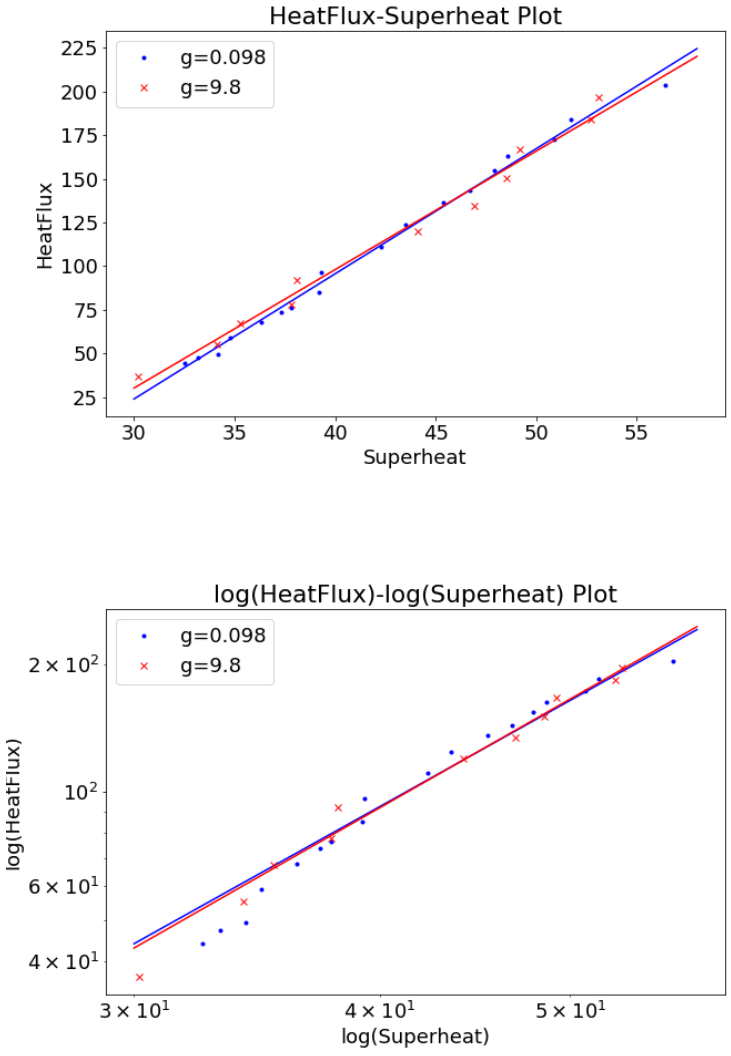
Report for Project 1

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.

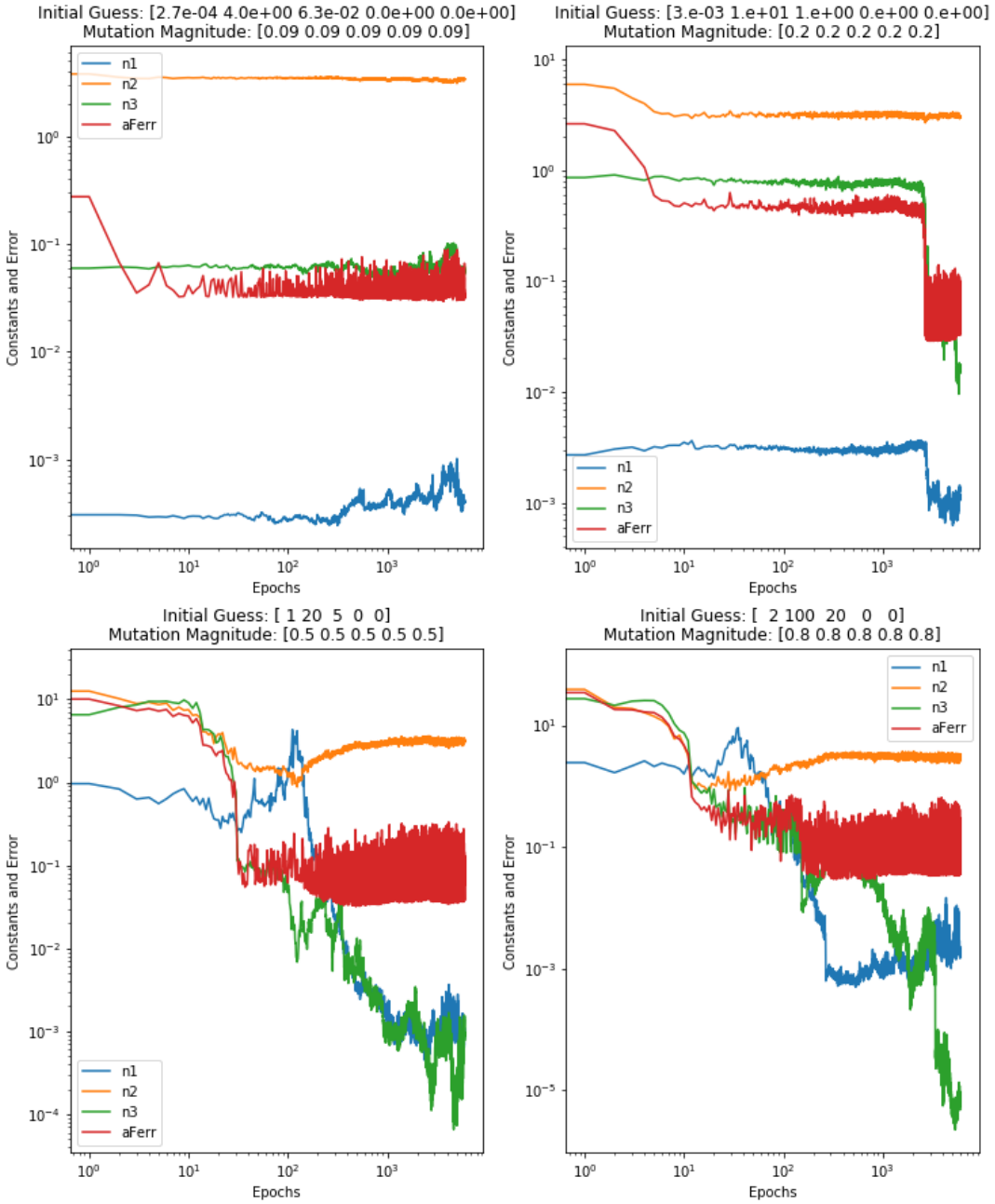
g	Lin. Regression	Exp. Regression
0.098	y = 7.167503 * x + -191.125966 MSE = 15.158677	y = 0.006760 * x ^(2.581364) MSE = 72.352377
9.8	y = 6.785368 * x + -173.429351 MSE = 38.131234	y = 0.005384 * x ^(2.641377) MSE = 32.514733

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.

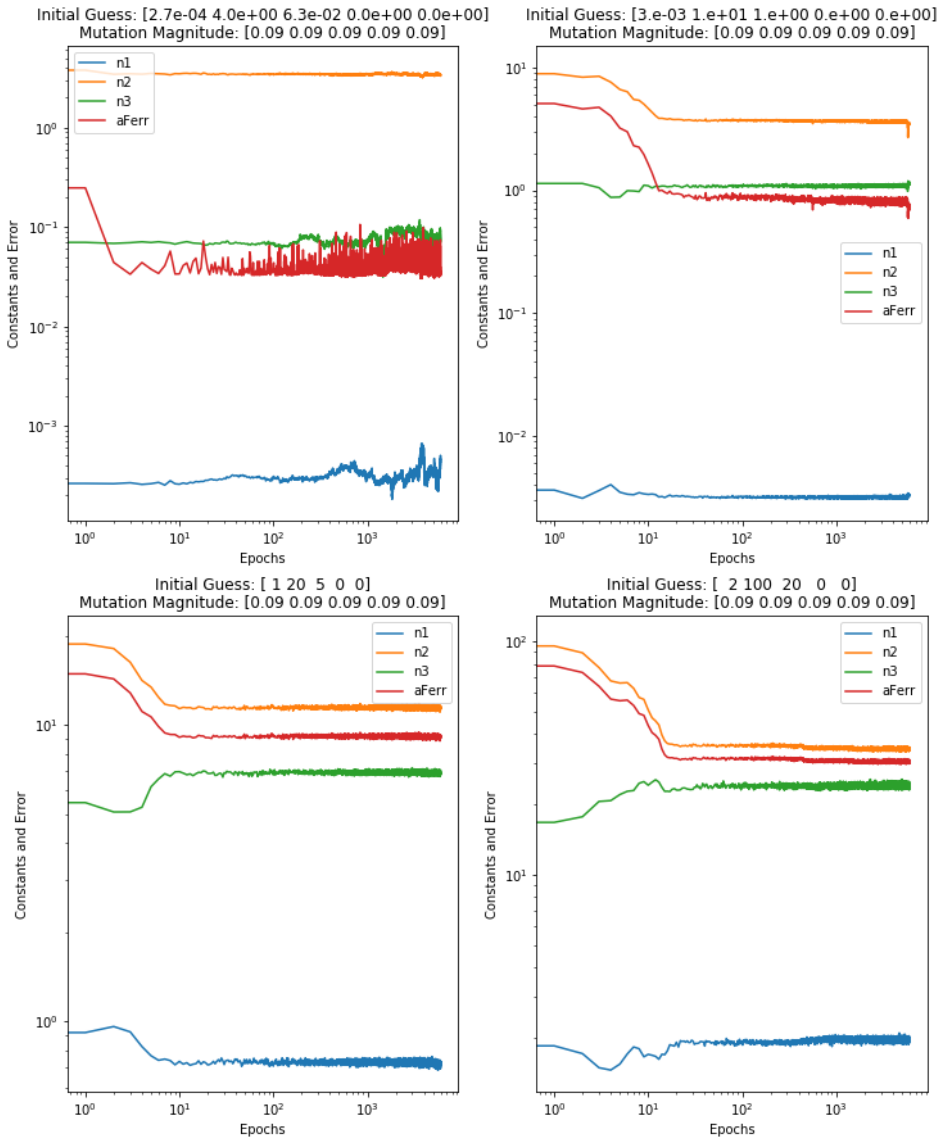
As this problem can also be considered as a linear regression problem, therefore we could create a baseline with linear regression algorithm (See Task2_LR.py). Listed below is the result of linear regression.

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	

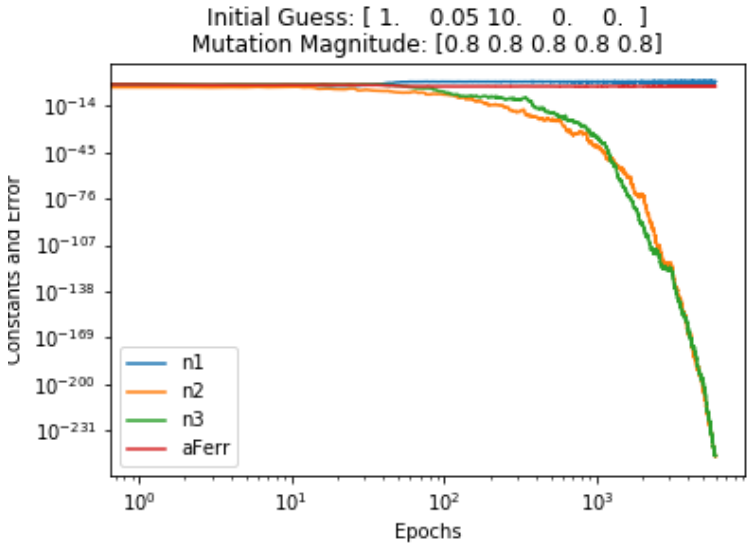
Experiment	Results	
	Relative Absolute Error	Best Epoch
1	0.02994	3161
2	0.5907	5825
3	8.830	5902
4	29.625	4881

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

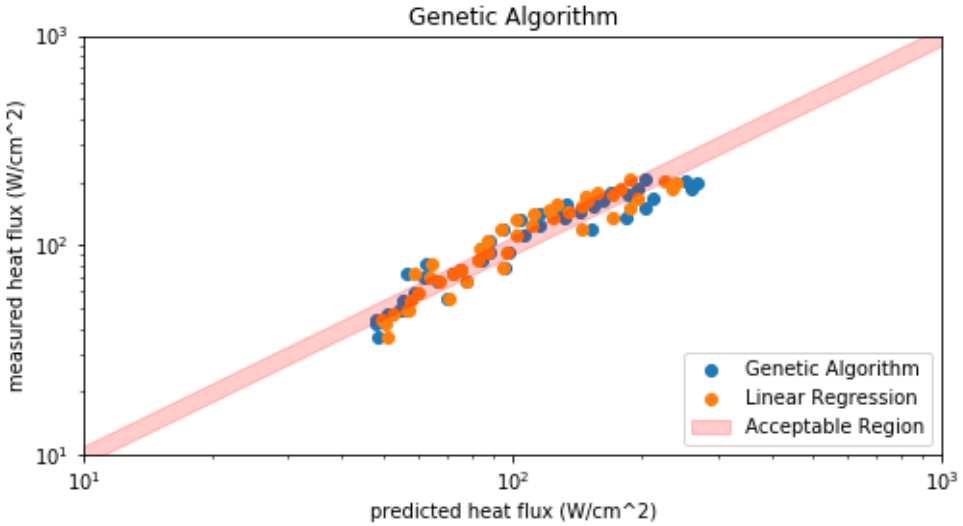


As shown in the graph, n2 and n3 rapidly slopes to zero. However, it's not the single case. Similar trends could also be observed in experiment 2, 3, 4 (the n3/green line).
Here is my explanation to this phenomenon: genes naturally have a higher probability to become less. Each gene has a uniform chance 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 acceptable region where absolute error is within 10%. An unacceptable rate is defined to describe the proportion of points outside the acceptable region. The statistical results are listed below.

Methodology	RMS Error	RMS Deviation	Unacceptable Rate
Genetic Algorithm	0.02679	0.1752	53.33%
Linear Regression	0.02433	0.1627	62.22%

For both of the 2 methodologies, deviation is larger than the uncertainty. It may not be a good fit for this case because of the high RMS deviation and unacceptable rate as well. The RMD deviation is above the noise in the data. Therefore, this model is not accurate enough to predict the trend for this problem. To further improve the accuracy of the prediction, we need to find some different models and maybe more parameters.

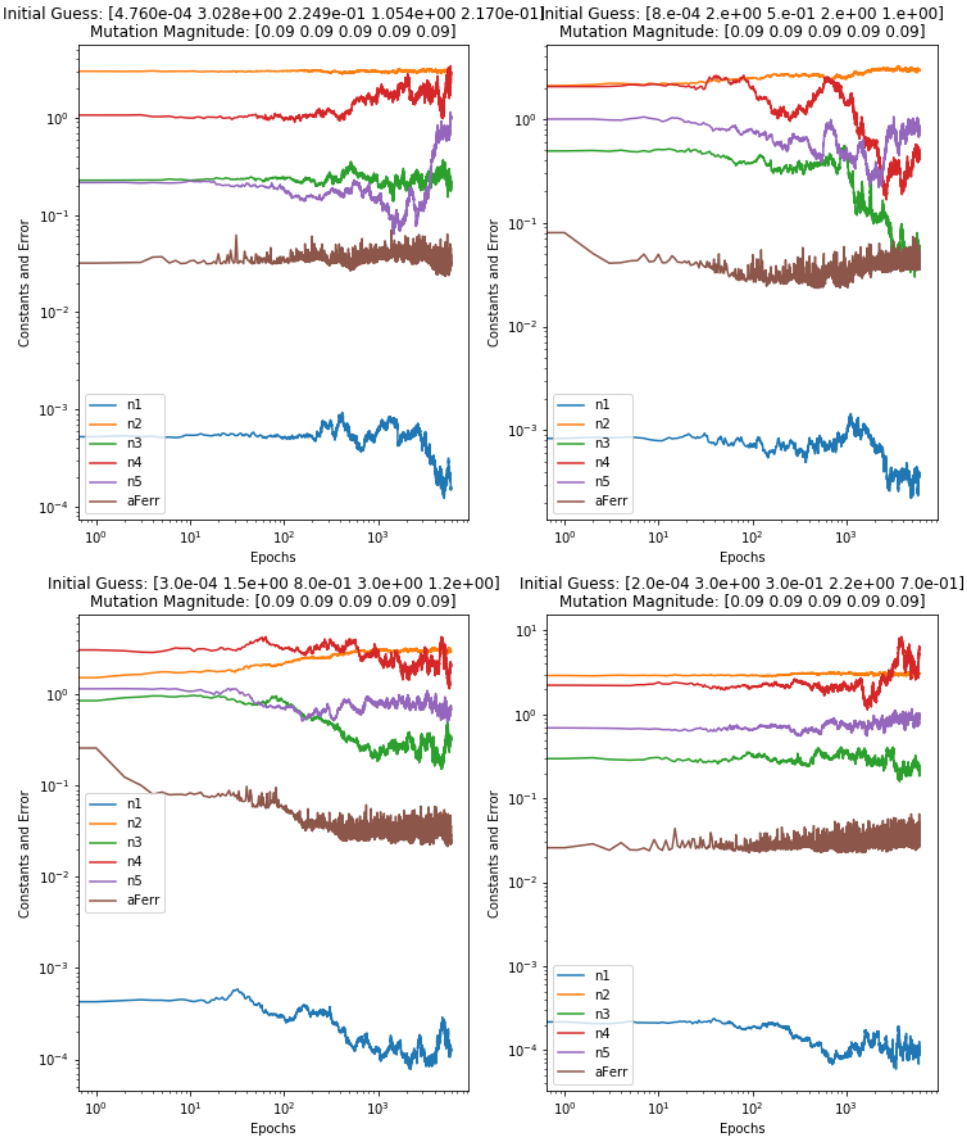
Code Modified:

- 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.

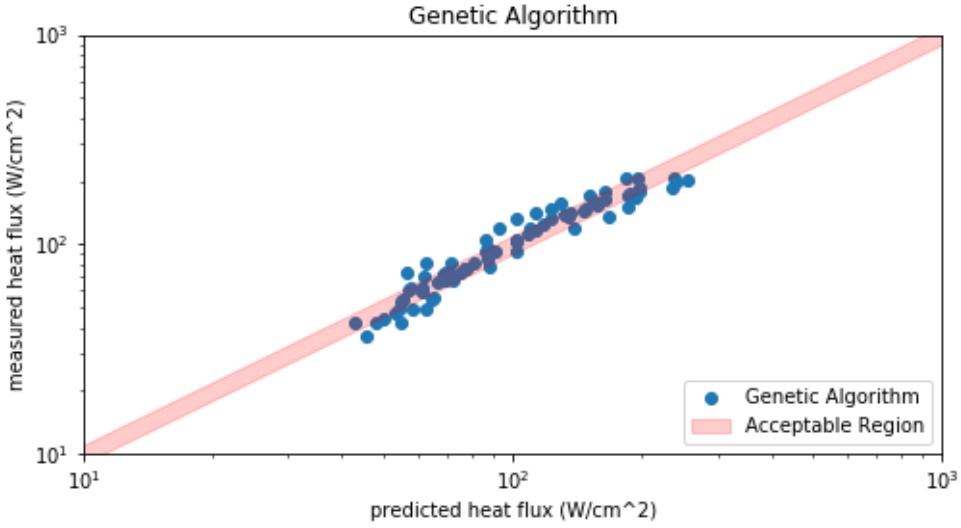
Experiment	Parameters					
	n1	n2	n3	n4	n5	Mutation Magnitude
1 (Original)	4.76e-4	3.028	0.2249	1.054	0.217	9%
1 (Optimal)	1.546e-4	2.976	0.3454	1.798	0.6690	
2 (Original)	8e-4	2	0.5	2	1	9%
2 (Optimal)	6.619e-4	2.675	0.3712	1.716	0.4266	
3 (Original)	3e-4	1.5	0.8	3	1.2	9%
3 (Optimal)	1.669e-4	2.946	0.4143	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 and other statistical features can also be calculated in the same way as Task 2. Results are shown below.



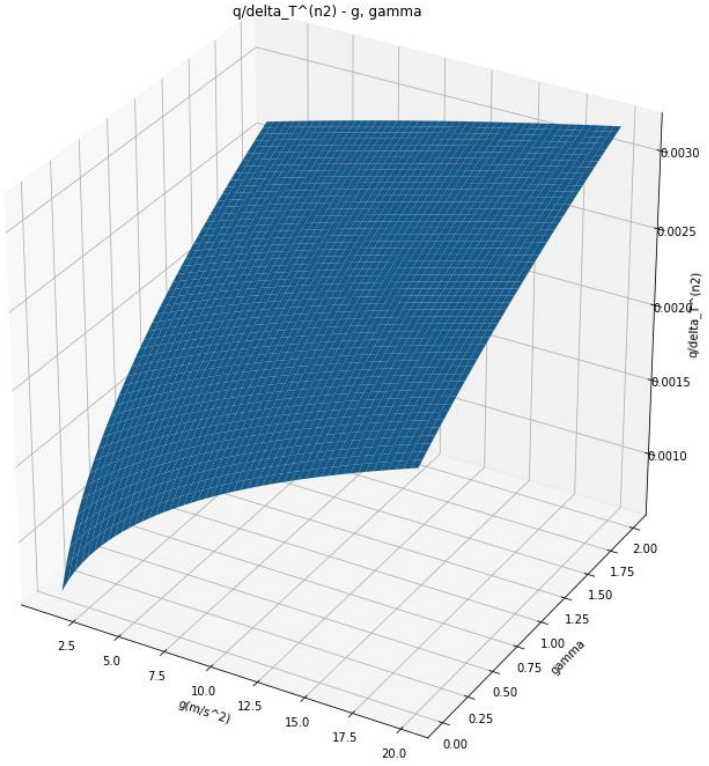
Methodology	RMS Error	RMS Deviation	Unacceptable Rate
Genetic Algorithm	0.01484	0.1289	42.86%

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. However, the RMS deviation of 12.89% is still higher than measurement uncertainty, while the unacceptable rate is still as high as 42.86%. There's still way to improve the model.

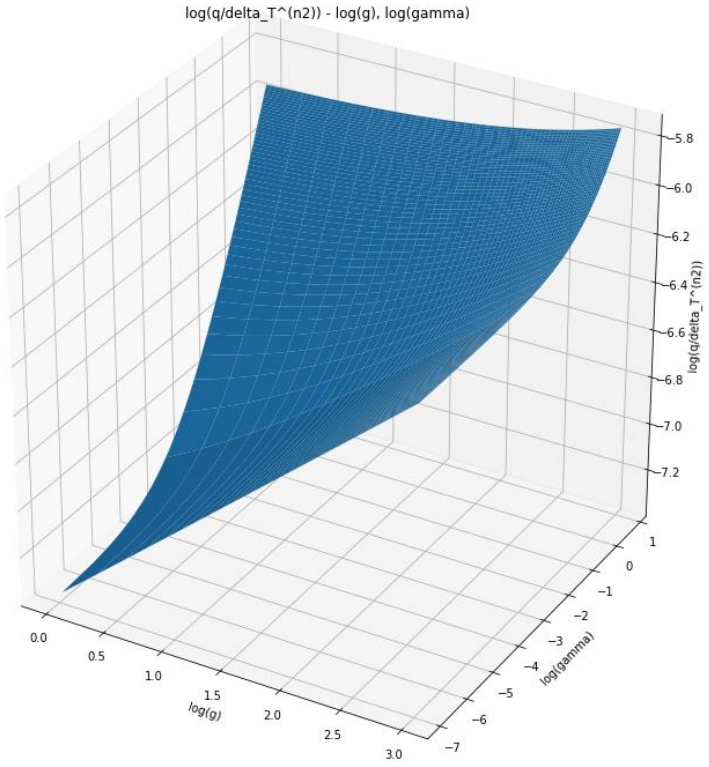
If we use the optimal parameters obtained from genetic algorithm, we can get the followed relationship:

$$\frac{q''}{(T_w - T_{sat})^{n_2}} = n_1(g + n_4 g_{en}\gamma)^{n_3} P^{n_5}$$

With the condition of $\mathrm{P} = 10\mathrm{kPa}$, we can create a surface plot of $\frac{q''}{(T_w - T_{sat})^{n_2}}$ versus g and γ 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 n_3 and n_4 have been exchanged to align with the former problem):

$$Q_s = n_1 {Ja_s}^{n_2} (\frac{g}{g_{en}} + n_4\gamma)^{n_3} {Pr_l}^{-n_5}$$

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

$$\ln(Q_s) = \ln(n_1) + n_2 \ln(Ja_s) + n_3 \ln\left(\frac{g}{g_{en}} + n_4\gamma\right) - n_5 \ln(Pr_l)$$

Therefore, $f_{err,i}$ should be:

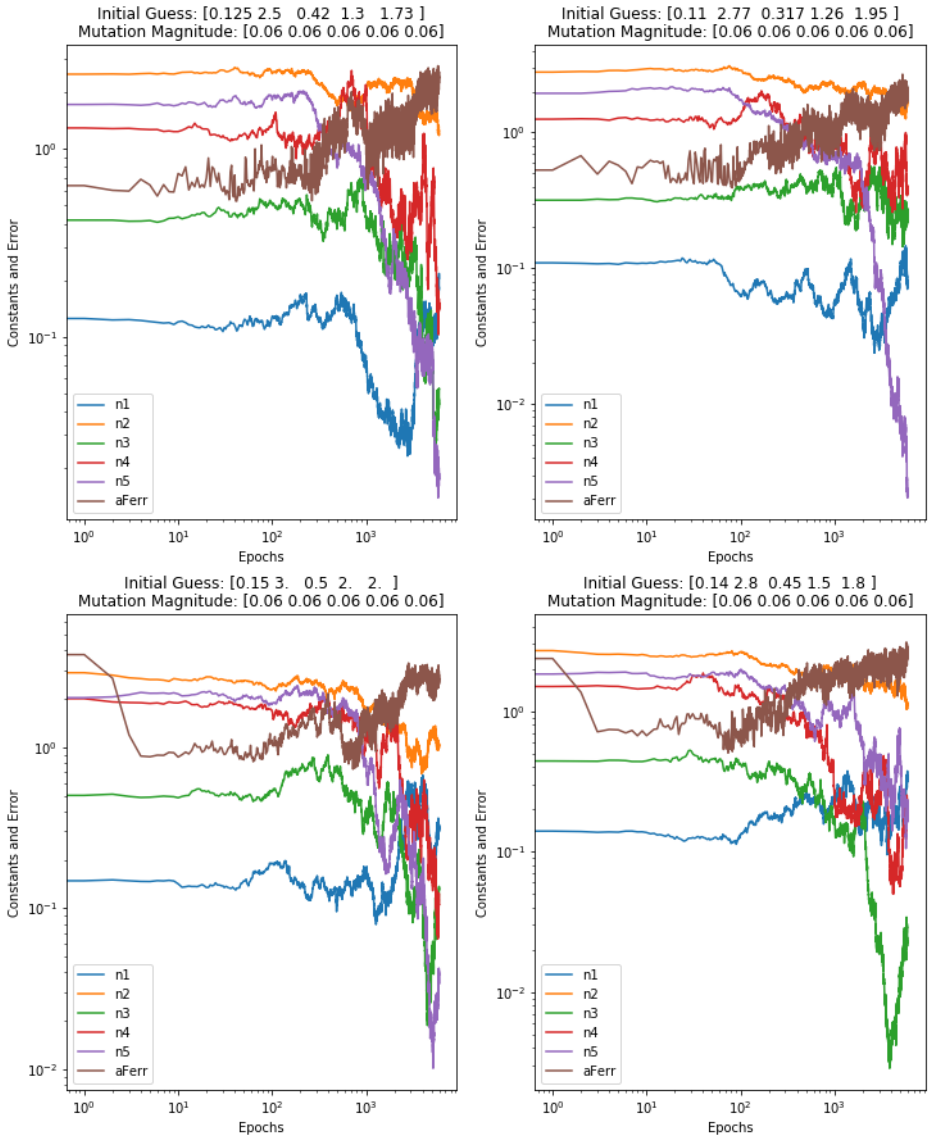
$$f_{err,i} = -\ln(Q_{s,data}) + \ln(n_1) + n_2 \ln(Ja_s) + n_3 \ln\left(\frac{g}{g_{en}} + n_4\gamma\right) - n_5 \ln(Pr_l)$$

And F_{err} should be the sum of all relative absolute $f_{err,i}$.

$$F_{err} = \sum_{i=1}^{N_D} \left| \frac{-\ln(Q_{s,data}) + \ln(n_1) + n_2 \ln(Ja_s) + n_3 \ln\left(\frac{g}{g_{en}} + n_4\gamma\right) - n_5 \ln(Pr_l)}{\ln(Q_{s,data})} \right|$$

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 Q_s 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 Q_s 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. Some of the detailed results are

listed below.

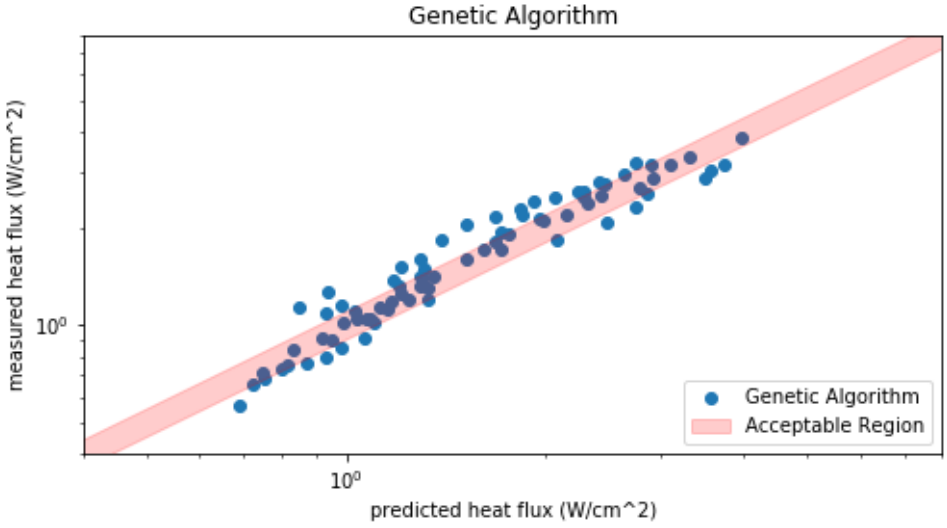
Experiment	Parameters					
	n1	n2	n3	n4	n5	Mutation Magnitude
1 (Original)	0.125	2.5	0.42	1.3	1.73	6%
1 (Optimal)	0.1074	2.6720	0.4437	1.1222	1.8679	
2 (Original)	0.11	2.77	0.317	1.26	1.95	6%
2 (Optimal)	0.0751	2.9145	0.3439	1.2323	1.9050	
3 (Original)	0.15	3	0.5	2	2	6%
3 (Optimal)	0.1290	2.3500	0.4783	1.2677	1.6153	
4 (Original)	0.14	2.8	0.45	1.5	1.8	6%
4 (Optimal)	0.1213	2.5997	0.3769	1.3261	1.8462	

Experiment	Results	
	Relative Absolute Error	Best Epoch
1	0.5271	42
2	0.3910	82
3	0.7301	727
4	0.5045	74

All of the solutions has a relative absolute log error around 0.5, which is large compared with those in the previous tasks. However, it is due to little target (Q_s). In fact, the model is performing as well as the previous one, which will be shown later. Solution from experiment 2 seems to minimize the error, and therefore is chosen as the optimal model for followed problems.

	n1	n2	n3	n4	n5
Chosen Parameters	0.0751	2.9145	0.3439	1.2323	1.9050

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



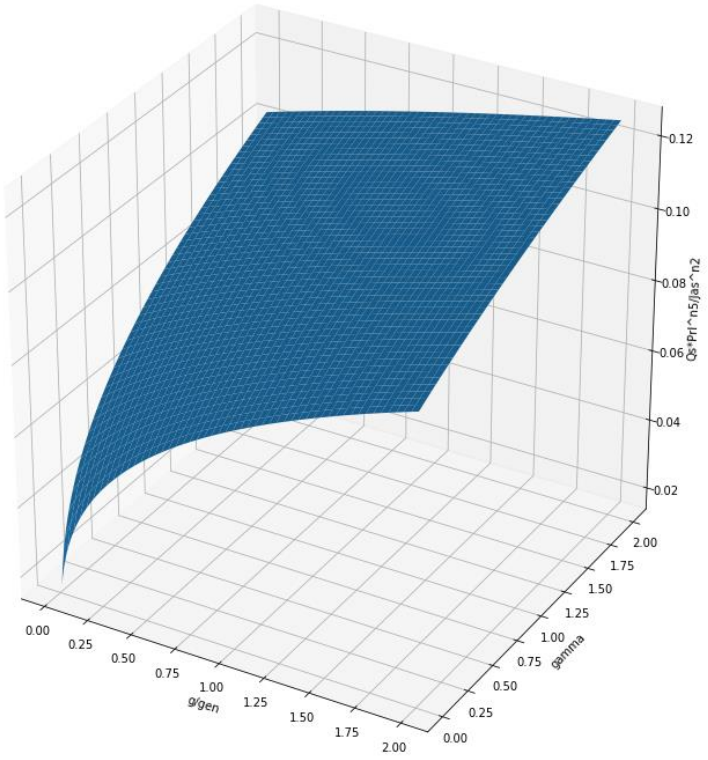
Methodology	RMS Error	RMS Deviation	Unacceptable Rate
Genetic Algorithm	0.01441	0.1234	44.16%

RMS error and RMS deviation are slightly less than the result of task 3. The unacceptable rate raised by a little, which I explain as the accumulation of error during the calculation process of non-dimensional parameters. Therefore, the non-dimensional, 5-parameter model is also performing well in this problem.

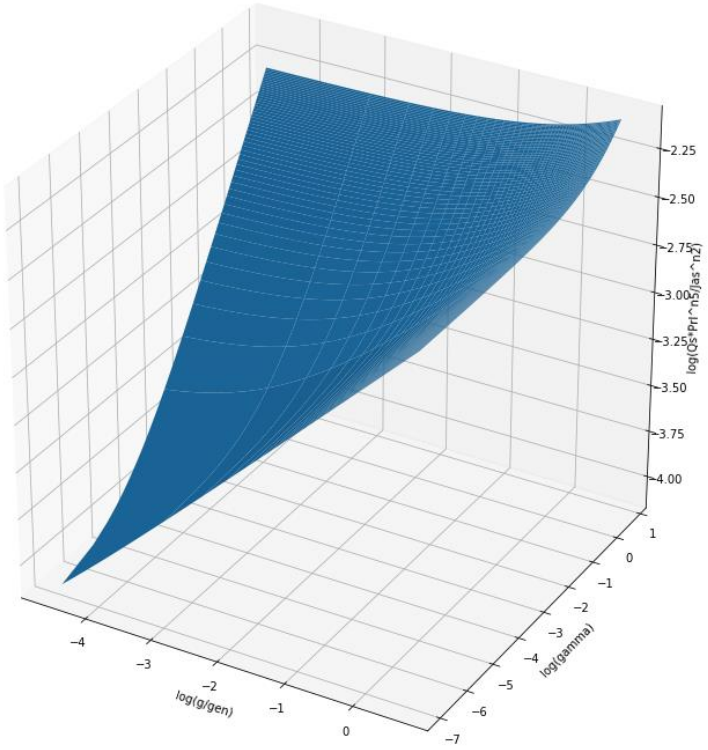
From the equation, we can obtain that:

$$\frac{Q_s {Pr_l}^{n_5}}{{Ja_s}^{n_2}} = n_1\left(\frac{g}{g_{en}} + n_4\gamma\right)^{n_3}$$

The surface plot follows like:



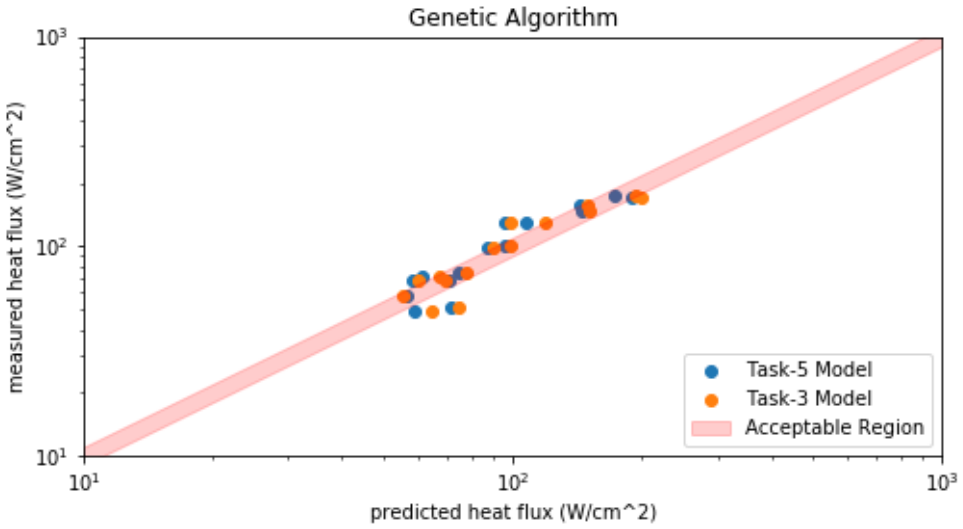
Also, we can manually transform the coordinates into log form.



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	Unacceptable Rate
Dimensionless	0.04148	0.1594	53.33%
Raw data	0.04445	0.1699	40.00%

Both of the models are performing well on the validation dataset, and dimensionless model has a slightly better RMS error and RMS deviation, while its unacceptable rate is a little bit higher.

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 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. 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 (dimensionless data)

Qs	Jas	Prl	g/gen	SurfaceTension
0.682902	5.61828	4.83	0.01	1.79
0.734003	5.739289	4.83	0.01	1.79
0.764974	5.912159	4.83	0.01	1.79
0.91673	6.015881	4.83	0.01	1.79
1.049903	6.275186	4.83	0.01	1.79
1.139718	6.448056	4.83	0.01	1.79
1.181528	6.534491	4.83	0.01	1.79
1.320896	6.77651	4.83	0.01	1.79
1.494331	6.793797	4.83	0.01	1.79
1.718868	7.312407	4.83	0.01	1.79
1.920177	7.519851	4.83	0.01	1.79
2.109098	7.848304	4.83	0.01	1.79
2.22214	8.073036	4.83	0.01	1.79
2.394027	8.28048	4.83	0.01	1.79
2.525652	8.401489	4.83	0.01	1.79
2.67586	8.79909	4.83	0.01	1.79
2.852392	8.937386	4.83	0.01	1.79
3.154355	9.749876	4.83	0.01	1.79
0.56831	5.220678	4.83	1	1.79
0.85324	5.894872	4.83	1	1.79
1.045258	6.102316	4.83	1	1.79
1.207853	6.534491	4.83	1	1.79
1.424648	6.586352	4.83	1	1.79
1.858236	7.623573	4.83	1	1.79
2.079676	8.10761	4.83	1	1.79
2.32744	8.384202	4.83	1	1.79
2.586045	8.505211	4.83	1	1.79
2.849295	9.110256	4.83	1	1.79
3.042861	9.179404	4.83	1	1.79
0.791405	4.888889	3.91	2	1.79
0.908996	5.115873	3.91	2	1.79
1.017254	5.168254	3.91	2	1.79
1.15911	4.97619	3.91	2	1.79
1.321498	5.325397	3.91	2	1.79
1.375627	5.290476	3.91	2	1.79
1.526815	5.342857	3.91	2	1.79
1.715334	6.02381	3.91	2	1.79
1.939316	6.02381	3.91	2	1.79
2.223028	6.180952	3.91	2	1.79
2.49554	6.425397	3.91	2	1.79
2.611264	6.652381	3.91	2	1.79

2.768052	6.826984	3.91	2	1.79
2.930439	6.984127	3.91	2	1.79
3.156289	7.368254	3.91	2	1.79
3.344807	7.542857	3.91	2	1.79
3.82637	8.031746	3.91	2	1.79
0.656577	5.134243	4.83	2	1.79
0.754134	5.358974	4.83	2	1.79
0.843949	5.393548	4.83	2	1.79
1.096359	5.600993	4.83	2	1.79
1.141267	5.428122	4.83	2	1.79
1.266697	5.61828	4.83	2	1.79
1.423099	6.275186	4.83	2	1.79
1.608923	6.275186	4.83	2	1.79
1.844299	6.430769	4.83	2	1.79
2.070385	6.638213	4.83	2	1.79
2.166393	6.862945	4.83	2	1.79
2.29647	7.070389	4.83	2	1.79
2.431192	7.191398	4.83	2	1.79
2.618564	7.588999	4.83	2	1.79
2.774966	7.779156	4.83	2	1.79
3.174486	8.28048	4.83	2	1.79
1.306946	7.209892	4.54	1	0
1.205106	7.03616	4.54	1	0
1.12024	6.862427	4.54	1	0
1.052346	6.688695	4.54	1	0
0.71288	5.906899	4.54	1	0
1.0184	6.514963	4.54	1	0
0.899586	6.428096	4.54	1	0
1.110296	6.292473	4.83	0.01	1.71
1.262052	6.6555	4.83	0.01	1.71
1.404517	6.828371	4.83	0.01	1.71
1.599631	7.191398	4.83	0.01	1.71
1.81178	7.450703	4.83	0.01	1.71
2.146262	7.848304	4.83	0.01	1.71
2.503973	8.28048	4.83	0.01	1.71
3.2132	8.79909	4.83	0.01	1.71

Appendix 4 – 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 5 – 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 = 1, 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)
        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



# 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 6 – 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 7 – 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, Unacceptable Rate: %f"%(mean, dev,
unacc_rate))
        return mean, dev, unacc_rate



# 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 8 – 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"] / ydata["ul"] / ydata["hlv"] *
np.sqrt(ydata["sigma"]/gen/(ydata["rhol"]-ydata["rhov"]))
ydata["Jas"] = 100 * ydata["cpl"] * ydata["Superheat"] / ydata["hlv"]
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 9 – 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 = 0.5
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/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, Unacceptable Rate: %f"%(mean, dev,
unacc_rate))
        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[1]
# 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(0.1, 10, 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 = 8, xmin = 0.4)
plt.ylim(ymax = 8, ymin = 0.4)
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, Unacceptable Rate: %f"%(mean, dev,
unacc_rate))
    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, Unacceptable Rate: %f"%(mean, dev,
unacc_rate))
```

```
    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)
```

Appendix 10 – 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 11 – 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 12 – 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"]))
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)
```