.0476, 0.0692])

const9\_5 = np.array([45.0, 4.18, 2394, 5.96e-4, 3.91, 990, 0.182, 0.0688])

# Read data

for n in const\_names:

ydata.insert(ydata.shape[1], n, 0)

# Add constants

ydata[const\_names] = ydata["p"].apply(lambda x: pd.Series(consts[str(x)]))

print(ydata.columns)

# non-dimensional constants

ydata["Qs"] = 10 \* ydata["HeatFlux"] / ydata["ul"] / ydata["hlv"] \* np.sqrt(ydata["sigma"]/gen/(ydata["rhol"]-ydata["rhov"]))\*10

ydata["Jas"] = 100 \* ydata["cpl"] \* ydata["Superheat"] / ydata["hlv"]

ydata["g/gen"] = ydata["g"] / gen

ydata.to\_csv("./data/data4.csv", index=False, sep=",")

print(ydata)

**Appendix 8 – Task1.ipynb**

# Read data from file

import pandas as pd

data1 = pd.read\_csv("./data/data1.csv")

# Separate the data

data1\_g0 = data1[data1["g"]==0.098]

data1\_g1 = data1[data1["g"]==9.8]

# Regression

from scipy import optimize

def f\_lin(x, a, b):

return a \* x + b

def f\_exp(x, c, d):

return c \* (x \*\* d)

def MSE(y, t):

return ((y-t)\*\*2).sum() / len(y)

a0, b0 = optimize.curve\_fit(f\_lin, data1\_g0["Superheat"], data1\_g0["HeatFlux"])[0]

a1, b1 = optimize.curve\_fit(f\_lin, data1\_g1["Superheat"], data1\_g1["HeatFlux"])[0]

c0, d0 = optimize.curve\_fit(f\_exp, data1\_g0["Superheat"], data1\_g0["HeatFlux"])[0]

c1, d1 = optimize.curve\_fit(f\_exp, data1\_g1["Superheat"], data1\_g1["HeatFlux"])[0]

y0\_lin = f\_lin(data1\_g0["Superheat"], a0, b0)

y1\_lin = f\_lin(data1\_g1["Superheat"], a1, b1)

MSE0\_lin = MSE(y0\_lin, data1\_g0["HeatFlux"])

MSE1\_lin = MSE(y1\_lin, data1\_g1["HeatFlux"])

y0\_exp = f\_exp(data1\_g0["Superheat"], c0, d0)

y1\_exp = f\_exp(data1\_g1["Superheat"], c1, d1)

MSE0\_exp = MSE(y0\_exp, data1\_g0["HeatFlux"])

MSE1\_exp = MSE(y1\_exp, data1\_g1["HeatFlux"])

print("Linear Regression MSE:")

print("g=0.098:")

print("y0 = %f \* x0 + %f, e = %f" % (a0, b0, MSE0\_lin))

print("y1 = %f \* x1 + %f, e = %f" % (a1, b1, MSE1\_lin))

print("Exponential Regression MSE:")

print("y0 = %f \* x0 ^(%f), e = %f" % (c0, d0, MSE0\_exp))

print("y1 = %f \* x1 ^(%f), e = %f" % (c1, d1, MSE1\_exp))

# Plot

import matplotlib.pyplot as plt

import numpy as np

plt.figure(figsize=(10,16))

r = np.linspace(30, 58, 100)

# Linear plot

plt.subplot(211)

plt.title("HeatFlux-Superheat Plot")

plt.xlabel("Superheat")

plt.ylabel("HeatFlux")

plt.subplots\_adjust(hspace=0.5)

plt.plot(data1\_g0["Superheat"], data1\_g0["HeatFlux"], '.', color="b", label="g=0.098")

plt.plot(r, f\_lin(r, a0, b0), color="b")

plt.plot(data1\_g1["Superheat"], data1\_g1["HeatFlux"], 'x', color="r", label="g=9.8")

plt.plot(r, f\_lin(r, a1, b1), color="r")

plt.legend()

# Log plot

plt.subplot(212)

plt.title("log(HeatFlux)-log(Superheat) Plot")

plt.xlabel("log(Superheat)")

plt.ylabel("log(HeatFlux)")

plt.loglog(data1\_g0["Superheat"], data1\_g0["HeatFlux"], '.', color="b", label="g=0.098")

plt.loglog(r, f\_exp(r, c0, d0), color="b")

plt.loglog(data1\_g1["Superheat"], data1\_g1["HeatFlux"], 'x', color="r", label="g=9.8")

plt.loglog(r, f\_exp(r, c1, d1), color="r")

plt.legend()

plt.savefig("./result/heatflux\_superheat.png")

**Appendix 9 – Task2.ipynb**

from random import random

from random import seed

seed(1)

from sklearn.linear\_model import LinearRegression

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Config

MFRAC = 0.5

EPOCH = 6000

# Initial guess

ni = np.array([0.00027, 4.0, 0.063, 1.215, 0.145])

delta\_n = np.array([0.0001, 0.001, 0.0001, 0.0001, 0.0001])

# Mutation step

mutation = np.array([0.09, 0.09, 0.09, 0.09, 0.09])

mutation\_decay = 1e-4

# Read data

ydata = pd.read\_csv("./data/data1.csv")

ND = ydata.shape[0] #number of data vectors in array

DI = ydata.shape[1] #number of data items in vector

NS = ND #total number of DNA strands

# log

lydata = ydata.copy()

lydata = np.log(lydata + 1e-12)

# Functions needed

# Initialize n

def initialize(ni, delta\_n):

# Initializing n

n = np.array([(ni + [random() for i in range(DI)] \* delta\_n) for i in range(ND)] ) # 45x5 randomized initial value

return n

# Relative Absolute Error

def AFERR(n, lydata):

Ferr = -lydata["HeatFlux"] + np.log(n[:, 0]) + n[:, 1] \* lydata["Superheat"] + n[:, 2] \* lydata["g"]

aFerr = np.abs(Ferr) / np.abs(lydata["HeatFlux"])

return aFerr

# Error and Selection

def selection(n, lydata, MFRAC=MFRAC):

# Error calculation

aFerr = AFERR(n, lydata)

aFerrMean = np.mean(aFerr)

aFerrMedian = np.median(aFerr)

# Sorting

sorted\_id = np.argsort(aFerr)

aFerr[:] = aFerr[sorted\_id]

n[:] = n[sorted\_id]

# Selection

clim = MFRAC \* aFerrMedian

nkeep = np.searchsorted(aFerr, clim)

return n, nkeep

# Mating

def mating(sorted\_n, nkeep, mutation, mutation\_decay, epoch=0):

if nkeep == sorted\_n.shape[0]:

print("EPOCH %d: NOT MATING"%(epoch))

return sorted\_n

for row in sorted\_n[nkeep:, :]:

# Randomly choose 2 parents

parents = [sorted\_n[np.random.randint(0, nkeep+1)] for i in range(2)]

# Choose which parent to inherit from

# inherit[i] = 0 or 1

inherit = [np.random.randint(0, 2) for i in range(DI)]

r = [parents[inherit[i]][i] for i in range(DI)]

# Mutation

mut\_rate = [(2\*(0.5-np.random.rand())) for i in range(DI)]

row[:] = r \* (1 + mut\_rate \* mutation)

if mutation\_decay:

mutation[:] = mutation \* (1-mutation\_decay)

return sorted\_n

# Train

def train(EPOCH=300, ni=ni, delta\_n=delta\_n, MFRAC=MFRAC, mutation=mutation, mutation\_decay=0):

n = initialize(ni, delta\_n)

n\_means, aFerr\_means = [], []

n\_best = np.array([0.0 for i in range(DI)])

aFerr\_min, best\_epoch = 100, 0

for epoch in range(EPOCH):

sorted\_n, nkeep = selection(n, lydata, MFRAC)

n = mating(sorted\_n, nkeep, mutation, mutation\_decay, epoch)

# statistic features

n\_mean = np.mean(n, axis=0)

aFerr\_mean = np.mean(AFERR(n\_mean.reshape(1,-1), lydata))

n\_means.append(n\_mean)

aFerr\_means.append(aFerr\_mean)

if aFerr\_mean < aFerr\_min:

aFerr\_min = aFerr\_mean

n\_best[:] = n\_mean

best\_epoch = epoch

# Result

print("ENDING: n1: %f, n2: %f, n3: %f, aFerrmean: %f" %(n\_mean[0], n\_mean[1], n\_mean[2], aFerr\_mean))

print("OPTIM: n1: %f, n2: %f, n3: %f, aFerrmean: %f, BestEpoch: %d" %(n\_best[0], n\_best[1], n\_best[2], aFerr\_min, best\_epoch))

return n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch

# Plotting

def const\_error(n\_means, aFerr\_means, i=-1, sub=111):

n\_means = np.array(n\_means)

x = range(n\_means.shape[0])

# Plotting

plt.subplot(sub)

if i >= 0:

plt.title("Initial Guess: {}\nMutation Magnitude: {}".format(nis[i], mutations[i]))

plt.xlabel("Epochs")

plt.ylabel("Constants and Error")

plt.plot(x, n\_means[:, 0], label="n1")

plt.plot(x, n\_means[:, 1], label="n2")

plt.plot(x, n\_means[:, 2], label="n3")

plt.plot(x, aFerr\_means, label="aFerr")

plt.legend()

plt.loglog()

def pred\_true(n, label):

# Calculating y\_pred

n = n.reshape(1,-1)

y\_pred = np.log(n[:, 0]) + n[:, 1] \* lydata["Superheat"] + n[:, 2] \* lydata["g"]

y\_pred = np.exp(y\_pred)

y\_true = np.exp(lydata["HeatFlux"])

# Plotting

plt.scatter(y\_pred, y\_true, label=label)

# Deviation and other statistic results

def stat(n):

n = n.reshape(1,-1)

y\_pred = np.exp(np.log(n[:, 0]) + n[:, 1] \* lydata["Superheat"] + n[:, 2] \* lydata["g"])

y\_true = np.exp(lydata["HeatFlux"])

relErr = (y\_pred-y\_true) / y\_true

mean = np.sqrt((np.mean(relErr \* relErr) / relErr.shape[0]))

dev = np.std(relErr)

err\_rate = np.where((relErr > 0.1)| (relErr < -0.1))[0].shape[0] / relErr.shape[0]

print("RMSError: %f, RMSDeviation: %f"%(mean, dev))

return mean, dev, err\_rate

# Main

# optimal: 0.0007726, 3.188, 0.05206

nis = [

np.array([0.00027, 4.0, 0.063, 0, 0]),

np.array([0.003, 10, 1, 0, 0]),

np.array([1, 20, 5, 0, 0]),

np.array([2, 100, 20, 0, 0])

]

mutations = [

np.array([0.09 for i in range(DI)]),

np.array([0.2 for i in range(DI)]), # doesn't converge for 0.09

np.array([0.5 for i in range(DI)]),

np.array([0.8 for i in range(DI)])

]

results = []

for i in range(4):

res = train(EPOCH, nis[i], delta\_n, MFRAC, mutations[i])

results.append(res)

# Plotting

plt.figure(figsize=(12,15))

for (i, (n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch)) in enumerate(results):

const\_error(n\_means, aFerr\_means, i, 221+i)

plt.savefig("./result/const\_and\_error.png")

# Bad initial value case

# May never converge

ni = np.array([1, 0.05, 10, 0, 0])

mutation = np.array([0.8 for i in range(DI)])

n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch = train(EPOCH, ni, delta\_n, MFRAC, mutation)

const\_error(n\_means, aFerr\_means)

plt.title("Initial Guess: {}\nMutation Magnitude: {}".format(ni, mutation))

plt.savefig("./result/const\_and\_error\_slope.png")

# Linear Regression Result

x = lydata[["Superheat", "g"]]

y = lydata["HeatFlux"]

model = LinearRegression()

model = model.fit(x, y)

n\_lr = np.array([[np.exp(model.intercept\_), model.coef\_[0], model.coef\_[1]]])

# y\_pred - y\_true

n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch = results[1]

plt.figure(figsize=(8,4))

pred\_true(n\_best, "Genetic Algorithm")

pred\_true(n\_lr, "Linear Regression")

r = np.linspace(10, 1000, 3)

y1 = 1.1 \* r

y2 = 0.9 \* r

plt.fill\_between(r, y1, y2, color="r", alpha=0.2, label="Acceptable Region")

plt.title('Genetic Algorithm')

plt.xlabel('predicted heat flux (W/cm^2)')

plt.ylabel('measured heat flux (W/cm^2)')

plt.loglog()

plt.legend(loc="lower right")

plt.xlim(xmax = 1000, xmin = 10)

plt.ylim(ymax = 1000, ymin = 10)

plt.savefig("./result/pred\_true.png")

stat(n\_best)

stat(n\_lr)

nis = [

np.array([0.00027, 4.0, 0.063, 0, 0]),

np.array([0.003, 10, 1, 0, 0]),

np.array([1, 20, 5, 0, 0]),

np.array([2, 100, 20, 0, 0])

]

mutations = [

np.array([0.09 for i in range(DI)]),

np.array([0.09 for i in range(DI)]), # doesn't converge for 0.09

np.array([0.09 for i in range(DI)]),

np.array([0.09 for i in range(DI)])

]

results = []

for i in range(4):

res = train(EPOCH, nis[i], delta\_n, MFRAC, mutations[i])

results.append(res)

# Plotting

plt.figure(figsize=(12,15))

for (i, (n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch)) in enumerate(results):

const\_error(n\_means, aFerr\_means, i, 221+i)

plt.savefig("./result/const\_and\_error\_bad.png")

EPOCH = 50000

train(EPOCH, np.array([2, 100, 20, 0, 0]), delta\_n, MFRAC, np.array([0.09 for i in range(DI)]))

**Appendix 10 – Task2\_LR.ipynb**

from sklearn.linear\_model import LinearRegression

import numpy as np

import pandas as pd

ydata = pd.read\_csv("./data/data1.csv")

lydata = ydata.copy()

lydata = np.log(lydata + 1e-12)

x = lydata[["Superheat", "g"]]

y = lydata["HeatFlux"]

# Model

model = LinearRegression()

model = model.fit(x, y)

print(np.exp(model.intercept\_))

print(model.coef\_)

def AFERR(n, lydata):

Ferr = -lydata["HeatFlux"] + np.log(n[:, 0]) + n[:, 1] \* lydata["Superheat"] + n[:, 2] \* lydata["g"]

aFerr = np.abs(Ferr) / np.abs(lydata["HeatFlux"])

return aFerr

n = np.array([[np.exp(model.intercept\_), model.coef\_[0], model.coef\_[1]]])

print(np.mean(AFERR(n, lydata)))

**Appendix 11 – Task3.ipynb**

from random import random

from random import seed

seed(1)

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Config

MFRAC = 0.5

EPOCH = 6000

gen = 9.8

# Initial guess

ni = np.array([0.000476, 3.028, 0.2249, 1.054, 0.217])

delta\_n = np.array([0.0001, 0.001, 0.0001, 0.0001, 0.0001])

# Mutation step

mutation = np.array([0.09, 0.09, 0.09, 0.09, 0.09])

mutation\_decay = 1e-4

# Read data

ydata = pd.read\_csv("./data/data2.csv")

ND = ydata.shape[0] #number of data vectors in array

DI = ydata.shape[1] #number of data items in vector

NS = ND #total number of DNA strands

# log

lydata = ydata.copy()

lydata = np.log(lydata + 1e-12)

# Functions needed

# Initialize n

def initialize(ni, delta\_n):

# Initializing n

n = np.array([(ni + [random() for i in range(DI)] \* delta\_n) for i in range(ND)] ) # 45x5 randomized initial value

return n

# Relative Absolute Error

def AFERR(n):

Ferr = -lydata["HeatFlux"] + np.log(n[:, 0]) + n[:, 1] \* lydata["Superheat"] + n[:, 2] \* np.log(ydata["g"] + n[:, 3]\*gen\*ydata["SurfaceTension"]) + n[:, 4]\*lydata["p"]

aFerr = np.abs(Ferr) / np.abs(lydata["HeatFlux"])

return aFerr

# Error and Selection

def selection(n, MFRAC=MFRAC):

# Error calculation

aFerr = AFERR(n)

aFerrMean = np.mean(aFerr)

aFerrMedian = np.median(aFerr)

# Sorting

sorted\_id = np.argsort(aFerr)

aFerr[:] = aFerr[sorted\_id]

n[:] = n[sorted\_id]

# Selection

clim = MFRAC \* aFerrMedian

nkeep = np.searchsorted(aFerr, clim)

return n, nkeep

# Mating

def mating(sorted\_n, nkeep, mutation, mutation\_decay, epoch=0):

if nkeep == sorted\_n.shape[0]:

print("EPOCH %d: NOT MATING"%(epoch))

return sorted\_n

for row in sorted\_n[nkeep:, :]:

# Randomly choose 2 parents

parents = [sorted\_n[np.random.randint(0, nkeep+1)] for i in range(2)]

# Choose which parent to inherit from

# inherit[i] = 0 or 1

inherit = [np.random.randint(0, 2) for i in range(DI)]

r = [parents[inherit[i]][i] for i in range(DI)]

# Mutation

mut\_rate = [(2\*(0.5-np.random.rand())) for i in range(DI)]

row[:] = r \* (1 + mut\_rate \* mutation)

if mutation\_decay:

mutation[:] = mutation \* (1-mutation\_decay)

return sorted\_n

# Train

def train(EPOCH=300, ni=ni, delta\_n=delta\_n, MFRAC=MFRAC, mutation=mutation, mutation\_decay=0):

n = initialize(ni, delta\_n)

n\_means, aFerr\_means = [], []

n\_best = np.array([0.0 for i in range(DI)])

aFerr\_min, best\_epoch = 1, 0

for epoch in range(EPOCH):

sorted\_n, nkeep = selection(n, MFRAC)

n = mating(sorted\_n, nkeep, mutation, mutation\_decay, epoch)

# statistic features

n\_mean = np.mean(n, axis=0)

aFerr\_mean = np.mean(AFERR(n\_mean.reshape(1,-1)))

n\_means.append(n\_mean)

aFerr\_means.append(aFerr\_mean)

if aFerr\_mean < aFerr\_min:

aFerr\_min = aFerr\_mean

n\_best[:] = n\_mean

best\_epoch = epoch

# Result

print("ENDING: n: {}, aFerrmean: {}".format(n\_mean, aFerr\_mean))

print("OPTIM: n: {}, aFerrmean: {}, BestEpoch: {}".format(n\_best, aFerr\_min, best\_epoch))

return n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch

# Plotting

def const\_error(n\_means, aFerr\_means, i=-1, sub=111):

n\_means = np.array(n\_means)

x = range(n\_means.shape[0])

# Plotting

plt.subplot(sub)

if i >= 0:

plt.title("Initial Guess: {}\nMutation Magnitude: {}".format(nis[i], mutations[i]))

plt.xlabel("Epochs")

plt.ylabel("Constants and Error")

plt.plot(x, n\_means[:, 0], label="n1")

plt.plot(x, n\_means[:, 1], label="n2")

plt.plot(x, n\_means[:, 2], label="n3")

plt.plot(x, n\_means[:, 3], label="n4")

plt.plot(x, n\_means[:, 4], label="n5")

plt.plot(x, aFerr\_means, label="aFerr")

plt.legend()

plt.loglog()

def pred\_true(n, label):

# Calculating y\_pred

n = n.reshape(1,-1)

y\_pred = np.log(n[:, 0]) + n[:, 1] \* lydata["Superheat"] + n[:, 2] \* np.log(ydata["g"] + n[:, 3]\*gen\*ydata["SurfaceTension"]) + n[:, 4]\*lydata["p"]

y\_pred = np.exp(y\_pred)

y\_true = ydata["HeatFlux"]

# Plotting

plt.scatter(y\_pred, y\_true, label=label)

# Deviation and other statistic results

def stat(n):

n = n.reshape(1,-1)

y\_pred = np.exp(np.log(n[:, 0]) + n[:, 1] \* lydata["Superheat"] + n[:, 2] \* np.log(ydata["g"] + n[:, 3]\*gen\*ydata["SurfaceTension"]) + n[:, 4]\*lydata["p"])

y\_true = ydata["HeatFlux"]

relErr = (y\_pred-y\_true) / y\_true

mean = np.sqrt((np.mean(relErr \* relErr) / relErr.shape[0]))

dev = np.std(relErr)

unacc\_rate = np.where((relErr > 0.1)| (relErr < -0.1))[0].shape[0] / relErr.shape[0]

print("RMSError: %f, RMSDeviation: %f"%(mean, dev))

return mean, dev

# Main

nis = [

np.array([0.000476, 3.028, 0.2249, 1.054, 0.217]),

np.array([0.0008, 2, 0.5, 2, 1]),

np.array([0.0003, 1.5, 0.8, 3, 1.2]),

np.array([0.0002, 3, 0.3, 2.2, 0.7])

]

mutations = [

np.array([0.09 for i in range(DI)]),

np.array([0.09 for i in range(DI)]), # doesn't converge for 0.09

np.array([0.09 for i in range(DI)]),

np.array([0.09 for i in range(DI)])

]

results = []

for i in range(4):

res = train(EPOCH, nis[i], delta\_n, MFRAC, mutations[i])

results.append(res)

# Plotting

plt.figure(figsize=(12,15))

for (i, (n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch)) in enumerate(results):

const\_error(n\_means, aFerr\_means, i, 221+i)

plt.savefig("./result/const\_and\_error\_5.png")

# n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch = results[2]

# np.save("./result/gene5\_0", n\_best)

n\_best = np.load("./result/gene5\_0.npy")

plt.figure(figsize=(8,4))

pred\_true(n\_best, "Genetic Algorithm")

r = np.linspace(10, 1000, 3)

y1 = 1.1 \* r

y2 = 0.9 \* r

plt.fill\_between(r, y1, y2, color="r", alpha=0.2, label="Acceptable Region")

plt.title('Genetic Algorithm')

plt.xlabel('predicted heat flux (W/cm^2)')

plt.ylabel('measured heat flux (W/cm^2)')

plt.loglog()

plt.legend(loc="lower right")

plt.xlim(xmax = 1000, xmin = 10)

plt.ylim(ymax = 1000, ymin = 10)

plt.savefig("./result/pred\_true\_5.png")

stat(n\_best)

from mpl\_toolkits.mplot3d import Axes3D

# Create a figure

fig = plt.figure(figsize=(10, 10))

ax = Axes3D(fig)

n1, n2, n3, n4, n5 = n\_best

x = np.linspace(1, 20, 1000)

y = np.linspace(0.001, 2, 1000)

X, Y = np.meshgrid(x, y)

Z = n1 \* np.power((X + n4 \* gen \* Y), n3) \* np.power(10, n5)

ax.plot\_surface(X, Y, Z)

ax.set\_xlabel("g(m/s^2)")

ax.set\_ylabel("gamma")

ax.set\_zlabel("q/delta\_T^(n2)")

plt.title("q/delta\_T^(n2) - g, gamma")

plt.savefig("./result/q\_g\_gamma.png")

from mpl\_toolkits.mplot3d import Axes3D

# Create a figure

fig = plt.figure(figsize=(10, 10))

ax = Axes3D(fig)

n1, n2, n3, n4, n5 = n\_best

x = np.linspace(1, 20, 1000)

y = np.linspace(0.001, 2, 1000)

X, Y = np.meshgrid(x, y)

Z = n1 \* np.power((X + n4 \* gen \* Y), n3) \* np.power(10, n5)

logX = np.log(X)

logY = np.log(Y)

logZ = np.log(Z)

ax.plot\_surface(logX, logY, logZ)

ax.set\_xlabel("log(g)")

ax.set\_ylabel("log(gamma)")

ax.set\_zlabel("log(q/delta\_T^(n2))")

plt.title("log(q/delta\_T^(n2)) - log(g), log(gamma)")

plt.savefig("./result/q\_g\_gamma\_log.png")

**Appendix 12 – Task4.ipynb**

import pandas as pd

import numpy as np

# Constants

gen=9.8

const\_names = ["Tsat", "cpl", "hlv", "ul", "Prl", "rhol", "rhov", "sigma"]

consts = {

"5.5": [34.9, 4.18, 2418, 7.19e-4, 4.83, 994, 0.0397, 0.0706],

"7.0": [38.0, 4.18, 2406, 6.53e-4, 4.54, 993, 0.0476, 0.0692],

"9.5": [45.0, 4.18, 2394, 5.96e-4, 3.91, 990, 0.182, 0.0688]

}

const5\_5 = np.array([34.9, 4.18, 2418, 7.19e-4, 4.83, 994, 0.0397, 0.0706])

const7\_0 = np.array([38.0, 4.18, 2406, 6.53e-4, 4.54, 993, 0.0476, 0.0692])

const9\_5 = np.array([45.0, 4.18, 2394, 5.96e-4, 3.91, 990, 0.182, 0.0688])

# Read data

ydata = pd.read\_csv("./data/data2.csv")

for n in const\_names:

ydata.insert(ydata.shape[1], n, 0)

# Add constants

ydata[const\_names] = ydata["p"].apply(lambda x: pd.Series(consts[str(x)]))

print(ydata.columns)

# non-dimensional constants

ydata["Qs"] = 10 \* ydata["HeatFlux"]\*10000 / ydata["ul"] / (ydata["hlv"]\*1000) \* np.sqrt(ydata["sigma"]/gen/(ydata["rhol"]-ydata["rhov"]))

ydata["Jas"] = 100 \* ydata["cpl"] \* 1000 \* ydata["Superheat"] / (ydata["hlv"]\*1000)

ydata["g/gen"] = ydata["g"] / gen

ydata = ydata[["Qs", "Jas", "Prl", "g/gen", "SurfaceTension"]]

ydata.to\_csv("./data/data3.csv", index=False, sep=",")

print(ydata)

**Appendix 13 – Task5.ipynb**

from random import random

from random import seed

seed(1)

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Config

MFRAC = 0.3

EPOCH = 6000

gen = 9.8

# Initial guess

ni = np.array([0.000476, 3.028, 0.2249, 1.054, 0.217])

delta\_n = np.array([0.0001, 0.001, 0.0001, 0.0001, 0.0001])

# Mutation step

mutation = np.array([0.09, 0.09, 0.09, 0.09, 0.09])

mutation\_decay = 1e-4

# Read data

ydata = pd.read\_csv("./data/data3.csv")

ND = ydata.shape[0] #number of data vectors in array

DI = ydata.shape[1] #number of data items in vector

NS = ND #total number of DNA strands

# log

lydata = ydata.copy()

lydata = np.log(lydata + 1e-12)

# Functions needed

# Initialize n

def initialize(ni, delta\_n):

# Initializing n

n = np.array([(ni + [random() for i in range(DI)] \* delta\_n) for i in range(ND)] ) # 77x5 randomized initial value

return n

# Relative Absolute Error

def AFERR(n):

Ferr = -lydata["Qs"] + np.log(n[:, 0]) + n[:, 1] \* lydata["Jas"] + n[:, 2] \* np.log(ydata["g/gen"] + n[:, 3]\*ydata["SurfaceTension"]) - n[:, 4]\*lydata["Prl"]

aFerr = np.abs(Ferr / lydata["Qs"])

return aFerr

# Error and Selection

def selection(n, MFRAC=MFRAC):

# Error calculation

aFerr = AFERR(n)

aFerrMean = np.mean(aFerr)

aFerrMedian = np.median(aFerr)

# Sorting

sorted\_id = np.argsort(aFerr)

aFerr[:] = aFerr[sorted\_id]

n[:] = n[sorted\_id]

# Selection

clim = MFRAC \* aFerrMedian

nkeep = np.searchsorted(aFerr, clim)

return n, nkeep

# Mating

def mating(sorted\_n, nkeep, mutation, mutation\_decay, epoch=0):

if nkeep == sorted\_n.shape[0]:

print("EPOCH %d: NOT MATING"%(epoch))

return sorted\_n

for row in sorted\_n[nkeep:, :]:

# Randomly choose 2 parents

parents = [sorted\_n[np.random.randint(0, nkeep+1)] for i in range(2)]

# Choose which parent to inherit from

# inherit[i] = 0 or 1

inherit = [np.random.randint(0, 2) for i in range(DI)]

r = [parents[inherit[i]][i] for i in range(DI)]

# Mutation

mut\_rate = [(2\*(0.5-np.random.rand())) for i in range(DI)]

row[:] = r \* (1 + mut\_rate \* mutation)

if mutation\_decay:

mutation[:] = mutation \* (1-mutation\_decay)

return sorted\_n

# Train

def train(EPOCH=300, ni=ni, delta\_n=delta\_n, MFRAC=MFRAC, mutation=mutation, mutation\_decay=0):

n = initialize(ni, delta\_n)

n\_means, aFerr\_means = [], []

n\_best = np.array([0.0 for i in range(DI)])

aFerr\_min, best\_epoch = 1, 0

for epoch in range(EPOCH):

sorted\_n, nkeep = selection(n, MFRAC)

n = mating(sorted\_n, nkeep, mutation, mutation\_decay, epoch)

# statistic features

n\_mean = np.mean(n, axis=0)

aFerr\_mean = np.mean(AFERR(n\_mean.reshape(1,-1)))

n\_means.append(n\_mean)

aFerr\_means.append(aFerr\_mean)

if aFerr\_mean < aFerr\_min:

aFerr\_min = aFerr\_mean

n\_best[:] = n\_mean

best\_epoch = epoch

# Result

print("ENDING: n: {}, aFerrmean: {}".format(n\_mean, aFerr\_mean))

print("OPTIM: n: {}, aFerrmean: {}, BestEpoch: {}".format(n\_best, aFerr\_min, best\_epoch))

return n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch

# Plotting

def const\_error(n\_means, aFerr\_means, i=-1, sub=111):

n\_means = np.array(n\_means)

x = range(n\_means.shape[0])

# Plotting

plt.subplot(sub)

if i >= 0:

plt.title("Initial Guess: {}\nMutation Magnitude: {}".format(nis[i], mutations[i]))

plt.xlabel("Epochs")

plt.ylabel("Constants and Error")

plt.plot(x, n\_means[:, 0], label="n1")

plt.plot(x, n\_means[:, 1], label="n2")

plt.plot(x, n\_means[:, 2], label="n3")

plt.plot(x, n\_means[:, 3], label="n4")

plt.plot(x, n\_means[:, 4], label="n5")

plt.plot(x, aFerr\_means, label="aFerr")

plt.legend()

plt.loglog()

def pred\_true(n, label):

# Calculating y\_pred

n = n.reshape(1,-1)

y\_pred = np.log(n[:, 0]) + n[:, 1] \* lydata["Jas"] + n[:, 2] \* np.log(ydata["g/gen"] + n[:, 3]\*ydata["SurfaceTension"]) - n[:, 4]\*lydata["Prl"]

y\_pred = np.exp(y\_pred)

y\_true = ydata["Qs"]

# Plotting

plt.scatter(y\_pred, y\_true, label=label)

# Deviation and other statistic results

def stat(n):

n = n.reshape(1,-1)

y\_pred = np.log(n[:, 0]) + n[:, 1] \* lydata["Jas"] + n[:, 2] \* np.log(ydata["g/gen"] + n[:, 3]\*ydata["SurfaceTension"]) - n[:, 4]\*lydata["Prl"]

y\_pred = np.exp(y\_pred)

y\_true = ydata["Qs"]

relErr = (y\_pred-y\_true) / y\_true

mean = np.sqrt((np.mean(relErr \* relErr) / relErr.shape[0]))

dev = np.std(relErr)

unacc\_rate = np.where((relErr > 0.1)| (relErr < -0.1))[0].shape[0] / relErr.shape[0]

print("RMSError: %f, RMSDeviation: %f"%(mean, dev))

return mean, dev, unacc\_rate

# Main

nis = [

np.array([0.125, 2.5, 0.42, 1.3, 1.73]),

np.array([0.110, 2.77, 0.317, 1.26, 1.95]),

np.array([0.15, 3, 0.5, 2, 2]),

np.array([0.14, 2.8, 0.45, 1.5, 1.8])

]

mutations = [

np.array([0.06 for i in range(DI)]),

np.array([0.06 for i in range(DI)]),

np.array([0.06 for i in range(DI)]),

np.array([0.06 for i in range(DI)])

]

results = []

for i in range(4):

res = train(EPOCH, nis[i], delta\_n, MFRAC, mutations[i])

results.append(res)

# Plotting

plt.figure(figsize=(12,15))

for (i, (n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch)) in enumerate(results):

const\_error(n\_means, aFerr\_means, i, 221+i)

plt.savefig("./result/const\_and\_error\_5\_2.png")

# n\_means, aFerr\_means, n\_best, aFerr\_min, best\_epoch = results[2]

# np.save("./result/gene5\_0\_2", n\_best)

n\_best = np.load("./result/gene5\_0\_2.npy")

plt.figure(figsize=(8,4))

pred\_true(n\_best, "Genetic Algorithm")

r = np.linspace(3, 80, 3)

y1 = 1.1 \* r

y2 = 0.9 \* r

plt.fill\_between(r, y1, y2, color="r", alpha=0.2, label="Acceptable Region")

plt.title('Genetic Algorithm')

plt.xlabel('predicted Qs')

plt.ylabel('measured Qs')

plt.loglog()

plt.legend(loc="lower right")

plt.xlim(xmax = 80, xmin = 3)

plt.ylim(ymax = 80, ymin = 3)

plt.savefig("./result/pred\_true\_5\_2.png")

stat(n\_best)

from mpl\_toolkits.mplot3d import Axes3D

# Create a figure

fig = plt.figure(figsize=(10, 10))

ax = Axes3D(fig)

n1, n2, n3, n4, n5 = n\_best

x = np.linspace(0.01, 2, 1000)

y = np.linspace(0.001, 2, 1000)

X, Y = np.meshgrid(x, y)

Z = n1 \* np.power((X + n4 \* Y), n3)

ax.plot\_surface(X, Y, Z)

ax.set\_xlabel("g/gen")

ax.set\_ylabel("gamma")

ax.set\_zlabel("Qs\*Prl^n5/Jas^n2")

plt.savefig("./result/q\_g\_gamma\_2.png")

from mpl\_toolkits.mplot3d import Axes3D

# Create a figure

fig = plt.figure(figsize=(10, 10))

ax = Axes3D(fig)

n1, n2, n3, n4, n5 = n\_best

x = np.linspace(0.01, 2, 1000)

y = np.linspace(0.001, 2, 1000)

X, Y = np.meshgrid(x, y)

Z = n1 \* np.power((X + n4 \* Y), n3)

logX = np.log(X)

logY = np.log(Y)

logZ = np.log(Z)

ax.plot\_surface(logX, logY, logZ)

ax.set\_xlabel("log(g/gen)")

ax.set\_ylabel("log(gamma)")

ax.set\_zlabel("log(Qs\*Prl^n5/Jas^n2)")

plt.savefig("./result/q\_g\_gamma\_log\_2.png")

def pred\_true\_3(n, label):

# Calculating y\_pred

n = n.reshape(1,-1)

y\_pred = np.log(n[:, 0]) + n[:, 1] \* lvalid["Superheat"] + n[:, 2] \* np.log(valid["g"] + n[:, 3]\*gen\*valid["SurfaceTension"]) + n[:, 4]\*lvalid["p"]

y\_pred = np.exp(y\_pred)

y\_true = valid["HeatFlux"]

# Plotting

plt.scatter(y\_pred, y\_true, label=label)

def pred\_true\_5(n, label):

# Calculating y\_pred

n = n.reshape(1,-1)

y\_pred = np.log(n[:, 0]) + n[:, 1] \* lvalid["Jas"] + n[:, 2] \* np.log(valid["g/gen"] + n[:, 3]\*valid["SurfaceTension"]) - n[:, 4]\*lvalid["Prl"]

y\_pred = np.exp(y\_pred)

y\_pred = y\_pred / valid["Qs"] \* valid["HeatFlux"]

y\_true = valid["HeatFlux"]

# Plotting

plt.scatter(y\_pred, y\_true, label=label)

# Deviation and other statistic results

def stat\_3(n):

n = n.reshape(1,-1)

y\_pred = np.exp(np.log(n[:, 0]) + n[:, 1] \* lvalid["Superheat"] + n[:, 2] \* np.log(valid["g"] + n[:, 3]\*gen\*valid["SurfaceTension"]) + n[:, 4]\*lvalid["p"])

y\_true = valid["HeatFlux"]

relErr = (y\_pred-y\_true) / y\_true

mean = np.sqrt((np.mean(relErr \* relErr) / relErr.shape[0]))

dev = np.std(relErr)

unacc\_rate = np.where((relErr > 0.1)| (relErr < -0.1))[0].shape[0] / relErr.shape[0]

print("RMSError: %f, RMSDeviation: %f"%(mean, dev))

return mean, dev, unacc\_rate

def stat\_5(n):

n = n.reshape(1,-1)

y\_pred = np.log(n[:, 0]) + n[:, 1] \* lvalid["Jas"] + n[:, 2] \* np.log(valid["g/gen"] + n[:, 3]\*valid["SurfaceTension"]) - n[:, 4]\*lvalid["Prl"]

y\_pred = np.exp(y\_pred)

y\_pred = y\_pred / valid["Qs"] \* valid["HeatFlux"]

y\_true = valid["HeatFlux"]

relErr = (y\_pred-y\_true) / y\_true

mean = np.sqrt((np.mean(relErr \* relErr) / relErr.shape[0]))

dev = np.std(relErr)

unacc\_rate = np.where((relErr > 0.1)| (relErr < -0.1))[0].shape[0] / relErr.shape[0]

print("RMSError: %f, RMSDeviation: %f"%(mean, dev))

return mean, dev, unacc\_rate

valid = pd.read\_csv("./data/data4.csv")

lvalid = np.log(valid + 1e-12)

n\_t3 = np.load("./result/gene5\_0.npy")

n\_t5 = np.load("./result/gene5\_0\_2.npy")

plt.figure(figsize=(8,4))

pred\_true\_5(n\_t5, "Task-5 Model")

pred\_true\_3(n\_t3, "Task-3 Model")

r = np.linspace(10, 1000, 3)

y1 = 1.1 \* r

y2 = 0.9 \* r

plt.fill\_between(r, y1, y2, color="r", alpha=0.2, label="Acceptable Region")

plt.title('Genetic Algorithm')

plt.xlabel('predicted heat flux (W/cm^2)')

plt.ylabel('measured heat flux (W/cm^2)')

plt.loglog()

plt.legend(loc="lower right")

plt.xlim(xmax = 1000, xmin = 10)

plt.ylim(ymax = 1000, ymin = 10)

plt.savefig("./result/pred\_true\_val.png")

stat\_5(n\_t5)

stat\_3(n\_t3)