Machine Learning Tools for Modeling Energy Transport and Conversion Processes
University of California, Berkeley
MEC ENG 249, Fall 2023
Instructor: Prof. Van P. Carey

Authors: Ali Rakhsha, Daniel Garcia Mijares

Project I: Predictive Genetic Algorithm Exploration

Abstract.

This report explores the application of predictive genetic algorithms in understanding complex phenomena and optimizing engineering systems. Two distinct sets of data are analyzed: the first pertains to heat flux in nucleate boiling processes, investigating the influence of factors like gravitational acceleration, wall superheat, pressure, and surface tension. The second set focuses on the design and optimization of a heat pipe heat exchanger (HPHE) for electronic component cooling. The analysis is structured into tasks, including data exploration, machine learning analysis with genetic algorithms, and model expansion. Results show that gravity has a notable impact on heat flux, demonstrated through statistical analysis. Genetic algorithms are employed to optimize model parameters, emphasizing the importance of initial guesses and their effect on model performance. Furthermore, the report discusses the advantages and disadvantages of raw data analysis versus dimensionless data analysis, highlighting their respective strengths and weaknesses.

Through the used of predictive genetic model equations that describe heat flux in nucleate boiling processes was produce. The best initial guess found produced a minimum absolute error (MAE) of 0.0288, and a root mean square error (RMSE) of 18.37. Similarly, after adding complexity to the model the best solution found had MAE and RMSE of 13.79 and 0.0237, respectively. Finally, for HPHE for electronic component cooling set of data was divided into training and validation. After optimizing the algorithm, the results obtain were as follow. Training set RMSE and MAE, 1.14 and 0.0164, respectively. Validation set RMSE and MAE, 1.23 and 0.0216, respectively.

Introduction.

The field of engineering and scientific research often involves the exploration of complex phenomena and the development of predictive models to understand the underlying relationships between various parameters. In this report, we delve into the realm of predictive genetic algorithms to explain the intricate dependencies within multivariate data related to two independent data sets. First set is in relation to heat flux in nucleate boiling processes. Our investigation aims to explore on how heat flux varies concerning key factors such as gravitational acceleration, wall superheat, pressure, and surface tension parameters. This section is structured around three distinct tasks: (1) initial setup and data exploration, (2) machine learning analysis with genetic algorithm, and (3) expanding the model and surface plot creation. The second set centers around the design and optimization of a heat pipe heat exchanger (HPHE) used for cooling electronic components within a cabinet. While a detailed physics-based model can be constructed to account for various factors such as wick flow, heat transport, and fluid properties, a simplified model can often suffice for well-designed heat pipes. This section is structure in two continuous tasks: (1) genetic algorithm implementation applying the separation

of data for training and validation, and (2) program testing and parameter optimization.

Results.

Task 1.1

Initial setup and data exploration. From figures 1-4 show a representation of the data at constant Pressure 5.5 MPa and surface tension parameter 1.79.

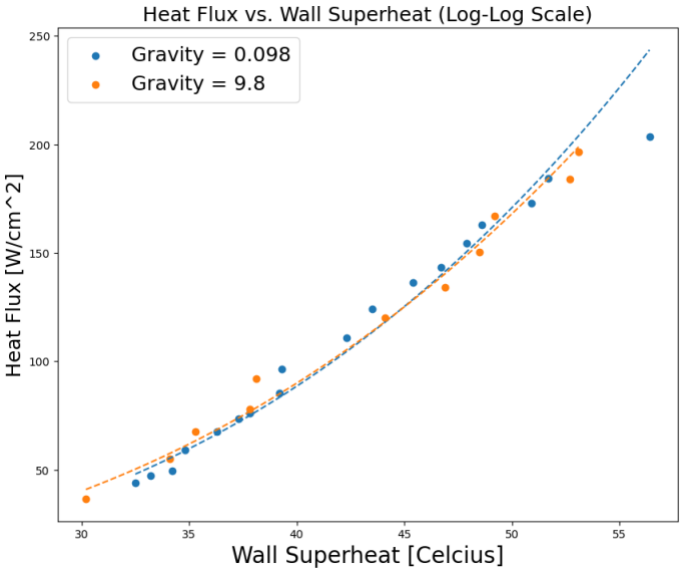


Figure 1. Log-log plot of Heat Flux vs. Wall Superheat for two levels of gravity (0.098 and 9.8 m/s^2)

OLS Regression Results

Dep. Variable:	y	R-squared:	0.983
Model:	OLS	Adj. R-squared:	0.981
Method:	Least Squares	F-statistic:	731.3
Date:	Tue, 26 Sep 2023	Prob (F-statistic):	1.41e-23
Time:	22:20:49	Log-Likelihood:	-96.287
No. Observations:	29	AIC:	198.6
Df Residuals:	26	BIC:	202.7
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	-981.8683	28.722	-34.185	0.000	-1040.907	-922.829
x1	293.4817	7.686	38.184	0.000	277.683	309.281
x2	0.5305	0.588	0.902	0.375	-0.678	1.739

Omnibus:	0.897	Durbin-Watson:	1.294
Prob(Omnibus):	0.639	Jarque-Bera (JB):	0.253
Skew:	0.202	Prob(JB):	0.881
Kurtosis:	3.215	Cond. No.	89.0

Figure 2. Statistical analysis results using Python statsmodel library for set of data corresponding to earth and micro gravity.

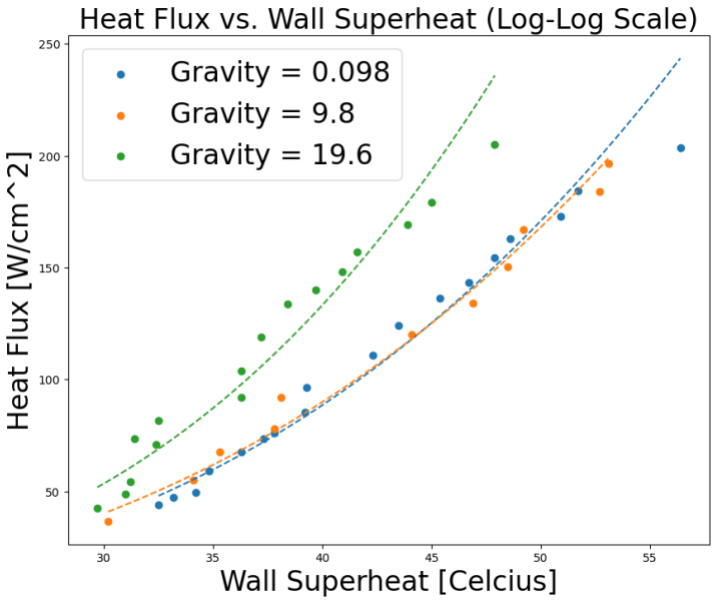


Figure 3. Log-log plot of Heat Flux vs. Wall Superheat for two levels of gravity (0.098, 9.8, 19.6 m/s^2)

OLS Regression Results

Dep. Variable:	y	R-squared:	0.917
Model:	OLS	Adj. R-squared:	0.913
Method:	Least Squares	F-statistic:	232.0
Date:	Fri, 29 Sep 2023	Prob (F-statistic):	2.00e-23
Time:	09:27:36	Log-Likelihood:	-183.82
No. Observations:	45	AIC:	373.6
Df Residuals:	42	BIC:	379.1
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	-935.1428	48.736	-19.188	0.000	-1033.496	-836.789
x1	283.3898	13.167	21.523	0.000	256.819	309.961
x2	5.1245	0.921	5.567	0.000	3.267	6.982

Omnibus:	1.446	Durbin-Watson:	0.283
Prob(Omnibus):	0.485	Jarque-Bera (JB):	1.386
Skew:	-0.396	Prob(JB):	0.500
Kurtosis:	2.665	Cond. No.	89.1

Figure 4. Statistical analysis results using Python statsmodel library for set of data corresponding to earth, micro, and 2X gravity.

Task 1.2

Table 1. Results for set of initial guesses at 9% perturbation.

Set	n1i	n2i	n3i	n1min	n2min	n3min	p	NGEN	RMSE	MAE
1	0.00027	4.000	0.063	0.00102	3.12	0.064	0.09	6000	28.68	0.0296
2	0.02091	2.125	0.334	0.02053	2.22	0.260	0.09	6000	53.74	0.1129
3	0.10500	6.200	0.163	0.10443	6.11	0.162	0.09	6000	1.73E+09	3.4422
4	6.00000	5.000	0.500	6.05890	4.91	0.503	0.09	6000	1.99E+09	3.4172

Table 2. Results for set of initial guesses at 15% perturbation.

Set	n1i	n2i	n3i	n1min	n2min	n3min	p	NGEN	RMSE	MAE
1	0.00027	4.000	0.063	0.00098	3.13	0.058	0.15	6000	27.17	0.0292
2	0.02091	2.125	0.334	0.01861	2.33	0.051	0.15	6000	18.21	0.0338
3	0.10500	6.200	0.163	0.10662	6.00	0.162	0.15	6000	1.17E+09	3.3618
4	6.00000	5.000	0.500	6.01128	4.76	0.493	0.15	6000	1.07E+09	3.2918

Table 3. Results for set of initial guesses at 30% perturbation.

Set	n1i	n2i	n3i	n1min	n2min	n3min	p	NGEN	RMSE	MAE
1	0.00027	4.000	0.063	0.00146	3.02	0.051	0.30	6000	24.38	0.0288
2	0.02091	2.125	0.334	0.00379	2.76	0.035	0.30	6000	19.72	0.0300
3	0.10500	6.200	0.163	0.01105	2.46	0.019	0.30	6000	20.12	0.0326
4	6.00000	5.000	0.500	0.74057	1.33	0.043	0.30	6000	29.83	0.0552

The third set from the previous tables will be used to observe the influence that the number of generations have in the obtained results from the predicted genetic algorithm. Specifically, as it relates to RMSE and MAE.

Table 4. Influence of NGEN for an initial guess.

Set	n1i	n2i	n3i	n1min	n2min	n3min	p	NGEN	RMSE	MAE
1	0.105	6.2	0.163	0.10626	5.80	0.169	0.26	3000	5.20E+08	3.1964
2	0.105	6.2	0.163	0.10490	5.75	0.163	0.26	6000	4.10E+08	3.1497
3	0.105	6.2	0.163	0.00208	2.92	0.034	0.26	12000	21.31	0.0291
4	0.105	6.2	0.163	0.02222	2.28	0.027	0.26	24000	19.16	0.0343

An extra code was generate using the provided code with the only goal to search through a linear space of each of the variables (n1, n2, n3) to obtain the lowest values of RMSE and MAE. 15,625 combinations were tested and only the best consistent initial value for the lowest RMSE was extracted. Table 5 shows the specified value at a range of perturbations. The predictive equation is as follow:

$$q" = n_1(T_w - T_{sat})^{n_2} g^{n_3}$$

Table 5. Results for best initial guess at a range of perturbations

Set	n1i	n2i	n3i	n1min	n2min	n3min	p	NGEN	RMSE	MAE
1	0.00735	2.64	0.045	0.00209	2.92	0.05	0.90	6000	22.29	0.0289
2	0.00735	2.64	0.045	0.00242	2.87	0.035	0.95	6000	20.64	0.0292
3	0.00735	2.64	0.045	0.00152	3	0.035	0.80	6000	22.84	0.0292
4	0.00735	2.64	0.045	0.0032	2.8	0.024	0.85	6000	20.52	0.0306
5	0.00735	2.64	0.045	0.00215	2.91	0.045	0.70	6000	21.86	0.0288
6	0.00735	2.64	0.045	0.00198	2.93	0.04	0.75	6000	21.7	0.0289
7	0.00735	2.64	0.045	0.00132	3.04	0.034	0.60	6000	23.27	0.0293
8	0.00735	2.64	0.045	0.00304	2.81	0.036	0.55	6000	20.03	0.0292
9	0.00735	2.64	0.045	0.00604	2.63	0.052	0.09	6000	18.99	0.0307
10	0.00735	2.64	0.045	0.00514	2.68	0.04	0.09	6000	18.96	0.0304
11	0.00735	2.64	0.045	0.00507	2.68	0.042	0.09	6000	18.88	0.0302
12	0.00735	2.64	0.045	0.00683	2.6	0.054	0.09	6000	18.95	0.031
13	0.00735	2.64	0.045	0.00462	2.7	0.043	0.07	6000	19.03	0.0299
14	0.00735	2.64	0.045	0.0064	2.62	0.049	0.07	6000	18.64	0.0308
15	0.00735	2.64	0.045	0.00554	2.65	0.044	0.07	6000	18.69	0.0304
16	0.00735	2.64	0.045	0.00568	2.65	0.047	0.07	6000	18.7	0.0305
17	0.00735	2.64	0.045	0.00674	2.59	0.039	0.05	6000	18.67	0.0308
18	0.00735	2.64	0.045	0.00619	2.63	0.054	0.05	6000	19.11	0.0308
19	0.00735	2.64	0.045	0.00447	2.71	0.046	0.05	6000	19.24	0.0299
20	0.00735	2.64	0.045	0.00644	2.61	0.046	0.05	6000	18.5	0.0309
21	0.00735	2.64	0.045	0.00708	2.58	0.045	0.03	6000	18.37	0.031
22	0.00735	2.64	0.045	0.00686	2.59	0.043	0.03	6000	18.44	0.0309
23	0.00735	2.64	0.045	0.00656	2.61	0.047	0.03	6000	18.53	0.0309
24	0.00735	2.64	0.045	0.00625	2.62	0.045	0.03	6000	18.51	0.0307
25	0.00735	2.64	0.045	0.00665	2.61	0.052	0.01	6000	19.13	0.031
26	0.00735	2.64	0.045	0.00735	2.58	0.049	0.01	6000	18.39	0.0311
27	0.00735	2.64	0.045	0.00724	2.58	0.04	0.01	6000	18.57	0.031
28	0.00735	2.64	0.045	0.00633	2.62	0.044	0.01	6000	18.47	0.0308

Resultant equation and descriptive behavior graph for the set 5 which describes the best MAE:

$$q" = 0.00215(T_w - T_{sat})^{2.91} g^{0.045}$$

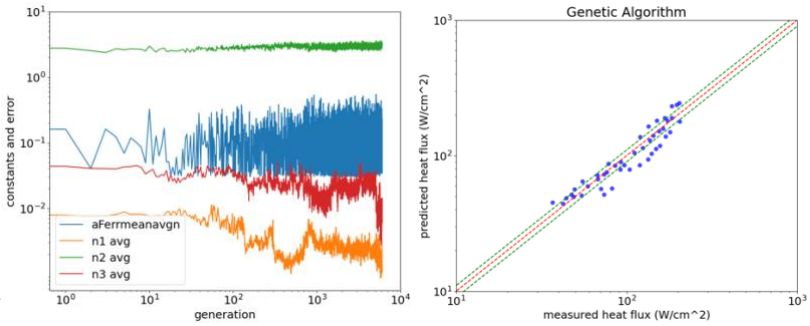


Figure 5. Resultant graphs from best initial guess to achieve lowest MAE. Average error as function of NGEN & q" predicted vs. q" data.

Resultant equation and descriptive behavior graph for the set 21 which describes the best RMSE:

$$q" = 0.00708(T_w - T_{sat})^{2.58} g^{0.045}$$

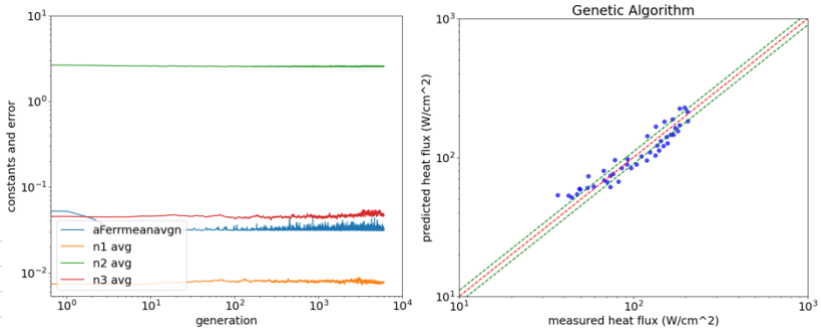


Figure 6. Resultant graphs from best initial guess to achieve lowest RMSE. Average error as function of NGEN & q" predicted vs. q" data.

Next is a comparison between the results obtained from the initial values given and the best found.

Table 6. RMSE & MAE comparison initial values given and best obtain.

Parameters		Initial Guess			
		Given		Best found	
p	NGEN	RMSE	MAE	RMSE	MAE
0.90	6000	24.03	0.0293	22.29	0.0289
0.95	6000	23.22	0.0287	20.64	0.0292
0.80	6000	20.99	0.03	22.84	0.0292
0.85	6000	22.13	0.0292	20.52	0.0306
0.70	6000	20.7	0.0289	21.86	0.0288
0.75	6000	21.93	0.0308	21.7	0.0289
0.60	6000	27.61	0.0292	23.27	0.0293
0.55	6000	23.19	0.0291	20.03	0.0292
0.09	6000	25.44	0.0287	18.99	0.0307
0.09	6000	35.33	0.0314	18.96	0.0304
0.09	6000	25.75	0.0291	18.88	0.0302
0.09	6000	31.57	0.0302	18.95	0.031
0.07	6000	4217.68	0.7296	19.03	0.0299
0.07	6000	28.99	0.0296	18.64	0.0308
0.07	6000	29.02	0.0298	18.69	0.0304
0.07	6000	4762.99	0.7544	18.7	0.0305
0.05	6000	2505.07	0.6205	18.67	0.0308
0.05	6000	3617.34	0.6963	19.11	0.0308
0.05	6000	2881.51	0.6476	19.24	0.0299
0.05	6000	1682.2	0.5389	18.5	0.0309
0.03	6000	5072.53	0.7678	18.37	0.031
0.03	6000	3817.8	0.7066	18.44	0.0309
0.03	6000	1378.02	0.4947	18.53	0.0309
0.03	6000	1654.05	0.5304	18.51	0.0307
0.01	6000	4159.33	0.7252	19.13	0.031
0.01	6000	2105.52	0.5806	18.39	0.0311
0.01	6000	3860.5	0.7096	18.57	0.031
0.01	6000	1704.4	0.537	18.47	0.0308

Task 1.3

Similarly, as in the previous section an extra code was generate using the provided code with the only goal to search through a linear space of each of the variables (n1, n2, n3, n4, n5) to obtain the lowest values of RMSE and MAE. 16,807 combinations were tested and only the best consistent initial value for the lowest RMSE was extracted. Table 7 & 8 show the results for a test run for 243 sets the algorithm then sorts the top 5 initial guess in accordance with RMSE and MAE, respectively.

Table 7. Top five initial guesses to achieve the lowest RMSE.

Set	n1i	n2i	n3i	n1min	n2min	p	NGEN	RMSE	MAE
13	0.0001	2	0.505	2.55	0.01	0.09	3000	14.62	0.0265
22	0.0001	2	1	2.55	0.01	0.09	3000	15.16	0.0274
25	0.0001	2	1	5	0.01	0.09	3000	16.19	0.0297
11	0.0001	2	0.505	0.1	2.505	0.09	3000	17.17	0.0311
14	0.0001	2	0.505	2.55	2.505	0.09	3000	17.72	0.0323

Table 8. Top five initial guesses to achieve the lowest MAE.

Set	n1i	n2i	n3i	n1min	n2min	p	NGEN	RMSE	MAE
13	0.0001	2	0.505	2.55	0.01	0.09	3000	14.62	0.0265
19	0.0001	2	1	0.1	0.01	0.09	3000	21.42	0.0265
12	0.0001	2	0.505	0.1	5	0.09	3000	19.38	0.0271
22	0.0001	2	1	2.55	0.01	0.09	3000	15.16	0.0274
16	0.0001	2	0.505	5	0.01	0.09	3000	19.15	0.0285

The best predictive equation found is as follow:

$$q" = 0.0011(T_w - T_{sat})^{2.624}(g + n_4 g_{en}\gamma)^{0.416}P^{0.31}$$

Table 9. Parameter values and resultant RMSE & MAE.

p	NGEN	RMSE	MAE
0.09	6000	13.79	0.0237

Table 10. Best initial values and resultant minimum and average.

	n1	n2	n3	n4	n5
Initial	0.0011	2.5	0.4	1.16	0.4
Minimum	0.00093	2.624	0.416	1.269	0.31
Average	0.00109	2.632	0.341	1.387	0.323

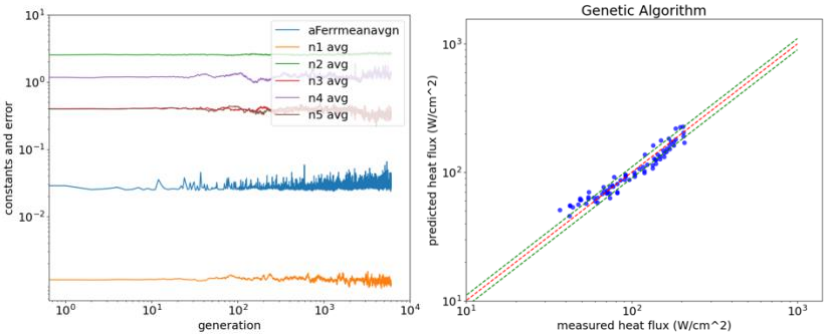


Figure 7. Resultant graphs from best initial guess to achieve for lowest RMSE. Average error as function of NGEN & q" predicted vs. q" data.

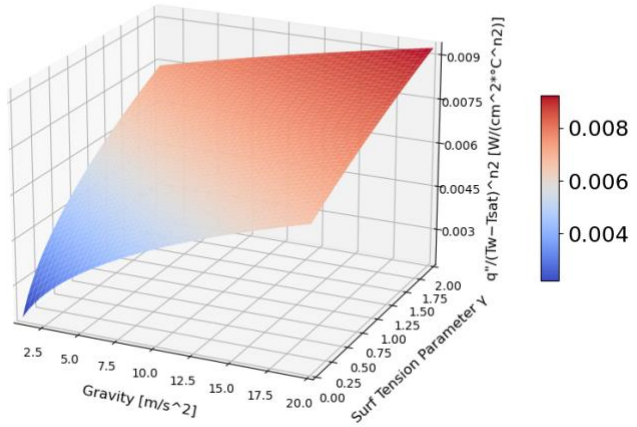


Figure 8. 3D surface plot from resulting curve-fit equation.

Task 2.2

The procedure to find the best initial values is described in the discussion section. The best resultant initial set for this last task is shown in table 11 with its respected minimum and average values. Moreover, on table 12 are the results for the set of their RMSE and MAE, for training and validation data, respectively.

The best predictive equation:

$$q = (UA)_c\left(1 + \alpha\frac{h_r}{h_{c,ref}}\right)\left[\frac{(UA)_c(T_{a,in}-T_{a,out})}{(UA)_e+(UA)_c(1+\alpha\frac{h_r}{h_{c,ref}})}\right]$$

$$where\ (UA)_e = 2.91\ W/℃$$
$$(UA)_c = 46.47\ W/℃$$
$$\alpha = 18.55$$

Table 11. Best initial values for (UA)e, (UA)c, and α.

	n1	n2	n3
Initial	3.5	50	22
Minimum	2.91	46.47	18.55
Average	2.93	44.68	28.67

Table 12. RMSE & MAE for the training and validation data.

		Training		Validation	
p	NGEN	RMSE	MAE	RMSE	MAE
0.09	6000	1.14	0.0164	1.231397	0.021686

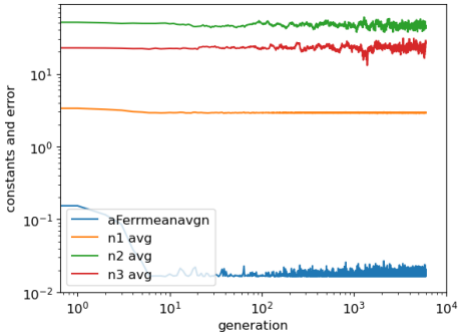


Figure 9. Function and constants (UA)e, (UA)c, and α average error per number of generations.

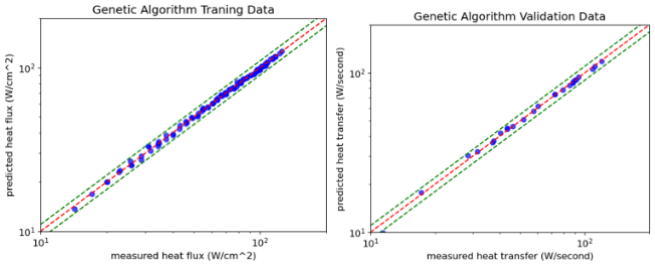


Figure 10. Training and validation data q' of predicted vs. data.

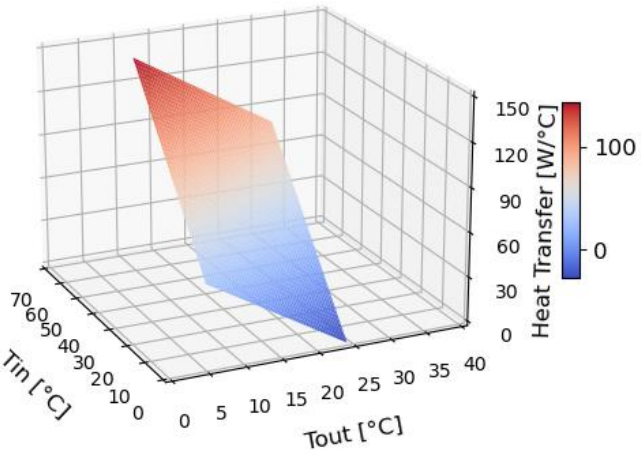
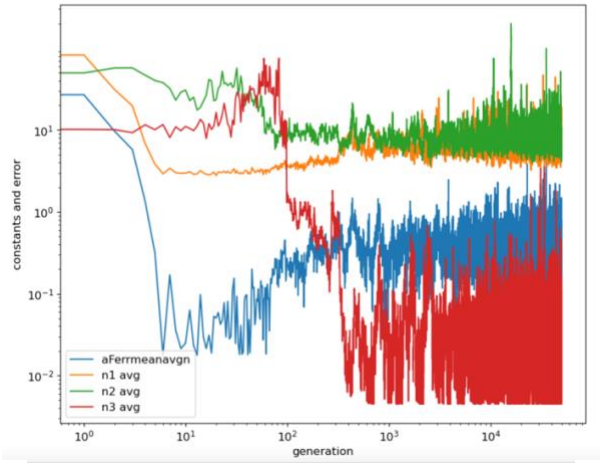


Figure 11. 3D surface plot from resulting curve-fit equation



Parameter	Value	Parameter	Value
ENDING n1	7.328	MINIMUM n1	2.916
ENDING n2	6.257	MINIMUM n2	37.85
ENDING n3	0.008634	MINIMUM n3	11.66
ENDING aFerrmean	0.2294	MINIMUM aFerrmean	0.01775

Parameter	Value	Parameter	Value
TIME AVG n1	6.622	RMS Deviation	1.191
TIME AVG n2	8.923	rms_deviation_validation	1.382
TIME AVG n3	0.05704	MAE_validation	0.01653
TIME AVG aFerrmean	0.4272		

Figure 12. Results from randomly selected values for Task 2 Step 1.

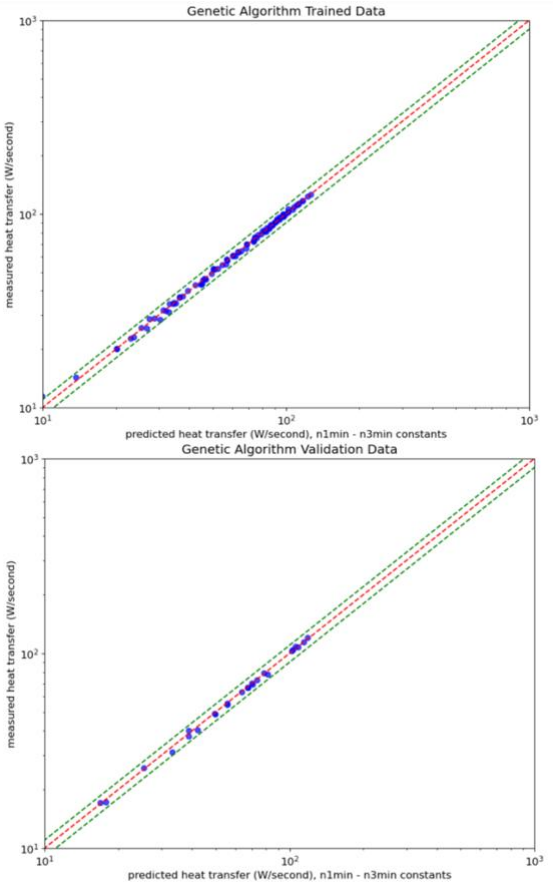
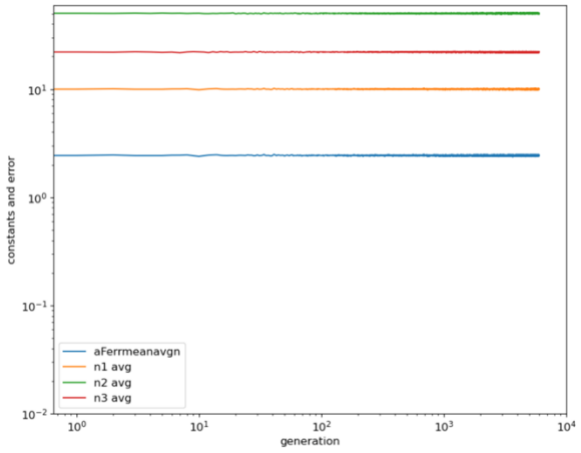


Figure 14. Predicted vs. measured data initial attempt.

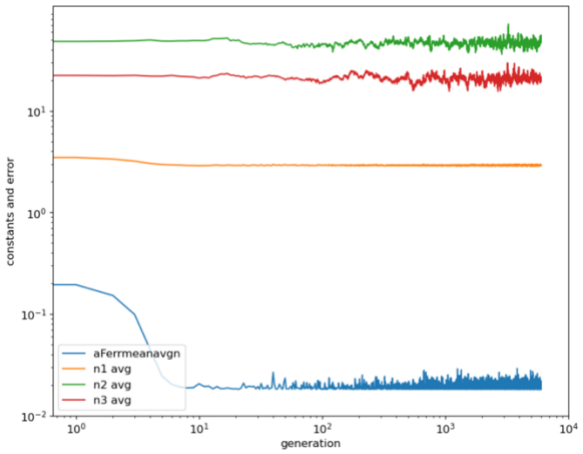


Parameter	Value	Parameter	Value
ENDING n1	9.971	MINIMUM n1	9.786
ENDING n2	49.52	MINIMUM n2	49.9
ENDING n3	22.08	MINIMUM n3	21.73
ENDING aFerrmean	2.424	MINIMUM aFerrmean	2.36

Parameter	Value
TIME AVG n1	9.998
TIME AVG n2	50.08
TIME AVG n3	22.0
TIME AVG aFerrmean	2.434

Parameter	Value
RMS Deviation	174.9
rms_deviation_validation	173.0
MAE_validation	2.369

Figure 15. Results from Step 2.



Parameter	Value	Parameter	Value
ENDING n1	2.913	MINIMUM n1	2.913
ENDING n2	53.49	MINIMUM n2	38.29
ENDING n3	21.32	MINIMUM n3	20.74
ENDING aFerrmean	0.01819	MINIMUM aFerrmean	0.01818

Parameter	Value
TIME AVG n1	2.914
TIME AVG n2	46.55
TIME AVG n3	20.91
TIME AVG aFerrmean	0.01922

Parameter	Value
RMS Deviation	1.179
rms_deviation_validation	1.503
MAE_validation	0.01467

Figure 13. Individual impact of each parameter on the output.

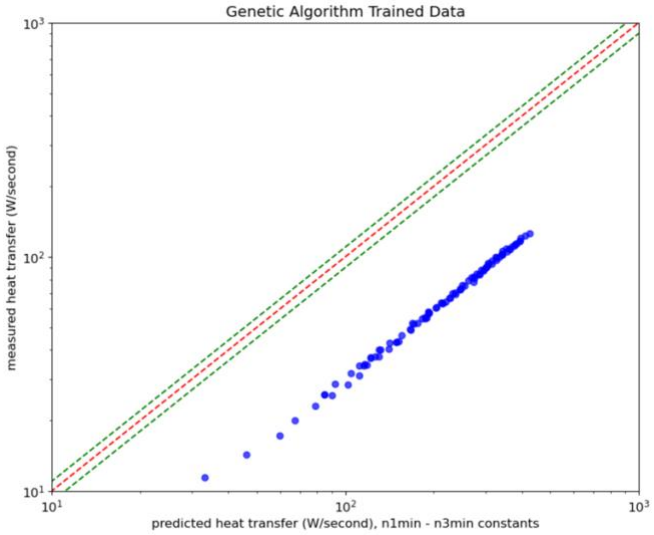
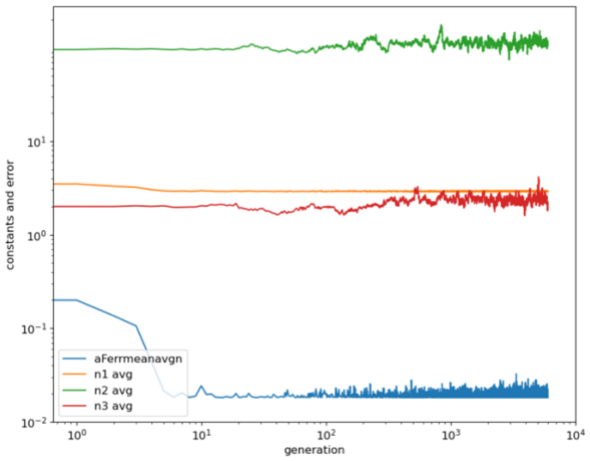


Figure 16. Assessing impact of each parameter



Parameter	Value	Parameter	Value
ENDING n1	2.92	MINIMUM n1	2.915
ENDING n2	108.3	MINIMUM n2	95.6
ENDING n3	2.029	MINIMUM n3	1.921
ENDING aFerrmean	0.01827	MINIMUM aFerrmean	0.01818

Parameter	Value	Parameter	Value
TIME AVG n1	2.917	RMS Deviation	1.173
TIME AVG n2	113.3	rms_deviation_validation	1.493
TIME AVG n3	2.396	MAE_validation	0.01467
TIME AVG aFerrmean	0.01923		

Figure 17. Assessing sensitivity

```
Results Table: 13
Set       n1i        n2i       n3i
1         100.00000  50.000    10.000
2         3.50000    50.000    22.000
3         10.00000   50.000    22.000
4         3.50000    100.000   2.000
5         3.50000    3.000     2.000

          error_min MAE_v     p         NGEN
Set
1         0.02      0.02      0.90      50000
2         0.02      0.02      0.09      6000
3         2.37      2.37      0.09      6000
4         0.02      0.02      0.09      6000
5         0.02      0.02      0.09      6000
```

Code structure modification summary.

Details can be found in the appendix section for each code pertaining its task.

Task 1.1

1. Parameters for evolution loop. Moved section after ydata array was constructed to define the length of the parameter in accordance with it.

```
# ================================================================
# Change to original code. Moved setion after the ydata array was
# constructed for the lenght of ND & NS be define as we change the
# size of the array

# Parameters for Evolution Loop
# set data parameters
ND = len(ydata)        #number of data vectors in array
DI = 5                 #number of data items in vector
NS = ND                #total number of DNA strands
# ================================================================
```

2. Plot curve-fit for each set of data on accordance with gravity.

```
# ================================================================
# Change to original code. Added section for ploting q" vs wall superheat
# ================================================================
# Create an array to store gravity values for micro and earth gravity
gravity_values = [0.098, 9.8]  # [Micro_G, Earth_G, Optional: 2XG]
# Create an array to store labels for the legend
legend_labels = [f'Gravity = {g}' for g in gravity_values]
# Create a figure and axis for the scatter plot
fig, ax = plt.subplots()
# Variable to adjust font size
font_size = 24
# List to store fit equations
fit_equations = []
# Loop through the gravity values to plot data and linear fits
for i, gravity in enumerate(gravity_values):
    # Extract the data for the current gravity value
    data = [[row[1], row[0]] for row in ydata if row[2] == gravity] # (wall superheat, heat flux)
    # Convert the data to a numpy array
    data = numpy.array(data)
    # Plot the data as scatter points on a log-log scale with the corresponding color
    ax.scatter(data[:, 0], data[:, 1], label=legend_labels[i])
    # Fit a linear regression line to the data
    coeffs = numpy.polyfit(numpy.log(data[:, 0]), numpy.log(data[:, 1]), 1)
    # Generate x values for the regression line
    x_fit = numpy.linspace(min(data[:, 0]), max(data[:, 0]), 100)
    # Calculate the corresponding y values for the regression line
    y_fit = numpy.exp(coeffs[1]) * x_fit**coeffs[0]
    # Plot the linear fit as a line
    ax.plot(x_fit, y_fit, linestyle='--')
    # Store the fit equation
    fit_equation = f'y = {numpy.exp(coeffs[1]):.6f} * x^{coeffs[0]:.2f}'
    fit_equations.append(fit_equation)
# Set labels and title
ax.set_xlabel('Wall Superheat [Celcius]', fontsize = font_size)
ax.set_ylabel('Heat Flux [W/cm^2]', fontsize = font_size)
ax.set_title('Heat Flux vs. Wall Superheat (Log-Log Scale)', fontsize = font_size)
# Add a legend
ax.legend(prop={'size': font_size})
# Show the plot
plt.show()
# ================================================================
```

3. Added section for statistical analysis.

```
# ================================================================
# Change to original code. Added section for. statistical analysis

# Import Statistical Model package
import statsmodels.api as sm
# Convert your data to a numpy array
ydata = numpy.array(ydata)
# Separate the independent variables (X) and the dependent variable (y
X = ydata[:, 1:3]  # Columns 1 and 2 (superheat and gravity)
y = ydata[:, 0]    # Column 0 (heat flux)
# Take the logarithm of the independent variables
X_log = numpy.log(X)
# Add a constant term (intercept) to the independent variables
X_log = sm.add_constant(X_log)
# Fit the multiple linear regression model
model = sm.OLS(y, X_log).fit()
# Print the regression results
print(model.summary())
# ================================================================
```

Task 1.2

Please refer to the Appendix for extra code (Task 1.2 - Extra) that iterates over an specify array to find the best initial guess.

1. Define array to evaluate multiple sets.

```
# ================================================================
# Chage to original code. Define a list with the goal to evalute
# different initial guesses in a for loop

# Define a list of n values, perturbation, and NGEN to evaluate
n_values_to_evaluate = [
    [(-1, 0.00027, 4.00, 0.0630, 0.0, 0.0), 0.09, 6000],  # Set 1
    [(-1, 0.02091, 2.125, 0.334, 0.0, 0.0), 0.09, 6000],  # Set 2
    [(-1, 0.1050, 6.20, 0.1630, 0.0, 0.0), 0.09, 6000],   # Set 3
    [(-1, 6, 5, 0.5, 0.0, 0.0), 0.09, 6000],              # Set 4
```

2. For loop initiation.

```
# ================================================================
# Change to original code. Section added to initiate For Loop to
# iterate through n number of initial guessses and define their
# parameters

# Create an empty list to store summaries
results = []

# For Loop initiation
for n_values, perturbation, NGEN in n_values_to_evaluate:

```

3. Calculating q” predicted and RMSE.

```
    # ================================================================
    # Change to original code. Calculating q" predicted and RMSE with
    # respect to the data q"

    # Initialize values
    qpppred =  [[0.0]]
    qppdata =  [[0.0]]
    for i in range(ND-1):
        qpppred.append([0.0])
        qppdata.append([0.0])
    # Calculate predicted and data values to plot
    for i in range(ND):
        qpppred[i] = n1min*(ydata[i][1]**n2min) * ((ydata[i][2])**n3min)
        qppdata[i] = ydata[i][0]
    # Calculationg RMS btw q"data and q"pred
    for i in range(ND):
        rms_dev[i] = (numpy.array(qppdata[i]) - numpy.array(qpppred[i]))**2
    rms_dev = numpy.sqrt(numpy.sum(rms_dev) / ND)
    print('RMS error: ', rms_dev)
```

4. Dictionary to store results from each set of initial guesses.

```
    # ================================================================
    # Change to original code. After calculating n1min, n2min, n3min,
    # and rms_dev, create a dictionary to store these values
    iteration_result = {
        'Set': len(results) + 1,
        'n1i': n1i,
        'n2i': n2i,
        'n3i': n3i,
        'n4i': n4i,
        'n5i': n5i,
        'n1min': n1min,
        'n2min': n2min,
        'n3min': n3min,
        'n4min': n4min,
        'n5min': n5min,
        'p': perturbation,
        'NGEN': NGEN,
        'rms_dev': rms_dev,
        'aFerrmeanavgnMin': aFerrmeanavgnMin,
        # Add more values as needed
    }

    results.append(iteration_result)  # Append the summary dictionary to the list
    #=================================================================
```

5. Section to plot the q” predicted vs. data.

```
    # ================================================================
    # Change to original code. Adding perfect algorithm prediction reference line
    # and +/-10% uncertainty boundaries

    x_values = numpy.logspace(0, 3, 100)   # Adjust the range of x values as needed
    y_values = x_values                    # y = k*x
    y_values1 = x_values + (0.1 * x_values) # y = k*x + 0.1*x (+10% uncertainty)
    y_values2 = x_values - (0.1 * x_values) # y = k*x - 0.1*x (-10% uncertainty)
    # Plot reference equations
    plt.plot(x_values, y_values1, color='green', linestyle='--', label='y = k*x + 0.1*x')
    plt.plot(x_values, y_values2, color='green', linestyle='--', label='y = k*x - 0.1*x')
    plt.plot(x_values, y_values, color='red', linestyle='--', label='y = k*x')
    # Plot Heat Flux data vs. predicted
    plt.scatter(qppdata, qpppred, label='Data', color='blue', alpha=0.7)
    plt.title('Genetic Algorithm')
    plt.xlabel('measured heat flux (W/cm^2)')
    plt.ylabel('predicted heat flux (W/cm^2)')
    plt.loglog()
    plt.xlim(xmax = 1000, xmin = 10)
    plt.ylim(ymax = 1000, ymin = 10)

    plt.show()
    # ================================================================
```

Task 1.3

All changes made for task 1.2 apply for task 1.3. Please refer to the Appendix for extra code (Task 1.3 - Extra) that iterates over an specify array to find the best initial guess.

1. New error function.

```
        '''CALCULATING ERROR (FITNESS)'''
        for i in range(ND):

            # ================================================================
            # New function error equation to accomodate for 5 variable equation
            Ferr[i] = n[i][0]*lydata[i][0] + math.log(n[i][1]) + n[i][2]*lydata[i][1]
            Ferr[i] = Ferr[i] + n[i][3] * math.log( ydata[i][2] + n[i][4]*9.8*ydata[i][3] )
            Ferr[i] = Ferr[i] + n[i][5] * lydata[i][4]
            # ================================================================
            aFerr[i] = abs(Ferr[i])/abs(lydata[i][0])  #- absolute fractional error
```

2. New average error function.

```
''' CALCULATING MEAN ERROR FOR POPULATION'''
for i in range(ND):

    # ================================================================
    # New average function error equation to accomodate for 5 variable equation
    Ferravgn[i] = -1.*lydata[i][0] + math.log(n1avg[k]) + n2avg[k]*lydata[i][1]
    Ferravgn[i] = Ferravgn[i] + n3avg[k] * math.log( ydata[i][2] + n4avg[k]*9.8*ydata[i][3] )
    Ferravgn[i] = Ferravgn[i] + n5avg[k] * math.log(ydata[i][4])
    # ================================================================

    aFerravgn[i] = abs(Ferravgn[i])/abs(lydata[i][0])
```

3. New q” predicted and RMSE.

```
# ================================================================
# Change to original code. Calculating q" predicted and RMSE with
# respect to the data q"
# ================================================================
#initialize values
qpppred =  [[0.0]]
qppdata =  [[0.0]]
for i in range(ND-1):
    qpppred.append([0.0])
    qppdata.append([0.0])
#calculate predicted and data values to plot
for i in range(ND):
    qpppred[i] = n1min * (ydata[i][1]**n2min) * ((ydata[i][2] + n4min*9.8*ydata[i][3])**n3min) * (ydata[i][4])**n5min
    qppdata[i] = ydata[i][0]
#Calculationg RMS btw qppdata and qpppred
for i in range(ND):
    rms_dev[i] = (numpy.array(qppdata[i]) - numpy.array(qpppred[i]))**2
rms_dev = numpy.sqrt(numpy.sum(rms_dev) / ND)

print('RMS error: ', rms_dev)
```

4. 3D surface plot code.

```
# Import necessary libraries
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator, FuncFormatter, MaxNLocator

# Define a custom formatting function to round to a specific number of signi
def format_z(value, _):
    # Specify the number of significant figures you want (e.g., 3)
    num_significant_figures = 3
    return f"{value:.{num_significant_figures}g}"

# Define the range of X and Y values
X = numpy.linspace(1,20,100) # Gravity
Y = numpy.linspace(0,2,100)  # Surface tension parameter
X, Y = numpy.meshgrid(X, Y)

# Calculate Z using your equation
Z = n1min * ((X + n4min * 9.8 * Y) ** n3min) * (10 ** n5min)

# Create a 3D figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot the 3D surface
surf = ax.plot_surface(X, Y, Z, cmap='coolwarm')
fig.colorbar(surf, shrink=0.35, aspect=10, pad=0.1)

# Set labels for the axes
ax.set_xlabel('Gravity [m/s^2]', fontsize=12, labelpad=10)
ax.set_ylabel('Surf Tension Parameter γ', fontsize=12, labelpad=10)
ax.set_zlabel('q"/(Tw-Tsat)^n2 [W/(cm^2*°C^n2)]', fontsize=12, labelpad=10)

# Set limits for the axes
ax.set_xlim(1.0, 20.0)
ax.set_ylim(0.0, 2.0)
# ax.set_zlim(0.001, 0.003)

# ax.zaxis.set_major_locator(LinearLocator(10))
# Reduce the number of ticks on the z-axis
ax.zaxis.set_major_locator(MaxNLocator(nbins=5))

# Customize tick formatting for z-axis using the custom formatting function
ax.zaxis.set_major_formatter(FuncFormatter(format_z))

# Customize tick formatting for z-axis
ax.tick_params(axis='both', which='major', labelsize=10)

# Set the view perspective to orient the plot
ax.view_init(elev=20, azim=-65)  # Adjust the angles as needed

# Show the plot
plt.show()
```

Task 2.1

All changes to previous sections apply.

1. Training and validation data separation. The temperatures here are transform from Celsius to Kelvins.

```
# ================================================================
# This section of the code uses a ramdom shuffle to to separate the
# data into to sets: Training and Validation, 80% and 20%, respectively.
# The section also transforms the units of tempearture from Celcius to
# Kelvin.
# ================================================================
# Shuffle the data randomly
np.random.shuffle(yarray)
# Calculate the split index for training and validation
split_index = int(0.8 * len(yarray))
# Split the data into training and validation sets
t_data = yarray[:split_index]
t_data_k = yarray[:split_index]
v_data = yarray[split_index:]
v_data_k = yarray[split_index:]
# Convert temperature from Celcius to Kelvins
# for calculation uniformity
for i in range (len(t_data)):
    t_data_k[i][1] =  t_data[i][1] + 273
    t_data_k[i][0] = t_data[i][0] + 273
for i in range (len(v_data)):
    v_data_k[i][1] =  v_data[i][1] + 273
    v_data_k[i][0] = v_data[i][0] + 273
```

2. Variables definition.

```
# ============================================
# Create Variables for the Bolztmann constt
# & the reference convective heat transfer
# coefficient for the fin section
# ============================================
Bolt = 5.67 * 10 **-8
hc = 70
```

3. New error function.

```
'''CALCULATING ERROR (FITNESS)'''
for i in range(ND):
    # ============================================================
    # New function error equation to accomodate HPHE error function
    # ============================================================
    # Calculate the temperature difference
    diff_t = (t_data[i][0] - t_data[i][1])
    # Calculate the temperature sum
    sum_t = (t_data_k[i][0] + t_data_k[i][1])
    # Calculate the sum of the temperature to the power 3
    t_p3 = (t_data_k[i][0])**3 + (t_data_k[i][1])**3
    # Calculate hr
    hr = n[i][2] * (1+ (1/hc)*n[i][3]* (Bolt)*((sum_t))*(t_p3))
    # Compute error function
    Ferr[i] = t_data[i][2] - (hr * (n[i][1] * diff_t)/(n[i][1] + hr))
    # ============================================================
    # New absolute fractional error equation to accomodate HPHE error function
    # ============================================================
    aFerr[i] = abs(Ferr[i])/abs(t_data[i][2])  #- absolute fractional error
```

4. New average error function.

```
''' CALCULATING MEAN ERROR FOR POPULATION'''
for i in range(ND):
    # ============================================================
    # New average function error equation to accomodate HPHE error function
    # ============================================================
    # Calculate the temperature difference
    diff_t = (t_data[i][0] - t_data[i][1])
    # Calculate the temperature sum
    sum_t = (t_data_k[i][0] + t_data_k[i][1])
    # Calculate the sum of the temperature to the power 3
    t_p3 = (t_data_k[i][0])**3 + (t_data_k[i][1])**3
    # Calculate hr
    hr_avg = n2avg[k] * ( 1 + (1/hc) * n3avg[k] * (Bolt)*((sum_t))*(t_p3))
    # Compute average error function
    Ferravgn[i] = t_data[i][2] - (hr_avg * (n1avg[k] * diff_t)/(n1avg[k] + hr_avg))
    # ============================================================
    # New average absolute error equation to accomodate HPHE error function
    # ============================================================
    aFerravgn[i] = abs(Ferravgn[i])/abs(t_data[i][2])
#-------------
aFerrmeanavgn[k] = numpy.mean(aFerravgn)
```

5. RMSE for training data.

```
# ================================
#RMS calculation for Traning Data
# ================================
#initialize values
qpppred =  [[0.0]]
qppdata =  [[0.0]]
for i in range(ND-1):
    qpppred.append([0.0])
    qppdata.append([0.0])
# Calculate predicted and data values to plot
for i in range(ND):
    # Calculate the temperature difference
    diff_t = (t_data[i][0] - t_data[i][1])
    # Calculate the temperature sum
    sum_t = (t_data_k[i][0] + t_data_k[i][1])
    # Calculate the sum of the temperature to the power 2
    t_p3 = (t_data_k[i][0])**3 + (t_data_k[i][1])**3
    # Calculate hr
    hr_min = n2min * (1 + (1/hc) * n3min * (Bolt) * ((sum_t)) * (t_p3))
    # Compute q' predicted
    qpppred[i] = hr_min * n1min * diff_t/(n1min + hr_min)
    qppdata[i] = t_data[i][2]
#Calculationg RMS btw data and predicted
for i in range(ND):
    rms_dev[i] = (numpy.array(qppdata[i]) - numpy.array(qpppred[i]))**2
rms_dev = numpy.sqrt(numpy.sum(rms_dev) / ND)
print('RMSE training: ', rms_dev)
```

6. RMSE for validation data.

```
# ================================
#RMS calculation for Validation Data
# ================================
#initialize values
qpppred_val =  [[0.0]]
qppdata_val =  [[0.0]]
for i in range(len(v_data)-1):
    qpppred_val.append([0.0])
    qppdata_val.append([0.0])
# Calculate predicted and data values to plot
for i in range(len(v_data)):
    # Calculate the temperature difference
    diff_t = (v_data[i][0] - v_data[i][1])
    # Calculate the temperature sum
    sum_t = (v_data_k[i][0] + v_data_k[i][1])
    # Calculate the sum of the temperature to the power 2
    t_p3 = (v_data_k[i][0])**3 + (v_data_k[i][1])**3
    # Calculate hr
    hr_min = n2min * (1 + (1/hc) * n3min * (Bolt) * ((sum_t)) * (t_p3))
    # Compute q' predicted
    qpppred_val[i] = hr_min * n1min * diff_t/(n1min + hr_min)
    qppdata_val[i] = v_data[i][2]
#Calculationg RMS btw data and predicted
for i in range(len(v_data)):
    rms_dev_val[i] = (numpy.array(qppdata_val[i]) - numpy.array(qpppred_
rms_dev_val = numpy.sqrt(numpy.sum(rms_dev_val) / len(v_data))
print('RMSE Validation: ', rms_dev_val)
```

Discussion.

Task 1.1

When we observe the behavior of the data provided for micro and earth gravity at 5.5 MPa and the surface tension parameter γ, there is not a significant difference in the heat flux for the solution with respect to gravity. Figure 1 shows that the behavior of heat flux is not affected by gravity. To confirm this behavior, a Python statsmodel library analysis was used. R-squared is possibly the most important measurement produced by this summary. R-squared is the measurement of how much of the independent variable is explained by changes in our dependent variables. In percentage terms, 0.983 would mean our model explains 98.3% of the change in heat flux. Next, P > |t| is the most important statistic in the summary; here, the variables x1 and x2 represent wall superheat and gravity, respectively. It uses the t-statistic to produce the p-value, a measurement of how likely our coefficient is measured through our model by chance. The p-value of 0.375 for gravity is saying there is a 37.5% chance that the heat flux variable has no effect on the dependent variable, gravity, and our results are produced by chance. More interesting is when data from 2X gravity is included in the analysis. In Figure 3, it is shown that the 2X gravity curve-fit is distinctive from micro and earth gravity. If the same statistical analysis is conducted, R-squared was found to be 0.917, and more importantly, it shows that for the whole set of data, gravity starts to influence the resultant heat flux.

Task 1.2

Table 1 shows four randomly selected values somewhat close to the initial values provided. It can be noticed that each set results in drastically different values for RMSE and MAE. In Tables 2 and 3, the perturbations are increased to 0.15 and 0.30, respectively. As we increase the perturbation values, the algorithm exhibits the expected results. If the initial guess was far away from the solution, it will probably get closer, the opposite if the value is somewhat close. If the perturbations are increased for those values, the result would be that with a bigger step through the hypersurface, the algorithm would end up stepping away from the solution, instead of towards it. In Table 4, it is noted that if the number of generations is increased, the error would be minimized; the more generations, the lower the error achieved. Here, the generations were set at 3000, 6000, 12000, and 24000.

To find the constants n1, n2, n3 for the best fit to the data, a separate code was written from the original code provided. The goal was to iterate through a range of possible values for each of the constants and generate an array. The code eventually summarizes the results obtained for the best initial values. The criteria were the set that achieved the lowest RMSE and MAE. Table 5 exhibits the best consistent initial values obtained. As it is noted, irrespective of the perturbation introduced, the RMSE and MAE are below 23 and 0.031, respectively. As expected, the top initial values for RMSE and MAE do not always correlate. This is mainly because they are different metrics. RMSE is often preferred when you want to penalize large errors more heavily or when the error distribution is assumed to be Gaussian. MAE is preferred when you want a more robust metric that is less affected by outliers and is easier to interpret in the context of the problem. The choice between RMSE and MAE depends on the specific requirements and characteristics of the modeling task. For this section, we are providing both sets of initial guesses and representing their equations and behavior in Figures 5 and 6. In the predicted vs. data graph, it can be observed that the MAE scatter points are distributed more on top of the perfect prediction, as opposed to the RMSE, which is more evenly distributed along the bandwidth of ±10%.

To extend the analysis, a comparison of the given set and the best-behaved set is shown in Table 6 with the goal of making evident that the chosen set behaves consistently well across RMSE and MAE.

To determine if the tightness of the fit, the RMSE and the uncertainty must be compared. I n this case, the uncertainty is estimated to be ±10%. Two scenarios need to be considered:

1. Lower Bound of Uncertainty (-10%)
   If we assume the worst-case scenario where the measured heat flux data is underestimated by 10%, we should subtract 10% from the RMSE for comparison:
   RMSE - 10% of RMSE = 13.79 - 0.10 * 13.79 = 13.79 - 1.379 = 12.411
   Now, the adjusted RMSE is 12.411.
2. Upper Bound of Uncertainty (+10%)
   If we assume the worst-case scenario where the measured heat flux data is overestimated by 10%, we should add 10% to the RMSE for comparison:
   RMSE + 10% of RMSE = 13.79 + 0.10 * 13.79 = 13.79 + 1.379 = 15.169
   Now, the adjusted RMSE is 15.169.

It appears that the fit, as measured by the RMSE, is generally tighter than the estimated uncertainty in the measured heat flux data, regardless of whether the data is underestimated or overestimated by 10%. The model is performing reasonably well in terms of fitting the data. The RMSE is a measure of the spread of errors in the predictions, and having it be smaller than the uncertainty indicates that the model's predictions are relatively consistent and accurate.

Task 1.3

To find the best set of initial values, a similar approach to Task 1.2 was taken (see Appendix Task 1.3 extra). 16,807 combinations were tested, and only the best consistent initial value for the lowest RMSE was extracted. Table 7 and 8 show the results for a test run of 243 sets. The algorithm then sorts the top 5 initial guesses according to RMSE and MAE, respectively. Table 9 and 10 contain the results of the best set of initial guesses found. Figure 8 represents the behavior of the solution space surface. The results are consistent with the discussions during the lecture. The gravitational term is to the power of ~0.5, and the surface tension

parameter is scaled by an n4, the 3D surface contains the described behavior.

Task 2.2

Step 1: To identify the optimal initial guess, we conducted a brief experiment using three randomly selected values for the parameter 'n': 100, 50, and 10. Given the uncertainty surrounding the ideal initial guess, we opted for a high perturbation setting of 0.9, coupled with a substantial number of generations (50,000). The outcomes of the first trial are presented in Figure 12. In this graph, significant noise is evident, primarily stemming from the elevated perturbation rate employed. Nevertheless, it is noteworthy that we successfully achieved a minimum Mean Absolute Error (MAE) of 1.7%, comfortably below the 5% threshold. Consequently, we have decided to select our subsequent initial guess based on the lowest 'n' parameter values. Furthermore, we observe a close alignment between the validation MAE and the trained MAE, indicating that our model is neither overfitting nor underfitting; it is effectively learning and generalizing. Here, also the predicted discrete output is compared to the collected data to assess the quality of the fit (see Figure 13).

Step 2: After analyzing the results, we determined the best range of values for the 'n' parameters. Since our error rate (MAE) was less than 5%, we were confident in our solution. We also noticed that the parameters stabilized much sooner than expected, so we reduced the number of generations to 6,000. For our next attempt, we chose the values (3.5, 50, 22) for the 'n' parameters, used a perturbation rate of 0.09, and ran it for 6,000 generations. New results are presented in Figure 14. In this run, you can clearly see that reducing the perturbation helped us reduce the amount of noise in our results. Our model showed good convergence, with an acceptable Mean Absolute Error (MAE) for both the trained and validation data. This suggests that the values (3.5, 50, 22) can be considered as a desirable set of solutions for our model.

Testing the Individual Impact of Each Parameter on the Output: In this test, we intentionally deviated slightly from the ideal initial guess for 'n1' to assess its impact on the model. Instead of the ideal value of 3.5, we used 'n1 = 10'. Below are the results of this deviation. As depicted in Figure 15, a significant shift in the output is evident, accompanied by a notable increase in MAE to 23%. This highlights the model's high sensitivity to changes in the 'n1' value. Additionally, Figure 16 illustrates the loss of our optimal performance.

In our latest test, we examined the impact of altering both 'n2' and 'n3' together and have presented the results in Figure 17. Notably, there is a lack of discernible changes in the outcome, suggesting that the model is less sensitive to variations in 'n2' and 'n3' when compared to 'n1'.

Table 13 provides the results to easily view all the output for each set of initial guesses.

Work methodology

The methodology followed for the project involved both members of the group working on each section of the project with the goal of understanding all the material. We began by working on each task sequentially. First, we would individually attempt the tasks, and later, we would meet to discuss results, approaches, and our understanding of the material. This approach facilitated constructive discussions on disagreements and helped us find the best solutions. The report was divided as follows:

Task 1	Daniel Garcia
Task 2	Ali Rakhsha
Report Draft Task 1	Daniel Garcia
Report Draft Task 2	Ali Rakhsha
Final Report	Daniel Garcia

Conclusion.

The capability of the predictive genetic algorithm is easily adaptable to different tasks. Once the algorithm was setup for a simple prediction that required 3 parameters, it then can be used to expand within the problem and outside of it, as it was for the second section of the project.
There were ???? main take away. First, the importance of clear statistical interpretation of the results. RMSE and MAE are both useful metrics for assessing model accuracy, but they have different properties and are suited to different situations. RMSE is often preferred when the goal is to penalize large errors more heavily or when the error

distribution is assumed to be Gaussian. MAE is preferred when the desire is a more robust metric that is less affected by outliers and is easier to interpret in the context of the problem. The choice between RMSE and MAE depends on the specific requirements and characteristics of the problem at hand.

Second, initial guesses play an intrinsic role in the performance of the algorithm. For the given tasks initial values were given and when they were not it was not complicated to find convergence for the solutions. Nevertheless, the computational power required to search through a problem as it augments the number of complexities is incredibly considerable. From task 1.2 to 1.3 by only adding 2 variables the required time for the algorithm to search through an array increments exponentially. For 3 constants if the search linear space is of 25 values each, the algorithm must go through $25^3 = 15{,}625$, if we add just two more constants $25^5 = 9{,}765{,}625$. Which brings me to my third and last take away. The advantages and disadvantages of raw data analysis versus dimensionless data analysis. The choice between them depends on the specific goals of the analysis and the nature of the available data. There are clear advantages in raw data analysis: (1) no need of transformation or manipulation of the original data, and (2) the results are a direct interpretation of the measure data. The disadvantage lies in: (1) the noise of the data, (2) visualization of results due to the high dimensionality, (3) computational analysis is highly intensive. In contrast, dimensionless data analysis: (1) simplifies the data by removing units or scales, making it easier to compare variables and perform certain types of analyses, (2) can reduce the dimensionality of the dataset, which can be helpful for visualization and simplifying models. Nevertheless, it does not come with its disadvantages: (1) loss of some information, which can make the interpretation of results less intuitive, (2) often relies on assumptions about the appropriate transformations, which may not always hold true for all variables, and (3) removing units or scales can make it difficult to retain the context of the data, which can be important for some types of analysis. To conclude, sometimes, as in task 2 of the project, a combination of both approaches may be the most effective way to extract meaningful insights from your data using dimensionless data for certain analyses while also preserving and exploring the raw data for deeper understanding.

Appendix.

Task 1.1 (a)

Install this code into a first cell of a new Anaconda notebook. Run the code, which will simply print the array and a single array element in the output region below the cell. Inspect the output to confirm it matches the array in the Appendix on the last page of this write-up. Note that in this code I have commented-out portions of the data with pressures other than 5.5 kPa. More on that in the next Task.

In [4]:

```
'''>>>>> start CodeP1.1F23
    V.P. Carey ME249, Fall 2023'''

# version 3 print function
from __future__ import print_function
# seed the pseudorandom number generator
from random import random
from random import seed
# seed random number generator
seed(1)

#import math and numpy packages
import math
import numpy

%matplotlib inline
# importing the required module
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 8] # for square canvas

#import copy
from copy import copy, deepcopy

#create arrays
ydata = []
lydata = []

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
# this is an array that is essentially a list of lists

#assembling data array
#store array where rows are data vectors
#[heat flux, superheat, gravity, surface tension parameter, pressure]

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([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])
```

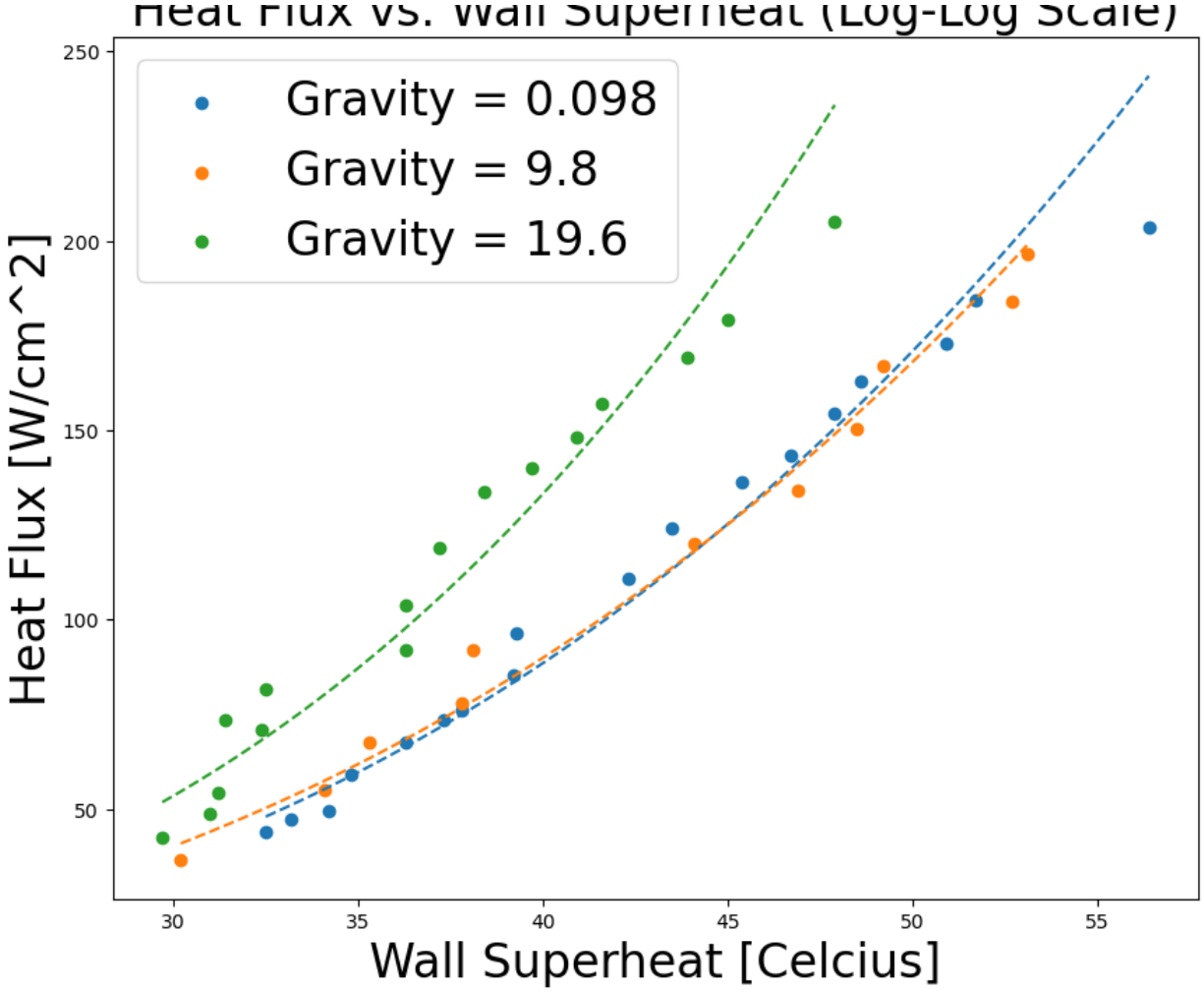
```
ydata.append([60.0, 37.5, 9.8, 0.00, 7.0])
ydata.append([53.0, 37.0, 9.8, 0.00, 7.0])

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])
'''
# print the data array
#print ('ydata =', ydata)

''' need deepcopy to create an array of the same size as ydata,
#   since this array is a list(rows) of lists (column entries)  '''
lydata = deepcopy(ydata)  # create array to store ln of data values

# ================================================================
# Change to original code. Moved setion after the ydata array was
# constructed for the lenght of ND & NS be define as we change the
# size of the array

# Parameters for Evolution Loop
# set data parameters
ND = len(ydata)       #number of data vectors in array
DI = 5                #number of data items in vector
NS = ND               #total number of DNA strands
# ================================================================

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
#now store log values for data
for j in range(DI):
    for i in range(ND):
        lydata[i][j]=math.log(ydata[i][j]+0.000000000010)

#OK now have stored array of log values for data

#end CodeP1.1F23
```

Task 1.1 (b)

Using the data in the first two groups in the Appendix listing (for g = 0.098 and 9.8 m/s plot of heat flux versus wall superheat for these two gravity levels to get a sense of how strongly heat flux varies with gravity and superheat. You can do this using a separate Python program (recommended) or you can use another platform such as Excel or Matlab if you prefer. A log-log plot is done in the second python code file so you can see an example there.

In [5]:
```
# ================================================================
# Change to original code. Added section for ploting q" vs wall superheat

# Create an array to store gravity values for micro and earth gravity
```

```
legend_labels = [f'Gravity = {g}' for g in gravity_values]
# Create a figure and axis for the scatter plot
fig, ax = plt.subplots()
# Variable to adjust font size
font_size = 24
# List to store fit equations
fit_equations = []
# Loop through the gravity values to plot data and linear fits
for i, gravity in enumerate(gravity_values):
    # Extract the data for the current gravity value
    data = [[row[1], row[0]] for row in ydata if row[2] == gravity] # (wall su
    # Convert the data to a numpy array
    data = numpy.array(data)
    # Plot the data as scatter points on a log-log scale with the correspondin
    ax.scatter(data[:, 0], data[:, 1], label=legend_labels[i])
    # Fit a linear regression line to the data
    coeffs = numpy.polyfit(numpy.log(data[:, 0]), numpy.log(data[:, 1]), 1)
    # Generate x values for the regression line
    x_fit = numpy.linspace(min(data[:, 0]), max(data[:, 0]), 100)
    # Calculate the corresponding y values for the regression line
    y_fit = numpy.exp(coeffs[1]) * x_fit**coeffs[0]
    # Plot the linear fit as a line
    ax.plot(x_fit, y_fit, linestyle='--')
    # Store the fit equation
    fit_equation = f'y = {numpy.exp(coeffs[1]):.6f} * x^{coeffs[0]:.2f}'
    fit_equations.append(fit_equation)
# Set labels and title
ax.set_xlabel('Wall Superheat [Celcius]', fontsize = font_size)
ax.set_ylabel('Heat Flux [W/cm^2]', fontsize = font_size)
ax.set_title('Heat Flux vs. Wall Superheat (Log-Log Scale)', fontsize = font_s
# Add a legend
ax.legend(prop={'size': font_size})
# Show the plot
plt.show()
# ======================================================================
```



Task 1.1 (extra)

Adding a section to confirm and quantify the data dependancies using statsmodels library. The desire values of g to be analyze need to be comment or uncomment in the ydata definition section - Task 1.1 (a).

In [6]:

```
# ======================================================================
# Change to original code. Added section for. statistical analysis

# Import Statistical Model package
import statsmodels.api as sm
# Convert your data to a numpy array
ydata = numpy.array(ydata)
# Separate the independent variables (X) and the dependent variable (y)
X = ydata[:, 1:3]  # Columns 1 and 2 (superheat and gravity)
y = ydata[:, 0]    # Column 0 (heat flux)
# Take the logarithm of the independent variables
X_log = numpy.log(X)
# Add a constant term (intercept) to the independent variables
X_log = sm.add_constant(X_log)
# Fit the multiple linear regression model
model = sm.OLS(y, X_log).fit()
# Print the regression results
print(model.summary())
# ======================================================================
```

```
==============================================================================
Dep. Variable:                      y   R-squared:                       0.917
Model:                            OLS   Adj. R-squared:                  0.913
Method:                 Least Squares   F-statistic:                     232.0
Date:                Fri, 29 Sep 2023   Prob (F-statistic):           2.00e-23
Time:                        09:27:36   Log-Likelihood:                -183.82
No. Observations:                  45   AIC:                             373.6
Df Residuals:                      42   BIC:                             379.1
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const       -935.1428     48.736    -19.188      0.000   -1033.496    -836.789
x1           283.3898     13.167     21.523      0.000     256.819     309.961
x2             5.1245      0.921      5.567      0.000       3.267       6.982
==============================================================================
Omnibus:                        1.446   Durbin-Watson:                   0.283
Prob(Omnibus):                  0.485   Jarque-Bera (JB):                1.386
Skew:                          -0.396   Prob(JB):                        0.500
Kurtosis:                       2.665   Cond. No.                         89.1
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correct
ly specified.
```

Task 1.2

Install CodeP1.2 F23 pasted it in the next cell in the Anaconda notebook so that running both cells in sequence will execute the genetic algorithm for simple equation models used in this Task.

In [275…

```
'''>>>>> start CodeP1.1F23
    V.P. Carey ME249, Fall 2023'''

# version 3 print function
from __future__ import print_function
# seed the pseudorandom number generator
from random import random
from random import seed
# seed random number generator
seed(1)

#import math and numpy packages
import math
import numpy

%matplotlib inline
# importing the required module
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 8] # for square canvas

#import copy
from copy import copy, deepcopy

#create arrays
ydata = []
lydata = []

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
# this is an array that is essentially a list of lists

#assembling data array
#store array where rows are data vectors
#[heat flux, superheat, gravity, surface tension parameter, pressure]

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])
'''
# print the data array
# print ('ydata =', ydata)

''' need deepcopy to create an array of the same size as ydata,
#   since this array is a list(rows) of lists (column entries)  '''
lydata = deepcopy(ydata)  # create array to store ln of data values

# ================================================================
# Change to original code. Moved setion after the ydata array was
# constructed for the lenght of ND & NS be define as we change the
# size of the array

#Parameters for Evolution Loop
#set data parameters
ND = len(ydata)       #number of data vectors in array
DI = 5                #number of data items in vector
NS = len(ydata)       #total number of DNA strands
# ================================================================

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
#now store log values for data
for j in range(DI):
    for i in range(ND):
        lydata[i][j]=math.log(ydata[i][j]+0.000000000010)

#OK now have stored array of log values for data

#end CodeP1.1F23
```

The objective here is to use machine-learning tools to determine how the heat flux varies with the other
parameters in the multivariate data. The data reflect the combined variation with these parameters,
and the objective of the machine learning analysis is to explore what the individual dependencies are.

The code can be modified by commenting in or out the n_values_to__evalutae defined list:

- Results for 4 different sets of initial guesses for n1,n2,n3
- Given initial guesses for n1,n2,n3. Range of perturbations from 90 to 1 percent
- Best found initial guesses for n1,n2,n3. Range of perturbations from 90 to 1 percent

Added section:

- For Loop for running multiple sets

- Calculation of RMSE
- Plot of q" predicted v. data
- Results table at the end

In [276…

```
'''>>>>> start CodeP1.2F23
    V.P. Carey ME249, Fall 2023'''

'''INITIALIZING PARAMETERS'''

# ====================================================================
# Chage to original code. Define a list with the goal to evalute
# different initial guesses in a for loop

# Define a list of n values, perturbation, and NGEN to evaluate
n_values_to_evaluate = [
    [(-1, 0.00027, 4.00, 0.0630, 0.0, 0.0), 0.09, 6000],  # Set 1
    [(-1, 0.02091, 2.125, 0.334, 0.0, 0.0), 0.09, 6000],  # Set 2
    [(-1, 0.1050, 6.20, 0.1630, 0.0, 0.0), 0.09, 6000],   # Set 3
    [(-1, 6, 5, 0.5, 0.0, 0.0), 0.09, 6000],              # Set 4

#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.90, 6000], # Given initial g
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.95, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.80, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.85, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.70, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.75, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.60, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.55, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.09, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.09, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.09, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.09, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.07, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.07, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.07, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.07, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.05, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.05, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.05, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.05, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.03, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.03, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.03, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.03, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.01, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.01, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.01, 6000],
#     [(-1, 0.00027, 4.00, 0.063, 1.215, 0.145), 0.01, 6000],

#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.90, 6000], # Best initial g
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.95, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.80, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.85, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.70, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.75, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.60, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.55, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.09, 6000],
```

```
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.09, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.09, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.09, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.07, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.07, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.07, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.07, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.05, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.05, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.05, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.05, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.03, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.03, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.03, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.03, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.01, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.01, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.01, 6000],
#     [(-1, 0.00735, 2.64, 0.0455, 1.215, 0.145), 0.01, 6000],
]
# ================================================================

# ================================================================
# Change to original code. Section added to initiate For Loop to
# iterate through n number of initial guessses and define their
# parameters

# Create an empty list to store summaries
results = []

# For Loop initiation
for n_values, perturbation, NGEN in n_values_to_evaluate:

    n = []
    ntemp = []
    gen=[0]
    n1avg = [0.0]
    n2avg = [0.0]
    n3avg = [0.0]
    n4avg = [0.0]
    n5avg = [0.0]
    meanAFerr=[0.0]
    aFerrmeanavgn=[0.0]
    rms_dev = [0.0] # New rms_dev added term

    # Set program parameters
    NGEN = NGEN      # number of generations (steps)
    MFRAC = 0.5   # faction of median threshold
    perturbation = perturbation # pertubation value

    # here the number of data vectors equals the number of DNA strands (or orga
    # they can be different if they are randomly paired to compute Ferr (survi
    for k in range(NGEN-1):
        gen.append(k+1)   # generation array stores the
        meanAFerr.append(0.0)
        aFerrmeanavgn.append(0.0)
        n1avg.append(0.0)
        n2avg.append(0.0)
        n3avg.append(0.0)
        n4avg.append(0.0)
```

```
    n5avg.append(0.0)

'''guesses for initial solution population'''
n0i, n1i, n2i, n3i, n4i, n5i = n_values # Assign values from the array

#- initialize arrays before start of evolution loop EL
#then - create array of DNA strands n[i] and ntemp[i] with dimesnion NS = 

#i initialize array where rows are dna vectors [n0i,n1i,...n5i] with randor
n =  [[-1., n1i+0.001*random(), n2i+0.1*random(), n3i+0.0001*random(), n4i-
for i in range(ND):
    n.append([-1., n1i+0.0001*random(), n2i+0.001*random(), n3i+0.0001*ranc
#print (n) # uncomment command to print array so it can be checked

# store also in wtemp
ntemp = deepcopy(n)

#initialize Ferr values an dother loop parameters
#define arrays of Ferr (error) functions
#individual solution error and absoute error
Ferr =  [[0.0]]
#population average solution error and absoute error
Ferravgn =  [[0.0]]
aFerr =  [[0.0]]
aFerravgn =  [[0.0]]

#store zeros in ND genes
for i in range(ND-1):
    #individual solution error and absoute error
    Ferr.append([0.0])
    aFerr.append([0.0])
    #population average solution error and absoute error
    Ferravgn.append([0.0])
    aFerravgn.append([0.0])
    rms_dev.append([0.0]) # New array rms_dev

aFerrmeanavgnMin = 1000000000.0
# these store the  n values for minimum population average error durng NGE
n1min = 0.0
n2min = 0.0
n3min = 0.0
n4min = 0.0
n5min = 0.0
aFerrta = 0.0
# these store the time averaged n values durng from generation 800 to NGEN
n1min = 0.0
n1ta = 0.0
n2ta = 0.0
n3ta = 0.0
n4ta = 0.0
n5ta = 0.0

'''START OF EVOLUTION LOOP'''
# ----------------------------------------
# k is generation number, NGEN IS TOTAL NUMBER OF GENERATIONS COMPUTED
for k in range(NGEN):

    '''In this program , the number of organisms (solutions) NS is taken to
    number of data points ND so for each generation, each solution can be c
    data point and all the data is compared in each generation.  The order
```

```
that holds the solution constants is constantly changing due to mating
is random.'''

'''CALCULATING ERROR (FITNESS)
In this program, the absolute error in the logrithm of the physical hea
used to evaluate fitness.'''

# Here we calculate error Ferr and absolute error aFerr for each data p
# for specified n(i), and calculate (mean aFerr) = aFerrmean
# and (median aFerr) = aFerrmedian for the data collection and specifie
# Note that the number data points ND equals the number of solutions (
#=============================================================
'''CALCULATING ERROR (FITNESS)'''
for i in range(ND):

    Ferr[i] = n[i][0]*lydata[i][0] + math.log(n[i][1]) + n[i][2]*lydata
    Ferr[i] = Ferr[i] + n[i][3]*math.log( ydata[i][2] )

    aFerr[i] = abs(Ferr[i])/abs(lydata[i][0])  #- absolute fractional e
#-------------
aFerrmean = numpy.mean(aFerr) #mean error for population for this genei
meanAFerr[k]=aFerrmean  #store aFerrmean for this generation gen[k]=k
aFerrmedian = numpy.median(aFerr) #median error for population for this

'''SELECTION'''
#pick survivors
#[2] calculate survival cutoff, set number kept = nkeep = 0
#=============================================
clim = MFRAC*aFerrmedian  #cut off limit is a fraction/multiplier MFRAC
nkeep = 0

# now check each organism/solution to see if aFerr is less than cut of
#if yes, store n for next generation population in ntemp, at end nkeep
#and number of new offspring = NS-nkeep
#=============================================
for j in range(NS):  # NS Ferr values, one for each solution in popula
    if (aFerr[j] < clim):
        nkeep = nkeep + 1
        #ntemp[nkeep][0] = n[j][0] = -1 so it is unchanged;
        ntemp[nkeep-1][1] = n[j][1];
        ntemp[nkeep-1][2] = n[j][2];
        ntemp[nkeep-1][3] = n[j][3];
        ntemp[nkeep-1][4] = n[j][4];
        ntemp[nkeep-1][5] = n[j][5];
#now have survivors in leading entries in list of ntemp vectors from 1
#compute number to be added by mating
nnew = NS - nkeep

'''MATING'''
#[4] for nnew new organisms/solutions,
# randomly pick two survivors, randomly pick DNA (n) from pair for eacl
#=============================================================
for j in range(nnew):
    # pick two survivors randomly
    nmate1 = numpy.random.randint(low=0, high=nkeep+1)
    nmate2 = numpy.random.randint(low=0, high=nkeep+1)

    #then randomly pick DNA from parents for offspring

    '''here, do not change property ntemp[nkeep+j+1][0], it's always fi
```

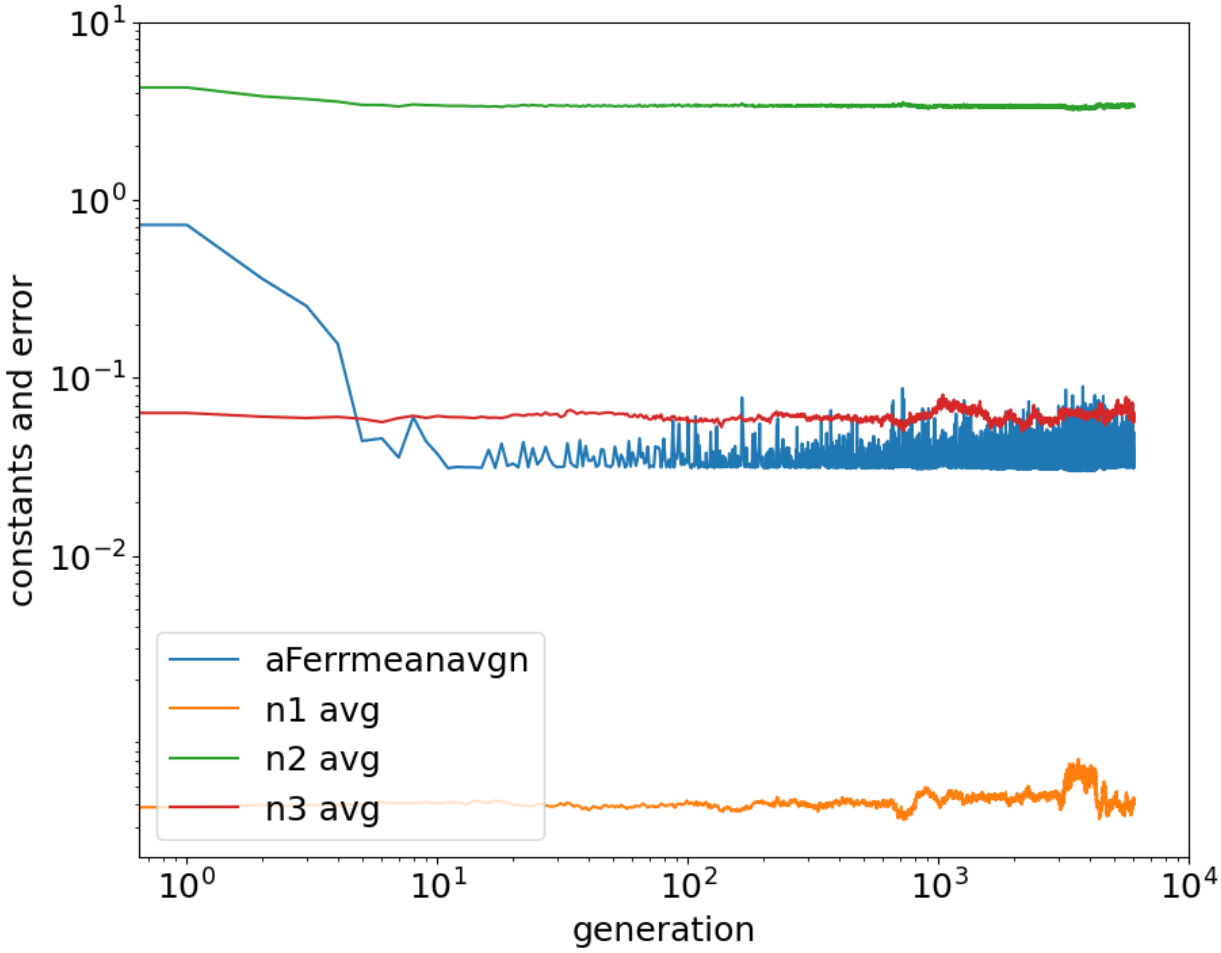
```
            # ================================================================
            # Change to original code. 0.09 substitute by "perturbation" varial
            # ================================================================

            #if (numpy.random.rand() < 0.5)
            #    ntemp[nkeep+j+1][0] = n[nmate1][0]
            #else
            #    ntemp[nkeep+j+1][0] = n[nmate2][0]

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][1] = n[nmate1][1]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][1] = n[nmate2][1]*(1.+perturbation*2.*(0.5-nur

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][2] = n[nmate1][2]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][2] = n[nmate2][2]*(1.+perturbation*2.*(0.5-nur

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][3] = n[nmate1][3]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][3] = n[nmate2][3]*(1.+perturbation*2.*(0.5-nur
            '''
            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][4] = n[nmate1][4]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][4] = n[nmate2][4]*(1.+perturbation*2.*(0.5-nur

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][5] = n[nmate1][5]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][5] = n[nmate2][5]*(1.+perturbation*2.*(0.5-nur
            '''
        #============================================
        n = deepcopy(ntemp)   # save ntemp as n for use in next generation (nex

        '''AVERAGING OVER POPULATION AND OVER TIME, FINDING MINIMUM ERROR SET (
        # [6] calculate n1avg[k], etc., which are average n values for populat
        # at this generation k
        #============================================
        #initialize average n's to zero and sum contribution of each member of
        n1avg[k] = 0.0;
        n2avg[k] = 0.0;
        n3avg[k] = 0.0;
        n4avg[k] = 0.0;
        n5avg[k] = 0.0;
        for j in range(NS):
            n1avg[k] = n1avg[k] + n[j][1]/NS;
            n2avg[k] = n2avg[k] + n[j][2]/NS;
            n3avg[k] = n3avg[k] + n[j][3]/NS;
            n4avg[k] = n4avg[k] + n[j][4]/NS;
            n5avg[k] = n5avg[k] + n[j][5]/NS;

        # Here we compute aFerravgn[i] = absolute Ferr of logrithm data point i
        # for this solutions generation k
        # aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of
        #
        #============================================
```

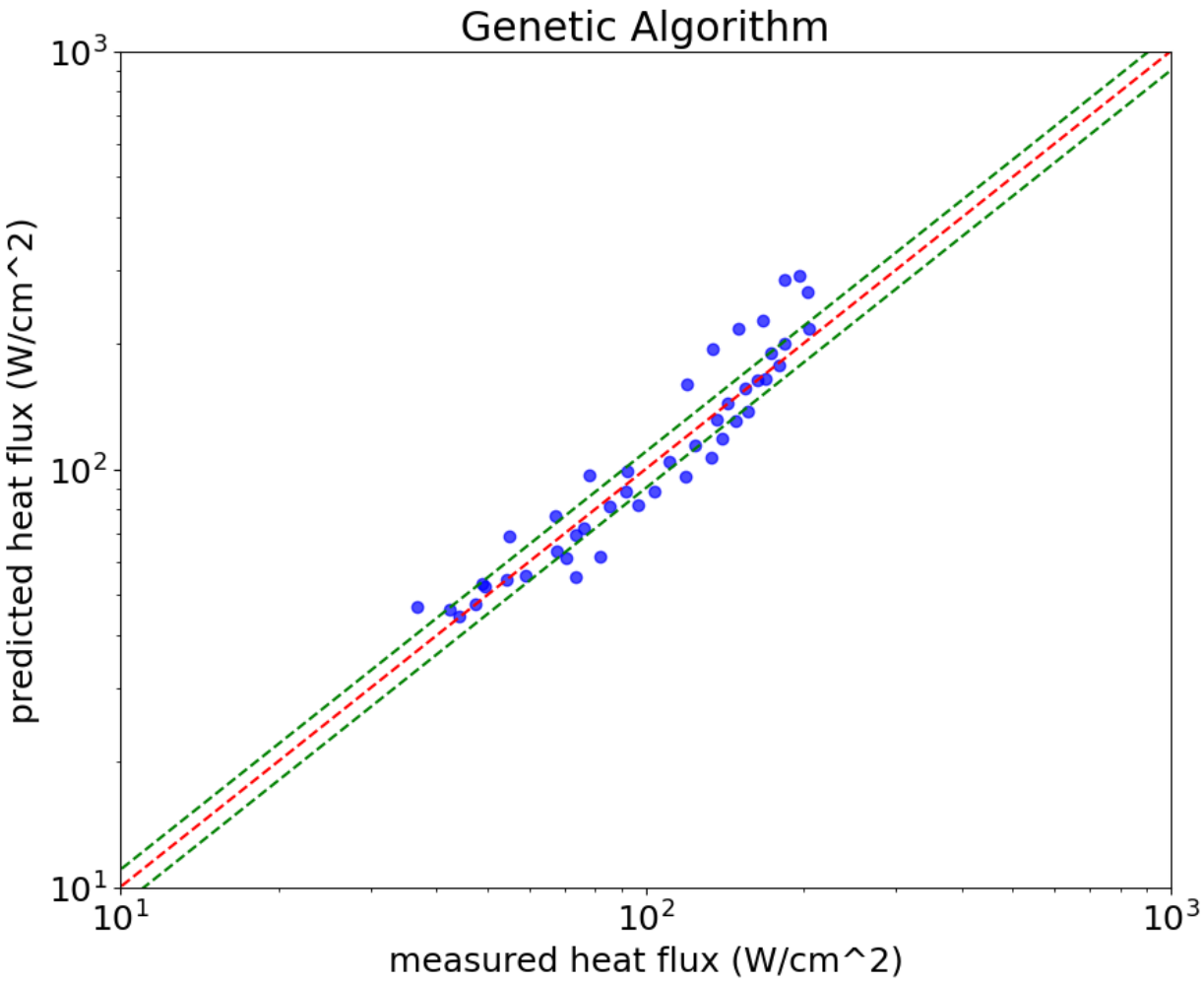
```
        ''' CALCULATING MEAN ERROR FOR POPULATION'''
        for i in range(ND):
            Ferravgn[i] = -1.*lydata[i][0] + math.log(n1avg[k]) + n2avg[k]*lyda
            Ferravgn[i] = Ferravgn[i] + n3avg[k]*math.log( ydata[i][2] )

            aFerravgn[i] = abs(Ferravgn[i])/abs(lydata[i][0])
        #-------------
        aFerrmeanavgn[k] = numpy.mean(aFerravgn)

        # next, update time average of n valaues in population (n1ta[k], etc.)
        # for generations = k > 800 up to total NGEN
        #============================================
        aFerrta = aFerrta + aFerrmeanavgn[k]/NGEN
        if (k > 800):
            n1ta = n1ta + n1avg[k]/(NGEN-800)
            n2ta = n2ta + n2avg[k]/(NGEN-800)
            n3ta = n3ta + n3avg[k]/(NGEN-800)
            n4ta = n4ta + n4avg[k]/(NGEN-800)
            n5ta = n5ta + n5avg[k]/(NGEN-800)

        # compare aFerrmeanavgn[k] to previous minimum value and save
        # it and corresponding n(i) values if the value for this generation k
        #============================================
        if (aFerrmeanavgn[k] < aFerrmeanavgnMin):
            aFerrmeanavgnMin = aFerrmeanavgn[k]
            n1min = n1avg[k]
            n2min = n2avg[k]
            n3min = n3avg[k]
            n4min = n4avg[k]
            n5min = n5avg[k]

        #print('avg n1-n4:', n1avg[k], n2avg[k], n3avg[k], n4avg[k], aFerrmeana
        #print ('kvalue =', k)
        '''end of evolution loop'''
        # ---------------------------------------------------------------------
        # ---------------------------------------------------------------------

    # ---------------------------------------------------------------------
    #final print and plot of results
    # ---------------------------------------------------------------------
    print('Initial Values:', n1i, n2i, n3i)
    print('ENDING: pop. avg n1-n3,aFerrmean:', n1avg[k], n2avg[k], n3avg[k], aF
    print('MINUMUM:  avg n1-n3,aFerrmeanMin:', n1min, n2min, n3min, aFerrmeanav
    print('TIME AVG:    avg n1-n3,aFerrmean:', n1ta, n2ta, n3ta, aFerrta)

    #SETTING UP PLOTS

    # =====================================================================
    # Change to original code. Calculating q" predicted and RMSE with
    # respect to the data q"

    # Initialize values
    qpppred =  [[0.0]]
    qppdata =  [[0.0]]
    for i in range(ND-1):
        qpppred.append([0.0])
        qppdata.append([0.0])
    # Calculate predicted and data values to plot
    for i in range(ND):
        qpppred[i] = n1min*(ydata[i][1]**n2min) * ((ydata[i][2])**n3min)
```

```
        qppdata[i] = ydata[i][0]
    # Calculationg RMS btw q"data and q"pred
    for i in range(ND):
        rms_dev[i] = (numpy.array(qppdata[i]) - numpy.array(qpppred[i]))**2
    rms_dev = numpy.sqrt(numpy.sum(rms_dev) / ND)
    print('RMS error: ', rms_dev)

    # ================================================================
    # Change to original code. After calculating n1min, n2min, n3min,
    # and rms_dev, create a dictionary to store these values
    iteration_result = {
        'Set': len(results) + 1,
        'n1i': n1i,
        'n2i': n2i,
        'n3i': n3i,
        'n4i': n4i,
        'n5i': n5i,
        'n1min': n1min,
        'n2min': n2min,
        'n3min': n3min,
        'n4min': n4min,
        'n5min': n5min,
        'p': perturbation,
        'NGEN': NGEN,
        'rms_dev': rms_dev,
        'aFerrmeanavgnMin': aFerrmeanavgnMin,
        # Add more values as needed
    }

    results.append(iteration_result)  # Append the summary dictionary to the li
    #================================================================================

    # constants evolution plots
    # x axis values are generation number
    # corresponding y axis values are mean absolute population error aFerrmeana
    # plotting the points

    plt.rcParams.update({'font.size': 18})

    # aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of org
    # computed using the mean n values
    plt.plot(gen, aFerrmeanavgn)
    plt.plot(gen, n1avg)
    plt.plot(gen, n2avg)
    plt.plot(gen, n3avg)
    plt.legend(['aFerrmeanavgn', 'n1 avg', 'n2 avg', 'n3 avg'], loc='lower left
    #plt.plot(gen, n4avg)
    #plt.plot(gen, n5avg)
    #plt.legend(['aFerrmeanavgn', 'n1 avg', 'n2 avg', 'n3 avg', 'n4 avg', 'n5 a

    # naming the x axis
    plt.xlabel('generation')
    # naming the y axis
    plt.ylabel('constants and error')
    plt.loglog()
    plt.yticks([0.01,0.1,1.0,10])
    plt.xticks([1,10,100,1000,10000])
    plt.show()

    '''CALCULATE PREDICTED VALUES AND RETRIEVE DATA VALUES TO PLOT'''
```

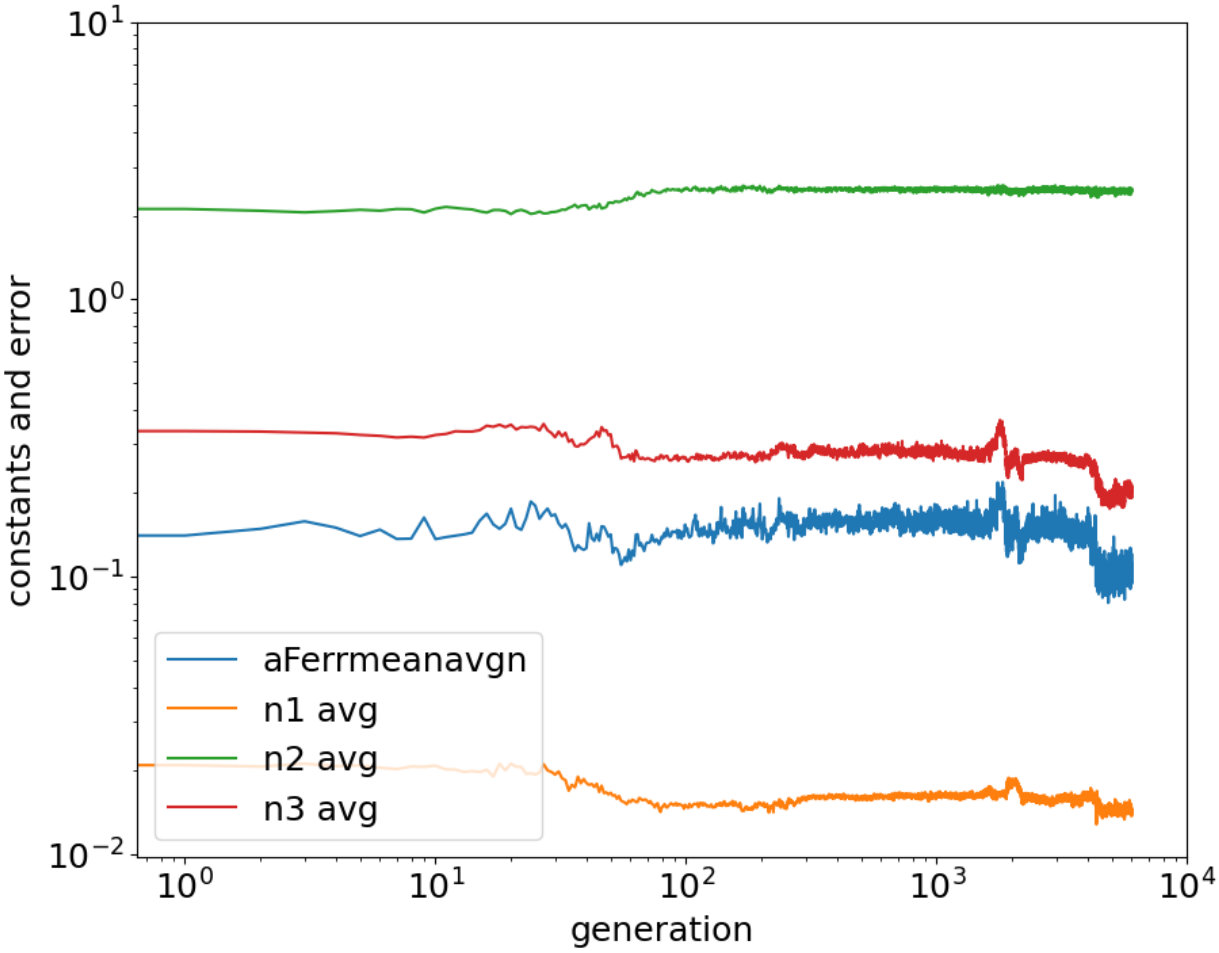
```
    # ================================================================================
    # Change to original code. Adding perfect algorithm prediction reference l
    # and +/-10% uncertainty boundaries

    x_values = numpy.logspace(0, 3, 100)   # Adjust the range of x values as n
    y_values = x_values                    # y = k*x
    y_values1 = x_values + (0.1 * x_values) # y = k*x + 0.1*x (+10% uncertainty
    y_values2 = x_values - (0.1 * x_values) # y = k*x - 0.1*x (-10% uncertainty
    # Plot reference equations
    plt.plot(x_values, y_values1, color='green', linestyle='--', label='y = k*x
    plt.plot(x_values, y_values2, color='green', linestyle='--', label='y = k*x
    plt.plot(x_values, y_values, color='red', linestyle='--', label='y = k*x')
    # Plot Heat Flux data vs. predicted
    plt.scatter(qppdata, qpppred, label='Data', color='blue', alpha=0.7)
    plt.title('Genetic Algorithm')
    plt.xlabel('measured heat flux (W/cm^2)')
    plt.ylabel('predicted heat flux (W/cm^2)')
    plt.loglog()
    plt.xlim(xmax = 1000, xmin = 10)
    plt.ylim(ymax = 1000, ymin = 10)

    plt.show()
    # ================================================================================

# ================================================================
# Change to original code. Adding results table for each iteration

# Table of Results.
print("Results Table 1:")
print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'n1min':<10}{'n2min':<10}
for result in results:
    print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{res
# ================================================================

# end CodeP1.2F23
```

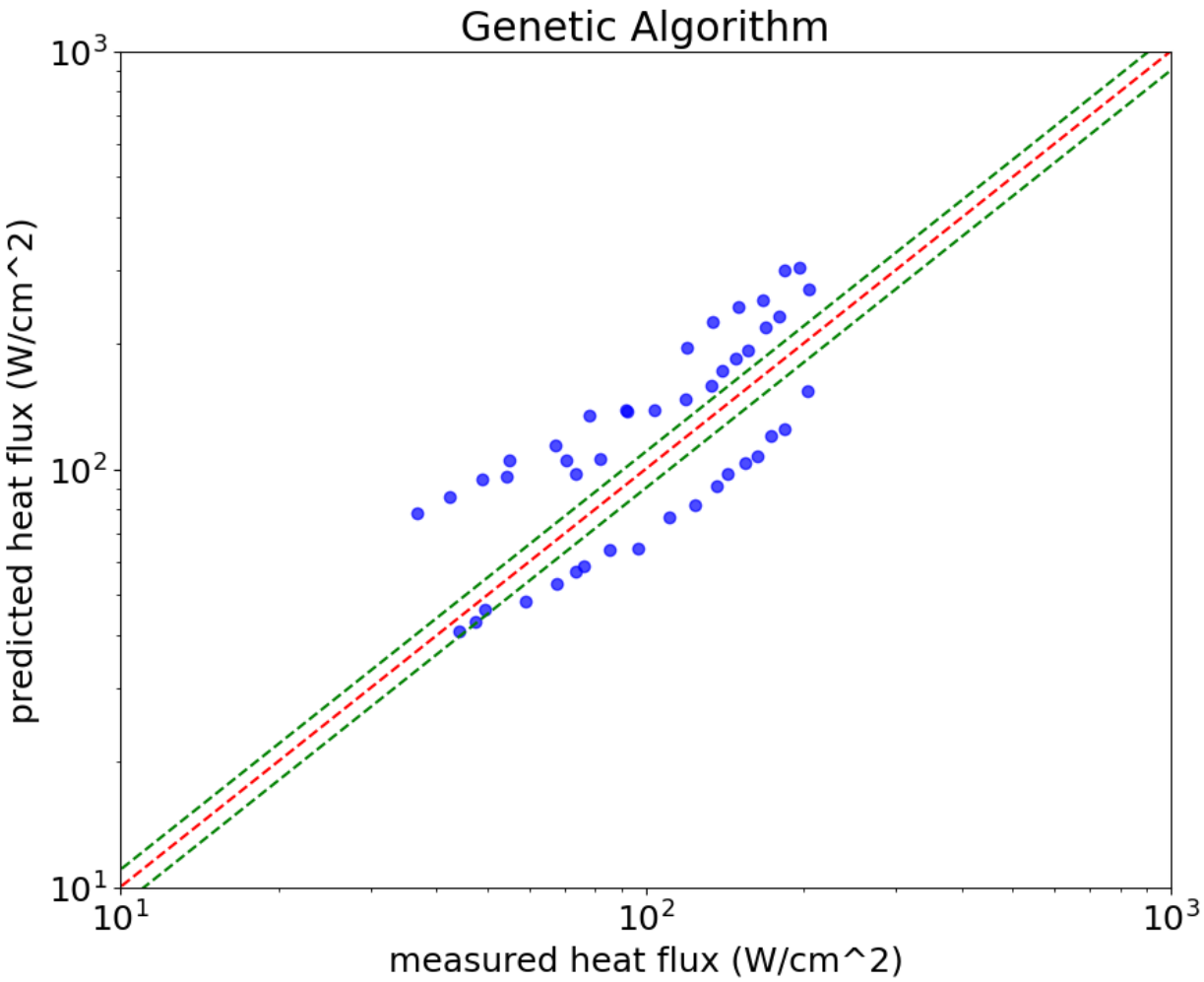
```
Initial Values: 0.00027 4.0 0.063
ENDING: pop. avg n1-n3,aFerrmean: 0.0004008858563538421 3.3582905551015045 0.0
58732771582190674 0.03212142573422008
MINUMUM:  avg n1-n3,aFerrmeanMin: 0.0006545847207557272 3.2377155389236623 0.0
6266097827573051 0.02980634985551748
TIME AVG:    avg n1-n3,aFerrmean: 0.00045554639002278797 3.3434233933523805 0.
06241900748748954 0.03613945450513338
RMS error:  30.280727632655182
```


constants and error
generation
aFerrmeanavgn
n1 avg
n2 avg
n3 avg

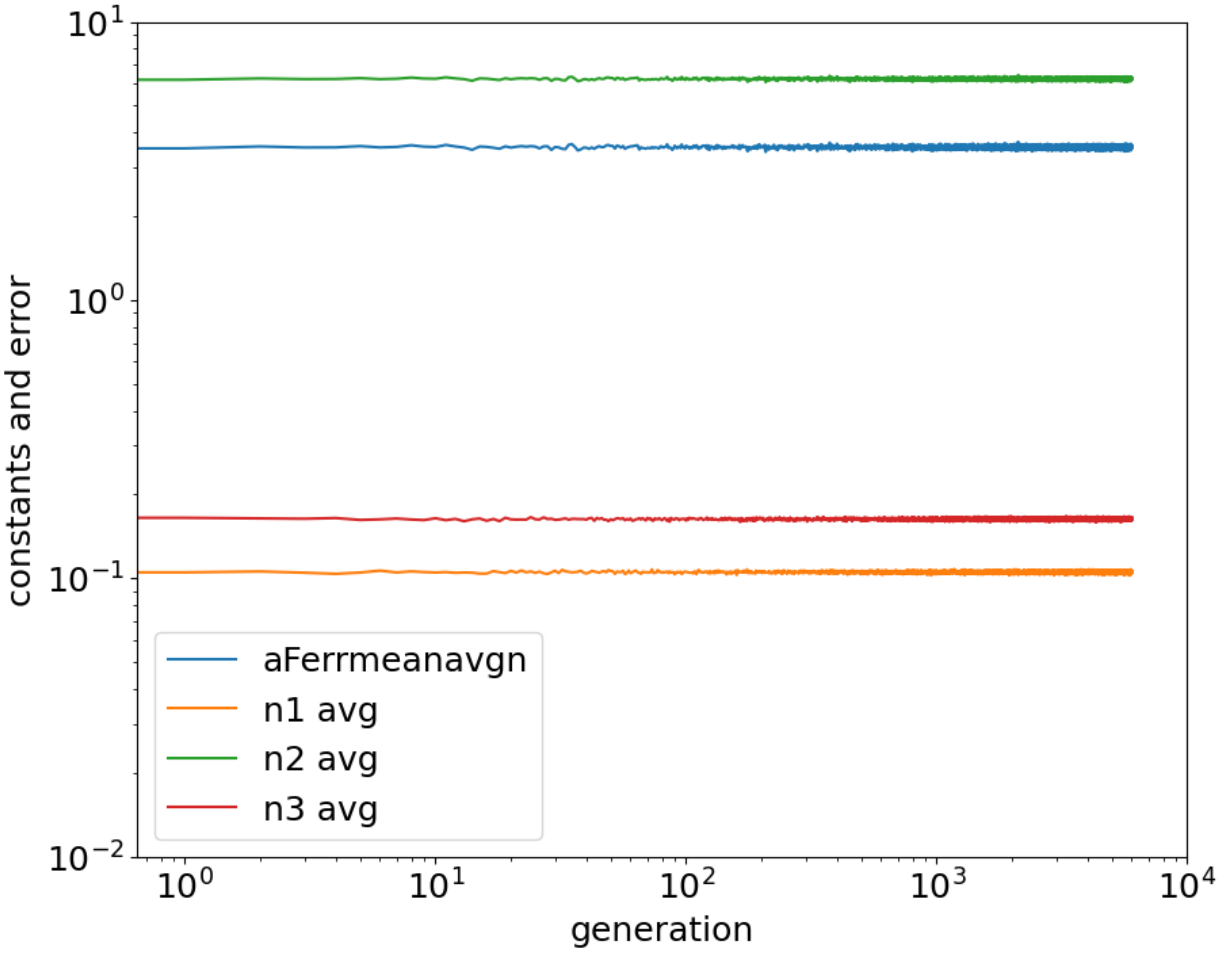


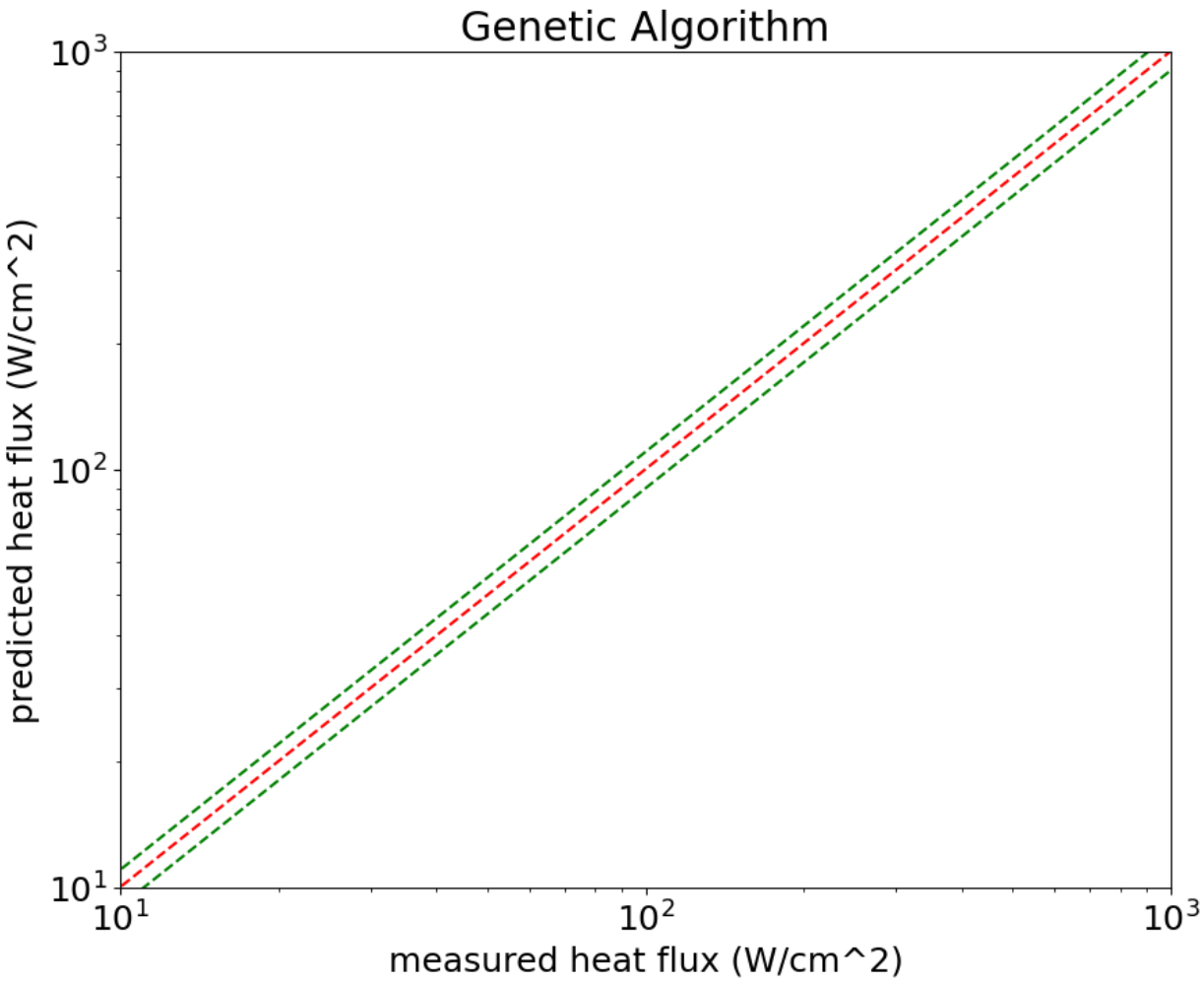

```
Initial Values: 0.02091 2.125 0.334
ENDING: pop. avg n1-n3,aFerrmean: 0.014235468965031955 2.463979037560665 0.200
4494847063887 0.10285673426064676
MINUMUM:  avg n1-n3,aFerrmeanMin: 0.01475288102425763 2.3973779508307156 0.179
89668713807672 0.08042706074814213
TIME AVG:    avg n1-n3,aFerrmean: 0.015616370456776127 2.4666994738637844 0.24
763689433237374 0.13711280287327895
RMS error:  50.82690729791071
```


constants and error
generation
aFerrmeanavgn
n1 avg
n2 avg
n3 avg

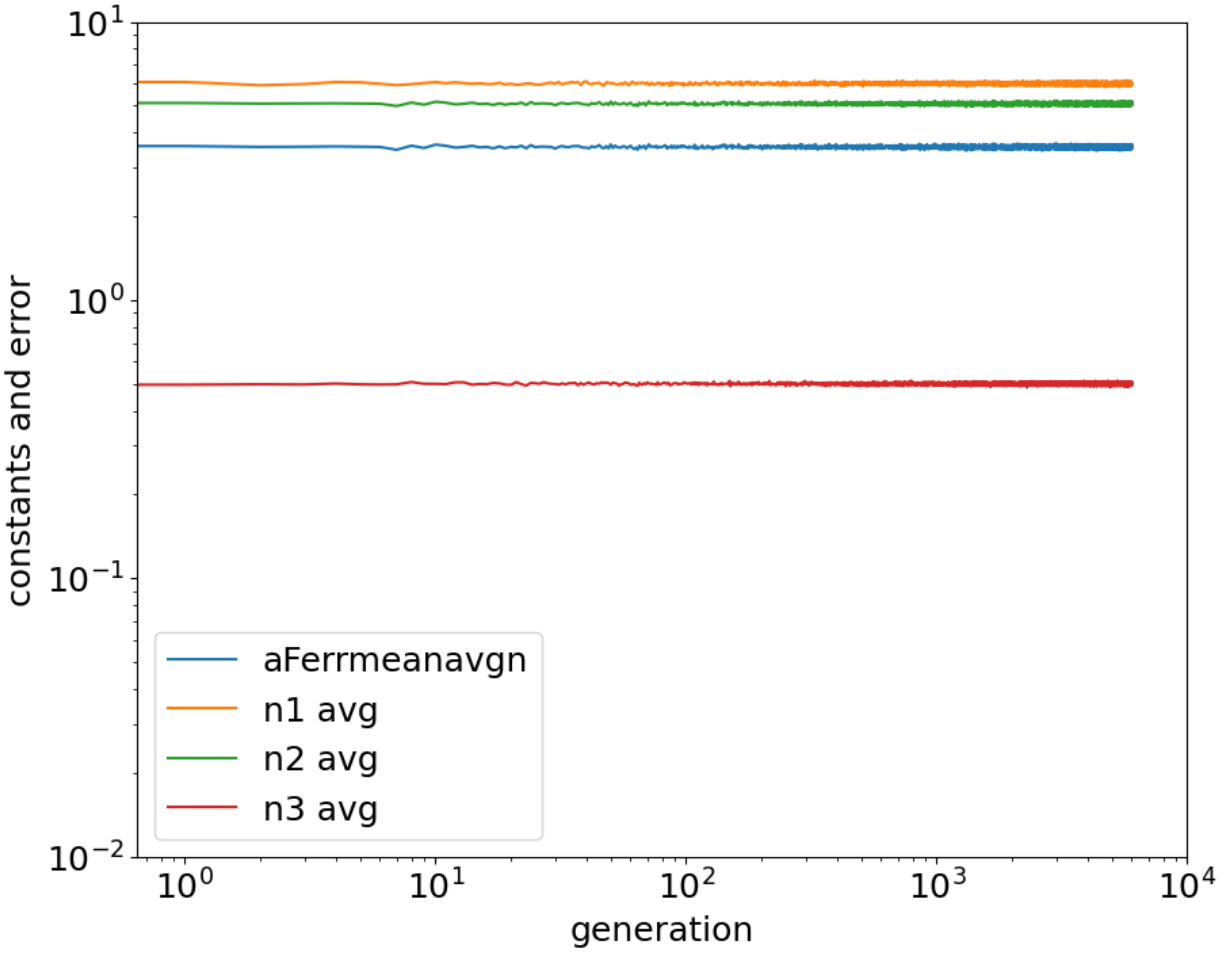


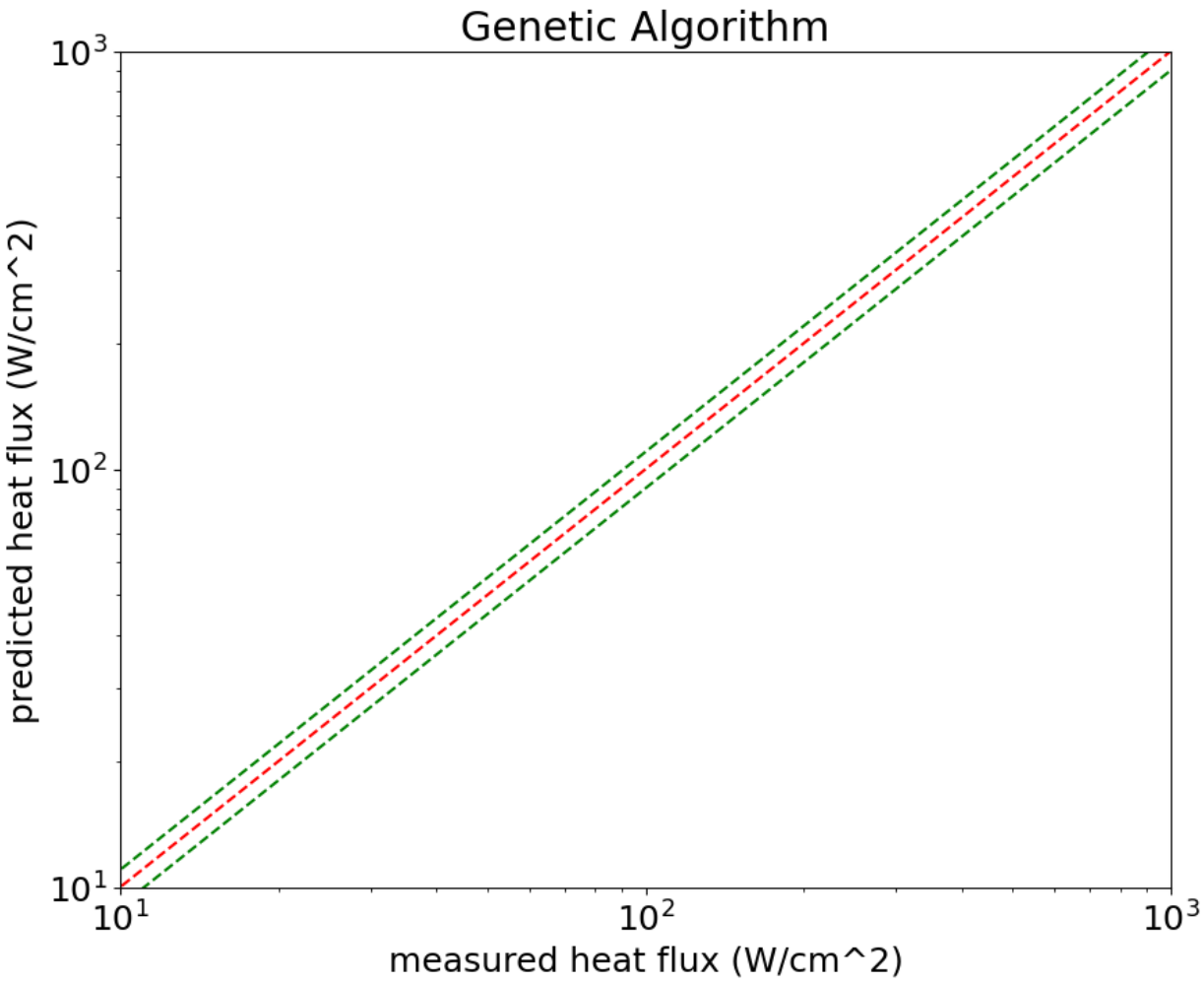

```
Initial Values: 0.105 6.2 0.163
ENDING: pop. avg n1-n3,aFerrmean: 0.10647768657396967 6.215696462246398 0.1659
3741298744905 3.5317899699216455
MINUMUM:  avg n1-n3,aFerrmeanMin: 0.10496588277320355 6.062239274109355 0.1631
432271718468 3.4048487992780734
TIME AVG:    avg n1-n3,aFerrmean: 0.10508796941061094 6.2273343662200045 0.162
98900090141802 3.5389732310528332
RMS error:  1448354020.8844028
```


constants and error
generation
aFerrmeanavgn
n1 avg
n2 avg
n3 avg




```
Initial Values: 6 5 0.5
ENDING: pop. avg n1-n3,aFerrmean: 6.0171046147458735 5.032418401034349 0.50399
36980621235 3.5145737303614415
MINUMUM:  avg n1-n3,aFerrmeanMin: 6.009287457647984 4.939371758920802 0.503222
4758898802 3.4393568533628573
TIME AVG:    avg n1-n3,aFerrmean: 5.9999450128182215 5.073557362727431 0.49988
76452358437 3.5472830896829195
RMS error:  2216184328.830863
```


constants and error
generation
aFerrmeanavgn
n1 avg
n2 avg
n3 avg




```
Results Table 1:
Set       n1i       n2i       n3i       n1min     n2min     n3min     p
NGEN      RMSE      MAE
1         0.00027   4.000     0.063     0.00065   3.24      0.063     0.09
6000      30.28     0.0298
2         0.02091   2.125     0.334     0.01475   2.40      0.180     0.09
6000      50.83     0.0804
3         0.10500   6.200     0.163     0.10497   6.06      0.163     0.09
6000      1448354020.883.4048
4         6.00000   5.000     0.500     6.00929   4.94      0.503     0.09
6000      2216184328.833.4394
```

In [ ]:

Task 1.2 (extra)

The code was generate using the provided code with the only goal to SEARCH THROUGH A LINEAR SPACE
of each of the variables (n1, n2, n3) to obtain the lowest values of RMSE and MAE. 15,625 combinations were tested. Below is an example for running this cells for 125 combinations.

In [5]:
```
'''>>>>> start CodeP1.1F23
    V.P. Carey ME249, Fall 2023'''

# version 3 print function
from __future__ import print_function
# seed the pseudorandom number generator
from random import random
from random import seed
# seed random number generator
seed(1)

#import math and numpy packages
import math
import numpy as np
import numpy as numpy

%matplotlib inline
# importing the required module
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 8] # for square canvas

#import copy
from copy import copy, deepcopy

#create arrays
ydata = []
lydata = []

#Parameters for Evolution Loop
#set data parameters
ND = 45       #number of data vectors in array
DI = 5        #number of data items in vector
NS = 45       #total number of DNA strands

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
# this is an array that is essentially a list of lists

#assembling data array
#store array where rows are data vectors
#[heat flux, superheat, gravity, surface tension parameter, pressure]

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, 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])

''' need deepcopy to create an array of the same size as ydata,
#   since this array is a list(rows) of lists (column entries)  '''
lydata = deepcopy(ydata)  # create array to store ln of data values

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
#now store log values for data
for j in range(DI):
    for i in range(ND):
        lydata[i][j]=math.log(ydata[i][j]+0.000000000010)

#OK now have stored array of log values for data
```

To automate finding the initial values (n1i, n2i, and n3i) for the lowest RMS deviation percent, we implement
a search algorithm. The approach is to use a grid search where it systematically explore different

combinations of initial values within certain ranges and then select the combination that results in an
RMS deviation closest to the desire RMS deviation.

Added section:

- Define linspace for each variable at a range
- For Loop for running through the multiple sets
- Calculation of RMSE an segregate top results
- Calculation of MAE an segregate top results
- Print both tables

In [6]:
```
# Define search ranges for n1i, n2i, and n3i
n1i_range = np.linspace(0.0001, 0.1, 5)  # Adjust the range as needed
n2i_range = np.linspace(2.0, 10.0, 5)     # Adjust the range as needed
n3i_range = np.linspace(0.001, 1.0, 5)     # Adjust the range as needed

# Initialize variables to store the best values and minimum RMS deviation
best_n1i = 0.0
best_n2i = 0.0
best_n3i = 0.0
min_rms_deviation = float('inf')  # Initialize with a large value
min_MAE = float('inf') # Initialize with a large value

# Define a list to store the top 5 results
top_results = []     # RMSE
top_results_MAE = [] # MAE

# Create an empty list to store summaries
results = []

# Iterate through the search ranges
for n1i in n1i_range:
    for n2i in n2i_range:
        for n3i in n3i_range:
            '''INITIALIZING PARAMETERS'''
            # Define a list of n values, perturbation, and NGEN to evaluate
            n_values_to_evaluate = [
                [(-1, n1i, n2i, n3i, 0.0, 0.0), 0.09, 5000]]  # Given Initial (

            for n_values, perturbation, NGEN in n_values_to_evaluate:

                n = []
                ntemp = []
                gen=[0]
                n1avg = [0.0]
                n2avg = [0.0]
                n3avg = [0.0]
                n4avg = [0.0]
                n5avg = [0.0]
                meanAFerr=[0.0]
                aFerrmeanavgn=[0.0]
                rms_dev = [0.0] # Define rms_dev

                # Set program parameters
                NGEN = NGEN      # number of generations (steps)
```

```
MFRAC = 0.5   # faction of median threshold
perturbation = perturbation # pertubation value

# here the number of data vectors equals the number of DNA stra
# they can be different if they are randomly paired to compute
for k in range(NGEN-1):
    gen.append(k+1)   # generation array stores the
    meanAFerr.append(0.0)
    aFerrmeanavgn.append(0.0)
    n1avg.append(0.0)
    n2avg.append(0.0)
    n3avg.append(0.0)
    n4avg.append(0.0)
    n5avg.append(0.0)

'''guesses for initial solution population'''
n0i, n1i, n2i, n3i, n4i, n5i = n_values

#- initialize arrays before start of evolution loop EL
#then - create array of DNA strands n[i] and ntemp[i] with dime

#i initialize array where rows are dna vectors [n0i,n1i,...n5i
n =  [[-1., n1i+0.001*random(), n2i+0.1*random(), n3i+0.0001*ra
for i in range(ND):
    n.append([-1., n1i+0.0001*random(), n2i+0.001*random(), n3i
#print (n) # uncomment command to print array so it can be chec

# store also in wtemp
ntemp = deepcopy(n)

#initialize Ferr values an dother loop parameters
#define arrays of Ferr (error) functions
#individual solution error and absoute error
Ferr =  [[0.0]]
#population average solution error and absoute error
Ferravgn =  [[0.0]]
aFerr =  [[0.0]]
aFerravgn =  [[0.0]]

#store zeros in ND genes
for i in range(ND-1):
    #individual solution error and absoute error
    Ferr.append([0.0])
    aFerr.append([0.0])
    #population average solution error and absoute error
    Ferravgn.append([0.0])
    aFerravgn.append([0.0])
    rms_dev.append([0.0])
#print (Ferr)

aFerrmeanavgnMin = 1000000000.0
# these store the  n values for minimum population average erro
n1min = 0.0
n2min = 0.0
n3min = 0.0
n4min = 0.0
n5min = 0.0
aFerrta = 0.0
# these store the time averaged n values durng from generation
n1min = 0.0
```

```
n1ta = 0.0
n2ta = 0.0
n3ta = 0.0
n4ta = 0.0
n5ta = 0.0

'''START OF EVOLUTION LOOP'''
# ---------------------------------------
# k is generation number, NGEN IS TOTAL NUMBER OF GENERATIONS (
for k in range(NGEN):

    '''In this program , the number of organisms (solutions) NS
    number of data points ND so for each generation, each solut
    data point and all the data is compared in each generation.
    that holds the solution constants is constantly changing du
    is random.'''

    '''CALCULATING ERROR (FITNESS)
    In this program, the absolute error in the logrithm of the
    used to evaluate fitness.'''

    # Here we calculate error Ferr and absolute error aFerr for
    # for specified n(i), and calculate (mean aFerr) = aFerrmea
    # and (median aFerr) = aFerrmedian for the data collection
    # Note that the number data points ND equals the number of
    #=========================================================
    '''CALCULATING ERROR (FITNESS)'''
    for i in range(ND):

        Ferr[i] = n[i][0]*lydata[i][0] + math.log(n[i][1]) + n[
        Ferr[i] = Ferr[i] + n[i][3]*math.log( ydata[i][2] )
        aFerr[i] = abs(Ferr[i])/abs(lydata[i][0])  #- absolute
    #-------------
    aFerrmean = numpy.mean(aFerr) #mean error for population fo
    meanAFerr[k]=aFerrmean  #store aFerrmean for this generatio
    aFerrmedian = numpy.median(aFerr) #median error for popula

    '''SELECTION'''
    #pick survivors
    #[2] calculate survival cutoff, set number kept = nkeep = (
    #=============================================
    clim = MFRAC*aFerrmedian  #cut off limit is a fraction/mult
    nkeep = 0

    # now check each organism/solution to see if aFerr is less
    #if yes, store n for next generation population in ntemp, a
    #and number of new offspring = NS-nkeep
    #=============================================
    for j in range(NS):  # NS Ferr values, one for each solutio
        if (aFerr[j] < clim):
            nkeep = nkeep + 1
            #ntemp[nkeep][0] = n[j][0] = -1 so it is unchanged,
            ntemp[nkeep-1][1] = n[j][1];
            ntemp[nkeep-1][2] = n[j][2];
            ntemp[nkeep-1][3] = n[j][3];
            ntemp[nkeep-1][4] = n[j][4];
            ntemp[nkeep-1][5] = n[j][5];
    #now have survivors in leading entries in list of ntemp vec
    #compute number to be added by mating
    nnew = NS - nkeep
```

```
            '''MATING'''
            #[4] for nnew new organisms/solutions,
            # randomly pick two survivors, randomly pick DNA (n) from p
            #=========================================================
            for j in range(nnew):
                # pick two survivors randomly
                nmate1 = numpy.random.randint(low=0, high=nkeep+1)
                nmate2 = numpy.random.randint(low=0, high=nkeep+1)

                #then randomly pick DNA from parents for offspring

                '''here, do not change property ntemp[nkeep+j+1][0], it
                #if (numpy.random.rand() < 0.5)
                #    ntemp[nkeep+j+1][0] = n[nmate1][0]
                #else
                #    ntemp[nkeep+j+1][0] = n[nmate2][0]

                if (numpy.random.rand() < 0.5):
                    ntemp[nkeep+j+1][1] = n[nmate1][1]*(1.+perturbatio
                else:
                    ntemp[nkeep+j+1][1] = n[nmate2][1]*(1.+perturbatio

                if (numpy.random.rand() < 0.5):
                    ntemp[nkeep+j+1][2] = n[nmate1][2]*(1.+perturbatio
                else:
                    ntemp[nkeep+j+1][2] = n[nmate2][2]*(1.+perturbatio

                if (numpy.random.rand() < 0.5):
                    ntemp[nkeep+j+1][3] = n[nmate1][3]*(1.+perturbatio
                else:
                    ntemp[nkeep+j+1][3] = n[nmate2][3]*(1.+perturbatio
                '''
                if (numpy.random.rand() < 0.5):
                    ntemp[nkeep+j+1][4] = n[nmate1][4]*(1.+perturbatio
                else:
                    ntemp[nkeep+j+1][4] = n[nmate2][4]*(1.+perturbatio

                if (numpy.random.rand() < 0.5):
                    ntemp[nkeep+j+1][5] = n[nmate1][5]*(1.+perturbatio
                else:
                    ntemp[nkeep+j+1][5] = n[nmate2][5]*(1.+perturbatio
                '''
            #=============================================
            n = deepcopy(ntemp)   # save ntemp as n for use in next ge

            '''AVERAGING OVER POPULATION AND OVER TIME, FINDING MINIMU
            # [6] calculate n1avg[k], etc., which are average n values
            # at this generation k
            #=============================================
            #initialize average n's to zero and sum contribution of ea
            n1avg[k] = 0.0;
            n2avg[k] = 0.0;
            n3avg[k] = 0.0;
            n4avg[k] = 0.0;
            n5avg[k] = 0.0;
            for j in range(NS):
                n1avg[k] = n1avg[k] + n[j][1]/NS;
                n2avg[k] = n2avg[k] + n[j][2]/NS;
                n3avg[k] = n3avg[k] + n[j][3]/NS;
```

```
            n4avg[k] = n4avg[k] + n[j][4]/NS;
            n5avg[k] = n5avg[k] + n[j][5]/NS;

        # Here we compute aFerravgn[i] = absolute Ferr of logrithm
        # for this solutions generation k
        # aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the p
        #
        #============================================
        ''' CALCULATING MEAN ERROR FOR POPULATION'''
        for i in range(ND):
            Ferravgn[i] = -1.*lydata[i][0] + math.log(n1avg[k]) + n
            Ferravgn[i] = Ferravgn[i] + n3avg[k]*math.log( ydata[i]

            #aFerravgn[i] = abs(Ferr[i])/abs(lydata[i][0])
            aFerravgn[i] = abs(Ferravgn[i])/abs(lydata[i][0])
        #-------------
        aFerrmeanavgn[k] = numpy.mean(aFerravgn)

        # next, update time average of n valaues in population (n1
        # for generations = k > 800 up to total NGEN
        #============================================
        aFerrta = aFerrta + aFerrmeanavgn[k]/NGEN
        if (k > 800):
            n1ta = n1ta + n1avg[k]/(NGEN-800)
            n2ta = n2ta + n2avg[k]/(NGEN-800)
            n3ta = n3ta + n3avg[k]/(NGEN-800)
            n4ta = n4ta + n4avg[k]/(NGEN-800)
            n5ta = n5ta + n5avg[k]/(NGEN-800)

        # compare aFerrmeanavgn[k] to previous minimum value and sa
        # it and corresponding n(i) values if the value for this ge
        #============================================
        if (aFerrmeanavgn[k] < aFerrmeanavgnMin):
            aFerrmeanavgnMin = aFerrmeanavgn[k]
            n1min = n1avg[k]
            n2min = n2avg[k]
            n3min = n3avg[k]
            n4min = n4avg[k]
            n5min = n5avg[k]

        #print('avg n1-n4:', n1avg[k], n2avg[k], n3avg[k], n4avg[k]
        #print ('kvalue =', k)
        '''end of evolution loop'''
        # ------------------------------------------------------------
        # ------------------------------------------------------------

    # ------------------------------------------------------------
    #final print and plot of results
    # ------------------------------------------------------------
    print('Initial Values:', n1i, n2i, n3i)

    #SETTING UP PLOTS
    #========
    #initialize values
    qpppred =  [[0.0]]
    qppdata =  [[0.0]]
    for i in range(ND-1):
        qpppred.append([0.0])
        qppdata.append([0.0])
    #calculate predicted and data values to plot
```

```
                for i in range(ND):
                    qpppred[i] = n1min*(ydata[i][1]**n2min) * ((ydata[i][2])**n
                    qppdata[i] = ydata[i][0]
                #Calculationg RMS btw qppdata and qpppred
                for i in range(ND):
                    rms_dev[i] = (numpy.array(qppdata[i]) - numpy.array(qpppred
                rms_dev = numpy.sqrt(numpy.sum(rms_dev) / ND)
                print('RMS error: ', rms_dev)
                print('MAE error: ', aFerrmeanavgnMin)

                # After calculating n1min, n2min, n3min, and rms_dev, create a
                iteration_result = {
                    'Set': len(results) + 1,
                    'n1i': n1i,
                    'n2i': n2i,
                    'n3i': n3i,
                    'n4i': n4i,
                    'n5i': n5i,
                    'n1min': n1min,
                    'n2min': n2min,
                    'n3min': n3min,
                    'n4min': n4min,
                    'n5min': n5min,
                    'p': perturbation,
                    'NGEN': NGEN,
                    'rms_dev': rms_dev,
                    'aFerrmeanavgnMin': aFerrmeanavgnMin,
                    # Add more values as needed
                }

                results.append(iteration_result)  # Append the summary dictiona

                # Check if the current combination results in a lower RMS devia
                if abs(rms_dev - 1.0) < abs(min_rms_deviation - 1.0):
                    # Update the best values
                    best_n1i = n1i
                    best_n2i = n2i
                    best_n3i = n3i
                    min_rms_deviation = rms_dev
                # Check if the current combination results in a lower MAE devia
                if abs(aFerrmeanavgnMin - 1.0) < abs(min_MAE - 1.0):
                    # Update the best values
                    best_n1i = n1i
                    best_n2i = n2i
                    best_n3i = n3i
                    min_MAE = aFerrmeanavgnMin

                # Check if this result is among the top 5 results with the lowe
                if len(top_results) < 5:
                    top_results.append(iteration_result)
                    top_results.sort(key=lambda x: x['rms_dev'])  # Sort the to
                elif rms_dev < top_results[-1]['rms_dev']:
                    top_results.pop()  # Remove the result with the highest RMS
                    top_results.append(iteration_result)
                    top_results.sort(key=lambda x: x['rms_dev'])  # Sort the to

                # Check if this result is among the top 5 results with the lowe
                if len(top_results_MAE) < 5:
                    top_results_MAE.append(iteration_result)
                    top_results_MAE.sort(key=lambda x: x['aFerrmeanavgnMin'])
```

```
                elif aFerrmeanavgnMin < top_results_MAE[-1]['aFerrmeanavgnMin']
                    top_results_MAE.pop()  # Remove the result with the highest
                    top_results_MAE.append(iteration_result)
                    top_results_MAE.sort(key=lambda x: x['aFerrmeanavgnMin'])

# ====================================================
# Long version Table (adds info of minimum n values)
# ====================================================
# # Print the best values and minimum RMS deviation
# print("Best n1i:", best_n1i)
# print("Best n2i:", best_n2i)
# print("Best n3i:", best_n3i)
# print("Minimum RMS Deviation:", min_rms_deviation)

# # Print the top 5 results
# print("\nTop 5 Results:")
# print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'n1min':<10}{'n2min':<10
# for result in top_results:
#     print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{r

# # Print the top 5 results
# print("\nTop 5 Results:")
# print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'n1min':<10}{'n2min':<10
# for result in top_results_MAE:
#     print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{r

# ====================================================
# Short version Table (ommits info of minimum n values)
# ====================================================
# Print the best values and minimum RMS deviation
print("Best n1i:", best_n1i)
print("Best n2i:", best_n2i)
print("Best n3i:", best_n3i)
print("Minimum RMS Deviation:", min_rms_deviation)

# Print the top 5 results
print("\nTop 5 Results:")
print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'p':<10}{'NGEN':<10}{'RMSE
for result in top_results:
    print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{res

# Print the top 5 results
print("\nTop 5 Results:")
print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'p':<10}{'NGEN':<10}{'RMSE
for result in top_results_MAE:
    print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{res
```

```
Initial Values: 0.0001 2.0 0.001
RMS error:  122.99932863010284
MAE error:  1.1121503723641992
Initial Values: 0.0001 2.0 0.25075
RMS error:  120.14677616885987
MAE error:  0.7836811519025815
Initial Values: 0.0001 2.0 0.5005
RMS error:  122.48526993708272
MAE error:  1.0972667140350778
Initial Values: 0.0001 2.0 0.75025
RMS error:  60.81613867001015
MAE error:  0.14374530774372574
Initial Values: 0.0001 2.0 1.0
RMS error:  63.38352863807872
MAE error:  0.16093036041050732
Initial Values: 0.0001 4.0 0.001
RMS error:  36.00226745013371
MAE error:  0.04529242537213741
Initial Values: 0.0001 4.0 0.25075
RMS error:  44.100717326821176
MAE error:  0.03670329692573617
Initial Values: 0.0001 4.0 0.5005
RMS error:  304.0217363911383
MAE error:  0.23182800044450874
Initial Values: 0.0001 4.0 0.75025
RMS error:  626.6296723881045
MAE error:  0.3651624056505042
Initial Values: 0.0001 4.0 1.0
RMS error:  4495.771467628089
MAE error:  0.5666305896963472
Initial Values: 0.0001 6.0 0.001
RMS error:  3664224.041689063
MAE error:  2.1071562353803412
Initial Values: 0.0001 6.0 0.25075
RMS error:  6686310.788694005
MAE error:  2.2131812551533856
Initial Values: 0.0001 6.0 0.5005
RMS error:  14424538.079820465
MAE error:  2.296309584341156
Initial Values: 0.0001 6.0 0.75025
RMS error:  29729174.028897993
MAE error:  2.3578509998698607
Initial Values: 0.0001 6.0 1.0
RMS error:  77368.13836408796
MAE error:  0.9443149893823927
Initial Values: 0.0001 8.0 0.001
RMS error:  6904229539.584641
MAE error:  3.6401773999542413
Initial Values: 0.0001 8.0 0.25075
RMS error:  9900828268.831926
MAE error:  3.7106437825690994
Initial Values: 0.0001 8.0 0.5005
RMS error:  25536450189.38909
MAE error:  3.8353052760139406
Initial Values: 0.0001 8.0 0.75025
RMS error:  52611548122.262085
MAE error:  3.900314231090663
Initial Values: 0.0001 8.0 1.0
RMS error:  25653688311.239826
MAE error:  3.6410590710361177
```

```
Initial Values: 0.0001 10.0 0.001
RMS error:  13485592539515.717
MAE error:  5.176448744452791
Initial Values: 0.0001 10.0 0.25075
RMS error:  21137133573185.617
MAE error:  5.278244625626821
Initial Values: 0.0001 10.0 0.5005
RMS error:  27906259317836.258
MAE error:  5.26675608319727
Initial Values: 0.0001 10.0 0.75025
RMS error:  93310578171463.11
MAE error:  5.430013687301563
Initial Values: 0.0001 10.0 1.0
RMS error:  117294952452748.69
MAE error:  5.38840718032519
Initial Values: 0.025075 2.0 0.001
RMS error:  21.62595323714515
MAE error:  0.034943042622842646
Initial Values: 0.025075 2.0 0.25075
RMS error:  21.839730835676754
MAE error:  0.035570418813070485
Initial Values: 0.025075 2.0 0.5005
RMS error:  60.40204485478221
MAE error:  0.14256014685517077
Initial Values: 0.025075 2.0 0.75025
RMS error:  42.11409394345727
MAE error:  0.07958707923694909
Initial Values: 0.025075 2.0 1.0
RMS error:  18.893849164245967
MAE error:  0.03545865919250508
Initial Values: 0.025075 4.0 0.001
RMS error:  83806.77655589355
MAE error:  1.3766261871168124
Initial Values: 0.025075 4.0 0.25075
RMS error:  113164.0772998366
MAE error:  1.4086649792695407
Initial Values: 0.025075 4.0 0.5005
RMS error:  212844.03488883394
MAE error:  1.4533498748037375
Initial Values: 0.025075 4.0 0.75025
RMS error:  17439.8015713391
MAE error:  0.7820991697819314
Initial Values: 0.025075 4.0 1.0
RMS error:  86742.28055524337
MAE error:  1.0480120567852094
Initial Values: 0.025075 6.0 0.001
RMS error:  140274747.36022356
MAE error:  2.905042876412726
Initial Values: 0.025075 6.0 0.25075
RMS error:  204024726.632964
MAE error:  2.963606357322868
Initial Values: 0.025075 6.0 0.5005
RMS error:  369555689.90866137
MAE error:  3.004791643990991
Initial Values: 0.025075 6.0 0.75025
RMS error:  702795907.8221807
MAE error:  3.0427243402079305
Initial Values: 0.025075 6.0 1.0
RMS error:  1321216599.7257519
MAE error:  3.076507179713151
```

```
Initial Values: 0.025075 8.0 0.001
RMS error:  386968685876.41956
MAE error:  4.5205356910826975
Initial Values: 0.025075 8.0 0.25075
RMS error:  478257993509.41614
MAE error:  4.558357710435979
Initial Values: 0.025075 8.0 0.5005
RMS error:  809064558573.0035
MAE error:  4.591907649611913
Initial Values: 0.025075 8.0 0.75025
RMS error:  1513893174318.8296
MAE error:  4.6345187828746965
Initial Values: 0.025075 8.0 1.0
RMS error:  2400385389531.011
MAE error:  4.636211336202667
Initial Values: 0.025075 10.0 0.001
RMS error:  614322257380472.5
MAE error:  6.015647995512594
Initial Values: 0.025075 10.0 0.25075
RMS error:  1002053406943801.9
MAE error:  6.121702354459113
Initial Values: 0.025075 10.0 0.5005
RMS error:  1271834964302553.5
MAE error:  6.103339004256678
Initial Values: 0.025075 10.0 0.75025
RMS error:  2402197870640723.5
MAE error:  6.145441667185289
Initial Values: 0.025075 10.0 1.0
RMS error:  4271125639184601.5
MAE error:  6.179229262641569
Initial Values: 0.050050000000000004 2.0 0.001
RMS error:  21.65566602328406
MAE error:  0.035644225767498865
Initial Values: 0.050050000000000004 2.0 0.25075
RMS error:  21.052818626132993
MAE error:  0.039611823016853356
Initial Values: 0.050050000000000004 2.0 0.5005
RMS error:  122.28373130560884
MAE error:  0.1773776965700723
Initial Values: 0.050050000000000004 2.0 0.75025
RMS error:  268.6887889058976
MAE error:  0.33866541257355987
Initial Values: 0.050050000000000004 2.0 1.0
RMS error:  55.36781436146778
MAE error:  0.11969807972121739
Initial Values: 0.050050000000000004 4.0 0.001
RMS error:  124796.6677996933
MAE error:  1.4668807581963794
Initial Values: 0.050050000000000004 4.0 0.25075
RMS error:  190591.67517874806
MAE error:  1.5250051535550813
Initial Values: 0.050050000000000004 4.0 0.5005
RMS error:  338056.62995984487
MAE error:  1.5553154333522539
Initial Values: 0.050050000000000004 4.0 0.75025
RMS error:  12941.081577060262
MAE error:  0.7127897362533809
Initial Values: 0.050050000000000004 4.0 1.0
RMS error:  62821.173794098504
MAE error:  0.9757864268436152
```

```
Initial Values: 0.050050000000000004 6.0 0.001
RMS error:  281572490.5509596
MAE error:  3.057124945750242
Initial Values: 0.050050000000000004 6.0 0.25075
RMS error:  425355155.1635738
MAE error:  3.1253580833276855
Initial Values: 0.050050000000000004 6.0 0.5005
RMS error:  584728297.2927467
MAE error:  3.1081590013070817
Initial Values: 0.050050000000000004 6.0 0.75025
RMS error:  1664962748.0226142
MAE error:  3.2324901908793886
Initial Values: 0.050050000000000004 6.0 1.0
RMS error:  2843989413.2907257
MAE error:  3.251987516254161
Initial Values: 0.050050000000000004 8.0 0.001
RMS error:  539833302212.27576
MAE error:  4.599303110873595
Initial Values: 0.050050000000000004 8.0 0.25075
RMS error:  806477471982.4878
MAE error:  4.675511189937672
Initial Values: 0.050050000000000004 8.0 0.5005
RMS error:  1631379903890.2966
MAE error:  4.745666474159756
Initial Values: 0.050050000000000004 8.0 0.75025
RMS error:  2167257663100.4001
MAE error:  4.7147593308038145
Initial Values: 0.050050000000000004 8.0 1.0
RMS error:  5767843149117.215
MAE error:  4.8285315078081394
Initial Values: 0.050050000000000004 10.0 0.001
RMS error:  1245075332788301.8
MAE error:  6.169797967742606
Initial Values: 0.050050000000000004 10.0 0.25075
RMS error:  2006926700890845.8
MAE error:  6.272917152876441
Initial Values: 0.050050000000000004 10.0 0.5005
RMS error:  2510383573477969.5
MAE error:  6.248682787972823
Initial Values: 0.050050000000000004 10.0 0.75025
RMS error:  6382871170345821.0
MAE error:  6.356731535832527
Initial Values: 0.050050000000000004 10.0 1.0
RMS error:  7739187692965325.0
MAE error:  6.311626614624055
Initial Values: 0.07502500000000001 2.0 0.001
RMS error:  21.734677917731197
MAE error:  0.0376752184903743
Initial Values: 0.07502500000000001 2.0 0.25075
RMS error:  19.88850873428518
MAE error:  0.03905280609039951
Initial Values: 0.07502500000000001 2.0 0.5005
RMS error:  143.61609715259328
MAE error:  0.18824379636864444
Initial Values: 0.07502500000000001 2.0 0.75025
RMS error:  135.49320126392348
MAE error:  0.23757782045769477
Initial Values: 0.07502500000000001 2.0 1.0
RMS error:  1998.731668293242
MAE error:  0.5349409108771987
```

```
Initial Values: 0.07502500000000001 4.0 0.001
RMS error:  214578.01277109847
MAE error:  1.584214130921613
Initial Values: 0.07502500000000001 4.0 0.25075
RMS error:  323966.10184303054
MAE error:  1.6393207439834165
Initial Values: 0.07502500000000001 4.0 0.5005
RMS error:  612790.3419100995
MAE error:  1.6829369543878712
Initial Values: 0.07502500000000001 4.0 0.75025
RMS error:  17532.391138946306
MAE error:  0.7821995337316053
Initial Values: 0.07502500000000001 4.0 1.0
RMS error:  180760.64195018017
MAE error:  1.1546369765347009
Initial Values: 0.07502500000000001 6.0 0.001
RMS error:  456359425.91847825
MAE error:  3.1615930549881024
Initial Values: 0.07502500000000001 6.0 0.25075
RMS error:  649077460.9550357
MAE error:  3.2170736010649224
Initial Values: 0.07502500000000001 6.0 0.5005
RMS error:  1152986270.1433012
MAE error:  3.252676832170501
Initial Values: 0.07502500000000001 6.0 0.75025
RMS error:  2466352853.256124
MAE error:  3.3169527041750686
Initial Values: 0.07502500000000001 6.0 1.0
RMS error:  3359174194.7405577
MAE error:  3.2859132433831273
Initial Values: 0.07502500000000001 8.0 0.001
RMS error:  922306146520.723
MAE error:  4.714472247529925
Initial Values: 0.07502500000000001 8.0 0.25075
RMS error:  1284719333306.381
MAE error:  4.776840502967434
Initial Values: 0.07502500000000001 8.0 0.5005
RMS error:  1833324477133.6213
MAE error:  4.776916175247646
Initial Values: 0.07502500000000001 8.0 0.75025
RMS error:  4656877962889.554
MAE error:  4.876556294182222
Initial Values: 0.07502500000000001 8.0 1.0
RMS error:  7553221865892.698
MAE error:  4.887946569643543
Initial Values: 0.07502500000000001 10.0 0.001
RMS error:  2082288367987499.0
MAE error:  6.280609446665543
Initial Values: 0.07502500000000001 10.0 0.25075
RMS error:  2308487598429669.5
MAE error:  6.308203707562661
Initial Values: 0.07502500000000001 10.0 0.5005
RMS error:  4256959073004963.5
MAE error:  6.363450387355192
Initial Values: 0.07502500000000001 10.0 0.75025
RMS error:  8494921125963285.0
MAE error:  6.420922843083046
Initial Values: 0.07502500000000001 10.0 1.0
RMS error:  1.7545349788401404e+16
MAE error:  6.483382359754298
```

```
Initial Values: 0.1 2.0 0.001
RMS error:  21.76162646681973
MAE error:  0.04233448398499607
Initial Values: 0.1 2.0 0.25075
RMS error:  20.890602330346958
MAE error:  0.03925041001522383
Initial Values: 0.1 2.0 0.5005
RMS error:  113.73299293498528
MAE error:  0.23478283376591433
Initial Values: 0.1 2.0 0.75025
RMS error:  205.2867206885043
MAE error:  0.3371647337411716
Initial Values: 0.1 2.0 1.0
RMS error:  3297.3950706777546
MAE error:  0.5403900111291084
Initial Values: 0.1 4.0 0.001
RMS error:  302287.77973953605
MAE error:  1.658242414030236
Initial Values: 0.1 4.0 0.25075
RMS error:  456173.6181542196
MAE error:  1.7139059669693413
Initial Values: 0.1 4.0 0.5005
RMS error:  743086.2033627332
MAE error:  1.727513790680371
Initial Values: 0.1 4.0 0.75025
RMS error:  8257.887871948315
MAE error:  0.6517805797008454
Initial Values: 0.1 4.0 1.0
RMS error:  150634.3765532122
MAE error:  1.1445982128217471
Initial Values: 0.1 6.0 0.001
RMS error:  732947378.4177806
MAE error:  3.263240414930224
Initial Values: 0.1 6.0 0.25075
RMS error:  780925571.3359139
MAE error:  3.25859033418987
Initial Values: 0.1 6.0 0.5005
RMS error:  1660828698.6106434
MAE error:  3.3330454161278484
Initial Values: 0.1 6.0 0.75025
RMS error:  2316930364.3102217
MAE error:  3.3047134022990985
Initial Values: 0.1 6.0 1.0
RMS error:  5216043032.997688
MAE error:  3.378270582483516
Initial Values: 0.1 8.0 0.001
RMS error:  1097059484803.3925
MAE error:  4.754319673119184
Initial Values: 0.1 8.0 0.25075
RMS error:  1307207216522.337
MAE error:  4.783944036665904
Initial Values: 0.1 8.0 0.5005
RMS error:  2948676750290.131
MAE error:  4.87430846855026
Initial Values: 0.1 8.0 0.75025
RMS error:  4398754303100.441
MAE error:  4.873913255294684
Initial Values: 0.1 8.0 1.0
RMS error:  9700850453634.357
MAE error:  4.943634115045085
```

```
Initial Values: 0.1 10.0 0.001
RMS error:  2896180659629198.0
MAE error:  6.352157502603827
Initial Values: 0.1 10.0 0.25075
RMS error:  3940140417891771.0
MAE error:  6.420985281154052
Initial Values: 0.1 10.0 0.5005
RMS error:  4598446877797262.0
MAE error:  6.380970145258282
Initial Values: 0.1 10.0 0.75025
RMS error:  1.0418063693208514e+16
MAE error:  6.466575471745845
Initial Values: 0.1 10.0 1.0
RMS error:  2.2264294011216628e+16
MAE error:  6.53205609600666
Best n1i: 0.050050000000000004
Best n2i: 4.0
Best n3i: 1.0
Minimum RMS Deviation: 18.893849164245967

Top 5 Results:
Set       n1i       n2i       n3i       p         NGEN      RMSE      MAE
30        0.02508   2.000     1.000     0.09      5000      18.89     0.0355
77        0.07503   2.000     0.251     0.09      5000      19.89     0.0391
102       0.10000   2.000     0.251     0.09      5000      20.89     0.0393
52        0.05005   2.000     0.251     0.09      5000      21.05     0.0396
26        0.02508   2.000     0.001     0.09      5000      21.63     0.0349

Top 5 Results:
Set       n1i       n2i       n3i       p         NGEN      RMSE      MAE
26        0.02508   2.000     0.001     0.09      5000      21.63     0.0349
30        0.02508   2.000     1.000     0.09      5000      18.89     0.0355
27        0.02508   2.000     0.251     0.09      5000      21.84     0.0356
51        0.05005   2.000     0.001     0.09      5000      21.66     0.0356
7         0.00010   4.000     0.251     0.09      5000      44.10     0.0367
```

In [ ]:

Task 1.3

Code modified to train a five constant model that includes variation of pressure P and the surface tension
parameter γ . Specifically, this new program is design to find the set of constants n1 through n5 in
the performance equation that best fits an expanded data set that includes variable γ and pressure

In this section the only modification was to move the parameters for evolution to the bottom to define
their lenght according to the data set (ydata) lenght

In [234…

```
'''>>>>> start CodeP1.1F23
    V.P. Carey ME249, Fall 2023'''

# version 3 print function
from __future__ import print_function
# seed the pseudorandom number generator
from random import random
from random import seed
# seed random number generator
seed(1)

#import math and numpy packages
import math
import numpy

%matplotlib inline
# importing the required module
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 8] # for square canvas

#import copy
from copy import copy, deepcopy

#create arrays
ydata = []
lydata = []

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
# this is an array that is essentially a list of lists

#assembling data array
#store array where rows are data vectors
#[heat flux, superheat, gravity, surface tension parameter, pressure]

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])

# print the data array
# print ('ydata =', ydata)

''' need deepcopy to create an array of the same size as ydata,
#   since this array is a list(rows) of lists (column entries)  '''
lydata = deepcopy(ydata)  # create array to store ln of data values

# Parameters for Evolution Loop
# set data parameters
ND = len(ydata)       #number of data vectors in array
DI = 5        #number of data items in vector
NS = ND       #total number of DNA strands

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
#now store log values for data
for j in range(DI):
    for i in range(ND):
        lydata[i][j]=math.log(ydata[i][j]+0.000000000010)

#OK now have stored array of log values for data

#end CodeP1.1F23
```

The modification specify in the task where performed:

- Uncommented the red text lines in the ydata array definition to add that additional data
- Change ND and NS to 77--> Parameters for evolution where moved to the bottom to define
  their lenght according to the data set lenght
- Modify the code to compute the appropriate error and heat flux quantities for the five constant mode
- Included constants n4 and n5 in the mating
- Modified for final output of constants
- Altered the plot for the five constant model to commodate for the new constt

Added section:

- For Loop for running multiple sets

- Calculation of RMSE
- Plot of q" predicted v. data
- Results table at the end

In [235…

```
'''INITIALIZING PARAMETERS'''
# Define a list of n values, perturbation, and NGEN to evaluate
n_values_to_evaluate = [
#     [(-1, 0.000476, 3.028, 0.2249, 1.054, 0.217), 0.09, 6000], # Given Initia
#     [(-1, 0.000476, 3.028, 0.2249, 1.054, 0.217), 0.2, 6000],
#     [(-1, 0.000476, 3.028, 0.2249, 1.054, 0.217), 0.3, 6000],
#     [(-1, 0.000476, 3.028, 0.2249, 1.054, 0.217), 0.4, 6000],
#     [(-1, 0.000476, 3.028, 0.2249, 1.054, 0.217), 0.5, 6000],
#     [(-1, 0.000476, 3.028, 0.2249, 1.054, 0.217), 0.6, 6000],
#     [(-1, 0.000476, 3.028, 0.2249, 1.054, 0.217), 0.7, 6000],
#     [(-1, 0.000476, 3.028, 0.2249, 1.054, 0.217), 0.8, 6000],
#     [(-1, 0.000476, 3.028, 0.2249, 1.054, 0.217), 0.9, 6000],


    [(-1, 0.0011, 2.5, 0.4, 1.16, 0.4), 0.09, 6000], # Best Initial guess found
#     [(-1, 0.0011, 2.5, 0.4, 1.16, 0.4), 0.09, 6000],
#     [(-1, 0.0011, 2.5, 0.4, 1.16, 0.4), 0.09, 6000],
#     [(-1, 0.0011, 2.5, 0.4, 1.16, 0.4), 0.09, 6000],
#     [(-1, 0.0011, 2.5, 0.4, 1.16, 0.4), 0.09, 6000],
#     [(-1, 0.0011, 2.5, 0.4, 1.16, 0.4), 0.09, 6000],
#     [(-1, 0.0011, 2.5, 0.4, 1.16, 0.4), 0.09, 6000],
#     [(-1, 0.0011, 2.5, 0.4, 1.16, 0.4), 0.09, 6000],
#     [(-1, 0.0011, 2.5, 0.4, 1.16, 0.4), 0.09, 6000],
]

results = []  # Create an empty list to store summaries

for n_values, perturbation, NGEN in n_values_to_evaluate:

    n = []
    ntemp = []
    gen=[0]
    n1avg = [0.0]
    n2avg = [0.0]
    n3avg = [0.0]
    n4avg = [0.0]
    n5avg = [0.0]
    meanAFerr=[0.0]
    aFerrmeanavgn=[0.0]
    rms_dev = [0.0] # Define RMSE

    # Set program parameters
    NGEN = NGEN      # number of generations (steps)
    MFRAC = 0.5   # faction of median threshold
    perturbation = perturbation # pertubation value

    # here the number of data vectors equals the number of DNA strands (or orga
    # they can be different if they are randomly paired to compute Ferr (survi
    for k in range(NGEN-1):
        gen.append(k+1)   # generation array stores the
        meanAFerr.append(0.0)
        aFerrmeanavgn.append(0.0)
        n1avg.append(0.0)
        n2avg.append(0.0)
```

```
    n3avg.append(0.0)
    n4avg.append(0.0)
    n5avg.append(0.0)

'''guesses for initial solution population'''
n0i, n1i, n2i, n3i, n4i, n5i = n_values

#- initialize arrays before start of evolution loop EL
#then - create array of DNA strands n[i] and ntemp[i] with dimesnion NS = 

#i initialize array where rows are dna vectors [n0i,n1i,...n5i] with randor
n =  [[-1., n1i+0.001*random(), n2i+0.1*random(), n3i+0.0001*random(), n4i-
for i in range(ND):
    n.append([-1., n1i+0.0001*random(), n2i+0.001*random(), n3i+0.0001*rand
#print (n) # uncomment command to print array so it can be checked

# store also in wtemp
ntemp = deepcopy(n)

#initialize Ferr values an dother loop parameters
#define arrays of Ferr (error) functions
#individual solution error and absoute error
Ferr =  [[0.0]]
#population average solution error and absoute error
Ferravgn =  [[0.0]]
aFerr =  [[0.0]]
aFerravgn =  [[0.0]]

#store zeros in ND genes
for i in range(ND-1):
    #individual solution error and absoute error
    Ferr.append([0.0])
    aFerr.append([0.0])
    #population average solution error and absoute error
    Ferravgn.append([0.0])
    aFerravgn.append([0.0])
    rms_dev.append([0.0])
#print (Ferr)

aFerrmeanavgnMin = 1000000000.0
# these store the  n values for minimum population average error durng NGE
n1min = 0.0
n2min = 0.0
n3min = 0.0
n4min = 0.0
n5min = 0.0
aFerrta = 0.0
# these store the time averaged n values durng from generation 800 to NGEN
n1min = 0.0
n1ta = 0.0
n2ta = 0.0
n3ta = 0.0
n4ta = 0.0
n5ta = 0.0

'''START OF EVOLUTION LOOP'''
# ----------------------------------------
# k is generation number, NGEN IS TOTAL NUMBER OF GENERATIONS COMPUTED
for k in range(NGEN):
```

```
'''In this program , the number of organisms (solutions) NS is taken to
number of data points ND so for each generation, each solution can be c
data point and all the data is compared in each generation.  The order
that holds the solution constants is constantly changing due to mating
is random.'''

'''CALCULATING ERROR (FITNESS)
In this program, the absolute error in the logrithm of the physical hea
used to evaluate fitness.'''

# Here we calculate error Ferr and absolute error aFerr for each data p
# for specified n(i), and calculate (mean aFerr) = aFerrmean
# and (median aFerr) = aFerrmedian for the data collection and specifie
# Note that the number data points ND equals the number of solutions (c
#=========================================================================
'''CALCULATING ERROR (FITNESS)'''
for i in range(ND):

    # =====================================================================
    # New function error equation to accomodate for 5 variable equation
    Ferr[i] = n[i][0]*lydata[i][0] + math.log(n[i][1]) + n[i][2]*lydata
    Ferr[i] = Ferr[i] + n[i][3] * math.log( ydata[i][2] + n[i][4]*9.8*y
    Ferr[i] = Ferr[i] + n[i][5] * lydata[i][4]
    # =====================================================================
    aFerr[i] = abs(Ferr[i])/abs(lydata[i][0])  #- absolute fractional e
#-------------
aFerrmean = numpy.mean(aFerr) #mean error for population for this genei
meanAFerr[k]=aFerrmean  #store aFerrmean for this generation gen[k]=k
aFerrmedian = numpy.median(aFerr) #median error for population for this

'''SELECTION'''
#pick survivors
#[2] calculate survival cutoff, set number kept = nkeep = 0
#=============================================
clim = MFRAC*aFerrmedian  #cut off limit is a fraction/multiplier MFRAC
nkeep = 0

# now check each organism/solution to see if aFerr is less than cut of
#if yes, store n for next generation population in ntemp, at end nkeep
#and number of new offspring = NS-nkeep
#=============================================
for j in range(NS):  # NS Ferr values, one for each solution in populat
    if (aFerr[j] < clim):
        nkeep = nkeep + 1
        #ntemp[nkeep][0] = n[j][0] = -1 so it is unchanged;
        ntemp[nkeep-1][1] = n[j][1];
        ntemp[nkeep-1][2] = n[j][2];
        ntemp[nkeep-1][3] = n[j][3];
        ntemp[nkeep-1][4] = n[j][4];
        ntemp[nkeep-1][5] = n[j][5];
#now have survivors in leading entries in list of ntemp vectors from 1
#compute number to be added by mating
nnew = NS - nkeep

'''MATING'''
#[4] for nnew new organisms/solutions,
# randomly pick two survivors, randomly pick DNA (n) from pair for each
#=============================================================
for j in range(nnew):
    # pick two survivors randomly
```

```
            nmate1 = numpy.random.randint(low=0, high=nkeep+1)
            nmate2 = numpy.random.randint(low=0, high=nkeep+1)

            #then randomly pick DNA from parents for offspring

            '''here, do not change property ntemp[nkeep+j+1][0], it's always fi
            #if (numpy.random.rand() < 0.5)
            #    ntemp[nkeep+j+1][0] = n[nmate1][0]
            #else
            #    ntemp[nkeep+j+1][0] = n[nmate2][0]

            # ================================================================
            # Change to original code. 0.09 substitute by "perturbation" varial
            # ================================================================

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][1] = n[nmate1][1]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][1] = n[nmate2][1]*(1.+perturbation*2.*(0.5-nur

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][2] = n[nmate1][2]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][2] = n[nmate2][2]*(1.+perturbation*2.*(0.5-nur

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][3] = n[nmate1][3]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][3] = n[nmate2][3]*(1.+perturbation*2.*(0.5-nur

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][4] = n[nmate1][4]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][4] = n[nmate2][4]*(1.+perturbation*2.*(0.5-nur

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][5] = n[nmate1][5]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][5] = n[nmate2][5]*(1.+perturbation*2.*(0.5-nur

        #============================================
        n = deepcopy(ntemp)   # save ntemp as n for use in next generation (nex


        '''AVERAGING OVER POPULATION AND OVER TIME, FINDING MINIMUM ERROR SET (
        # [6] calculate n1avg[k], etc., which are average n values for populati
        # at this generation k
        #============================================
        #initialize average n's to zero and sum contribution of each member of
        n1avg[k] = 0.0;
        n2avg[k] = 0.0;
        n3avg[k] = 0.0;
        n4avg[k] = 0.0;
        n5avg[k] = 0.0;
        for j in range(NS):
            n1avg[k] = n1avg[k] + n[j][1]/NS;
            n2avg[k] = n2avg[k] + n[j][2]/NS;
            n3avg[k] = n3avg[k] + n[j][3]/NS;
            n4avg[k] = n4avg[k] + n[j][4]/NS;
            n5avg[k] = n5avg[k] + n[j][5]/NS;
```

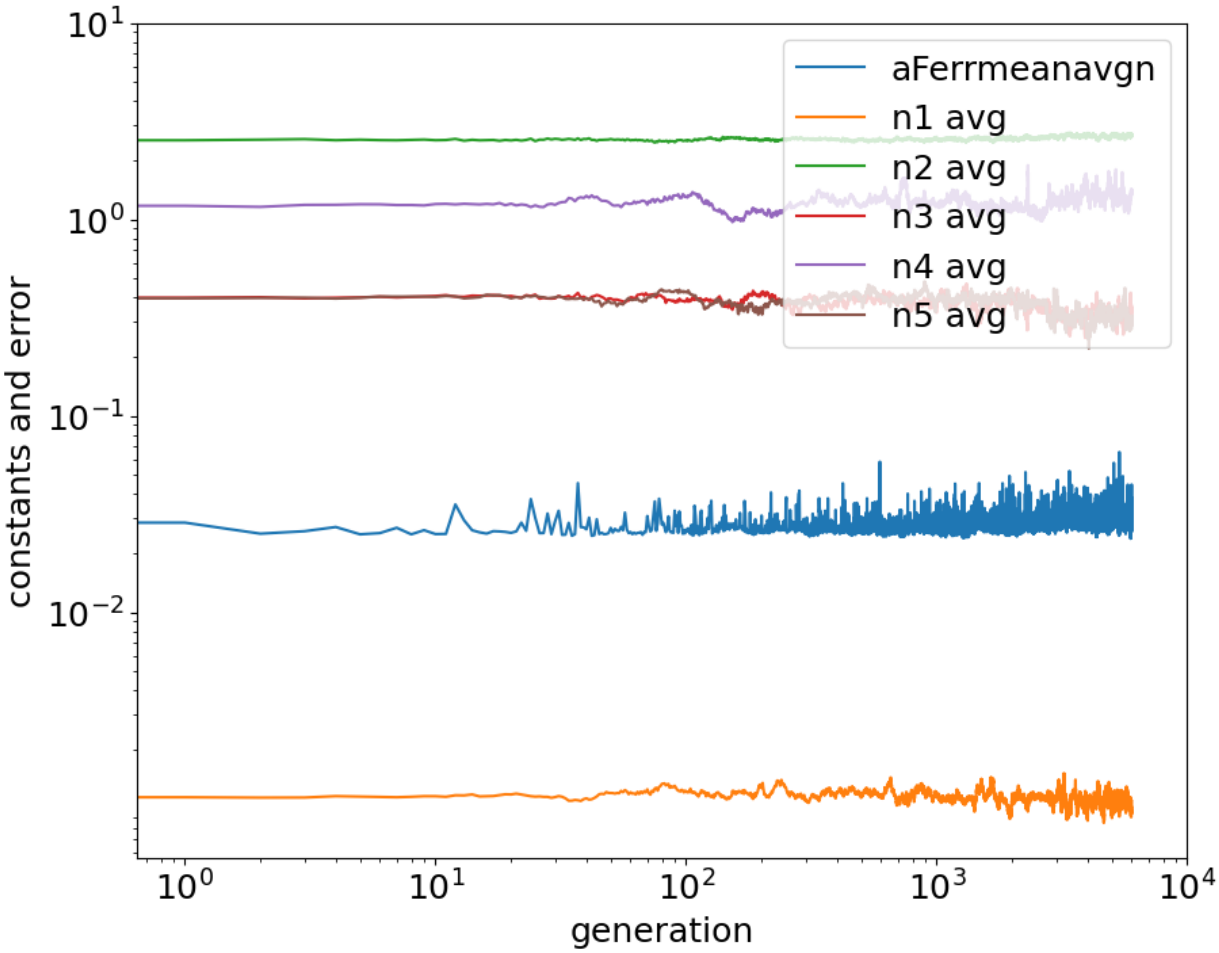
```
        # Here we compute aFerravgn[i] = absolute Ferr of logrithm data point i
        # for this solutions generation k
        # aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of
        #
        #=============================================
        ''' CALCULATING MEAN ERROR FOR POPULATION'''
        for i in range(ND):

            # ================================================================
            # New average function error equation to accomodate for 5 variable
            Ferravgn[i] = -1.*lydata[i][0] + math.log(n1avg[k]) + n2avg[k]*lyda
            Ferravgn[i] = Ferravgn[i] + n3avg[k] * math.log( ydata[i][2] + n4av
            Ferravgn[i] = Ferravgn[i] + n5avg[k] * math.log(ydata[i][4])
            # ================================================================

            aFerravgn[i] = abs(Ferravgn[i])/abs(lydata[i][0])
        #-------------
        aFerrmeanavgn[k] = numpy.mean(aFerravgn)

        # next, update time average of n valaues in population (n1ta[k], etc.)
        # for generations = k > 800 up to total NGEN
        #=============================================
        aFerrta = aFerrta + aFerrmeanavgn[k]/NGEN
        if (k > 800):
            n1ta = n1ta + n1avg[k]/(NGEN-800)
            n2ta = n2ta + n2avg[k]/(NGEN-800)
            n3ta = n3ta + n3avg[k]/(NGEN-800)
            n4ta = n4ta + n4avg[k]/(NGEN-800)
            n5ta = n5ta + n5avg[k]/(NGEN-800)

        # compare aFerrmeanavgn[k] to previous minimum value and save
        # it and corresponding n(i) values if the value for this generation k i
        #=============================================
        if (aFerrmeanavgn[k] < aFerrmeanavgnMin):
            aFerrmeanavgnMin = aFerrmeanavgn[k]
            n1min = n1avg[k]
            n2min = n2avg[k]
            n3min = n3avg[k]
            n4min = n4avg[k]
            n5min = n5avg[k]

        #print('avg n1-n4:', n1avg[k], n2avg[k], n3avg[k], n4avg[k], aFerrmeana
        #print ('kvalue =', k)
        '''end of evolution loop'''
        # ---------------------------------------------------------------------
        # ---------------------------------------------------------------------


    # -------------------------------------------------------------------------
    #final print and plot of results
    # -------------------------------------------------------------------------
    print('Initial Values:', n1i, n2i, n3i, n4i, n5i)
    print('ENDING: pop. avg n1-n3,aFerrmean:', n1avg[k], n2avg[k], n3avg[k], n4
    print('MINUMUM:  avg n1-n3,aFerrmeanMin:', n1min, n2min, n3min, n4min , n5m
    print('TIME AVG:    avg n1-n3,aFerrmean:', n1ta, n2ta, n3ta, n4ta, n5ta, aF

    #SETTING UP PLOTS

    # =================================================================
```

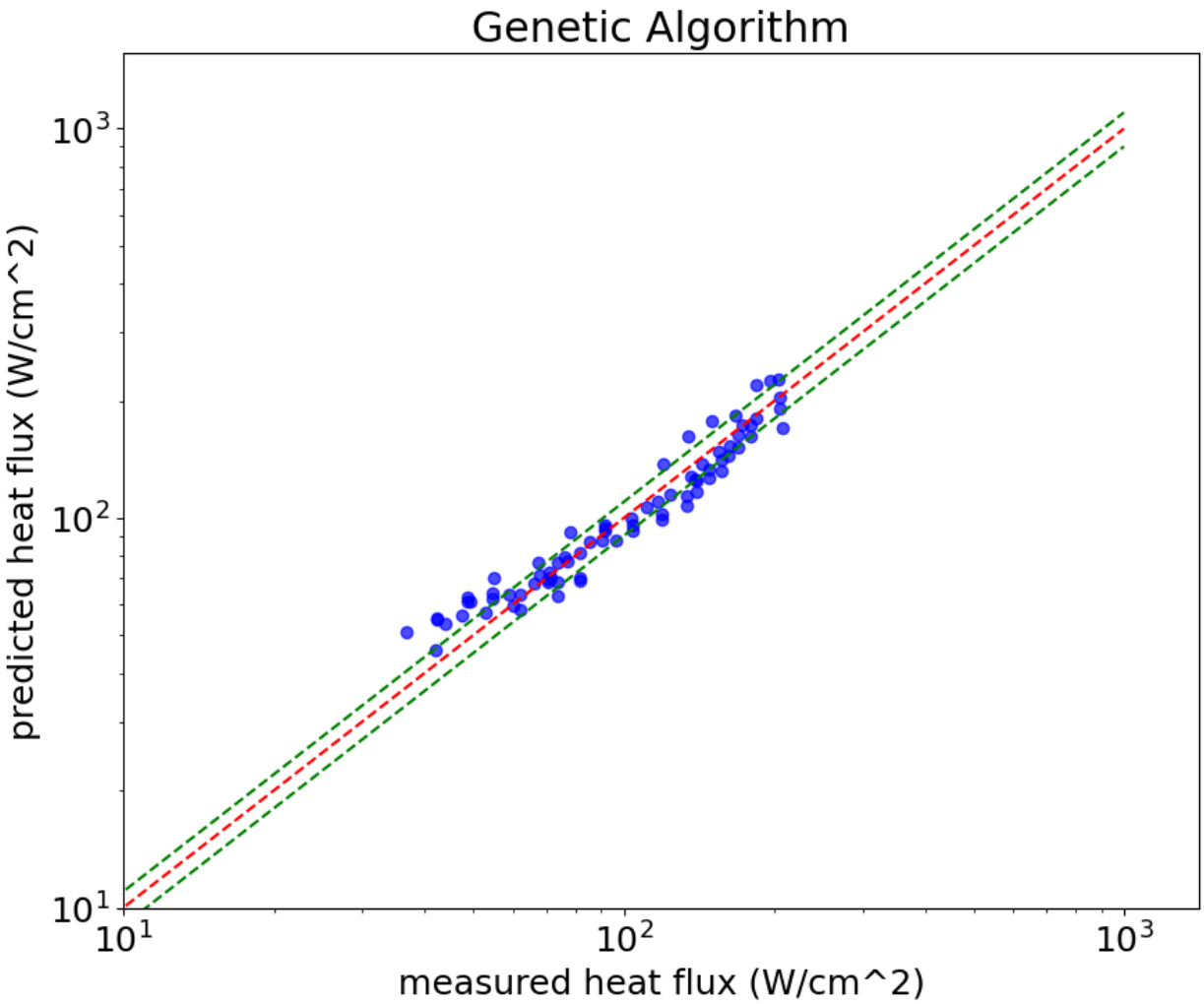
```
# Change to original code. Calculating q" predicted and RMSE with
# respect to the data q"
# ================================================================
#initialize values
qpppred =  [[0.0]]
qppdata =  [[0.0]]
for i in range(ND-1):
    qpppred.append([0.0])
    qppdata.append([0.0])
#calculate predicted and data values to plot
for i in range(ND):
    qpppred[i] = n1min * (ydata[i][1]**n2min) * ((ydata[i][2] + n4min*9.8*y
    qppdata[i] = ydata[i][0]
#Calculationg RMS btw qppdata and qpppred
for i in range(ND):
    rms_dev[i] = (numpy.array(qppdata[i]) - numpy.array(qpppred[i]))**2
rms_dev = numpy.sqrt(numpy.sum(rms_dev) / ND)

print('RMS error: ', rms_dev)

# ================================================================
# Change to original code. After calculating n1min, n2min, n3min,
# n4min, n5min and rms_dev, create a dictionary to store these values
# ================================================================
# After calculating n1min, n2min, n3min, and rms_dev, create a dictionary
# to store these values
iteration_result = {
    'Set': len(results) + 1,
    'n1i': n1i,
    'n2i': n2i,
    'n3i': n3i,
    'n4i': n4i,
    'n5i': n5i,
    'n1min': n1min,
    'n2min': n2min,
    'n3min': n3min,
    'n4min': n4min,
    'n5min': n5min,
    'n1avg[k]' : n1avg[k],
    'n2avg[k]' : n2avg[k],
    'n3avg[k]' : n3avg[k],
    'n4avg[k]' : n4avg[k],
    'n5avg[k]' : n5avg[k],
    'p': perturbation,
    'NGEN': NGEN,
    'rms_dev': rms_dev,
    'aFerrmeanavgnMin': aFerrmeanavgnMin,
    # Add more values as needed
}

results.append(iteration_result)  # Append the summary dictionary to the li
#========

# constants evolution plots
# x axis values are generation number
# corresponding y axis values are mean absolute population error aFerrmeana
# plotting the points

plt.rcParams.update({'font.size': 18})
```

```
    # aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of org
    # computed using the mean n values
    plt.plot(gen, aFerrmeanavgn)
    plt.plot(gen, n1avg)
    plt.plot(gen, n2avg)
    plt.plot(gen, n3avg)
    plt.plot(gen, n4avg)
    plt.plot(gen, n5avg)
    plt.legend(['aFerrmeanavgn', 'n1 avg', 'n2 avg', 'n3 avg', 'n4 avg', 'n5 av

    # naming the x axis
    plt.xlabel('generation')
    # naming the y axis
    plt.ylabel('constants and error')
    plt.loglog()
    plt.yticks([0.01,0.1,1.0,10])
    plt.xticks([1,10,100,1000,10000])
    plt.show()


    '''CALCULATE PREDICTED VALUES AND RETRIEVE DATA VALUES TO PLOT'''

    # ============================================================================
    # Change to original code. Adding perfect algorithm prediction reference li
    # and +/-10% uncertainty boundaries
    # ============================================================================
    x_values = numpy.logspace(0, 3, 100)   # Adjust the range of x values as ne
    y_values = x_values                     # y = k*x
    y_values1 = x_values + (0.1 * x_values) # y = k*x + 0.1*x (+10% uncertainty
    y_values2 = x_values - (0.1 * x_values) # y = k*x - 0.1*x (-10% uncertainty
    # Plot reference equations
    plt.plot(x_values, y_values1, color='green', linestyle='--', label='y = k*x
    plt.plot(x_values, y_values2, color='green', linestyle='--', label='y = k*x
    plt.plot(x_values, y_values, color='red', linestyle='--', label='y = k*x')
    # Plot Heat Flux predicted vs. data
    plt.scatter(qppdata, qpppred, label='Data', color='blue', alpha=0.7)
    plt.title('Genetic Algorithm')
    plt.xlabel('measured heat flux (W/cm^2)')
    plt.ylabel('predicted heat flux (W/cm^2)')
    plt.loglog()
    plt.xlim(xmin = 10)
    plt.ylim(ymin = 10)

    plt.show()

# Table of Results
print("Results Table 1:")
# print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'n4i':<10}{'n5i':<10}{'n
# for result in results:
#     print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{i

print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'n4i':<10}{'n5i':<10}{'p'
for result in results:
    print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{res

print("Results Table 2:")
print(f"{'Set':<10}{'n1min':<10}{'n2min':<10}{'n3min':<10}{'n4min':<10}{'n5min
for result in results:
    print(f"{result['Set']:<10}{result['n1min']:<10.5f}{result['n2min']:<10.3f}
```

```
print("Results Table 3:")
print(f"{'Set':<10}{'n1avg[k]':<10}{'n2avg[k]':<10}{'n3avg[k]':<10}{'n4avg[k]':
for result in results:
    print(f"{result['Set']:<10}{result['n1avg[k]']:<10.5f}{result['n2avg[k]']:<

# end CodeP1.2F23
```

```
Initial Values: 0.0011 2.5 0.4 1.16 0.4
ENDING: pop. avg n1-n3,aFerrmean: 0.0010923071899640346 2.6321554033134893 0.3
41433376905888 1.3865138088042923 0.32253136355572715 0.025907594141767007
MINUMUM:  avg n1-n3,aFerrmeanMin: 0.000932083573767573 2.6241479719509546 0.41
563568965805585 1.2685245994001693 0.31041419278861004 0.023709922373660736
TIME AVG:    avg n1-n3,aFerrmean: 0.00111876505753195 2.6190885639786226 0.345
4593467955371 1.25856809082465 0.3457641229563033 0.028084529160756055
RMS error:  13.786317719481094
```



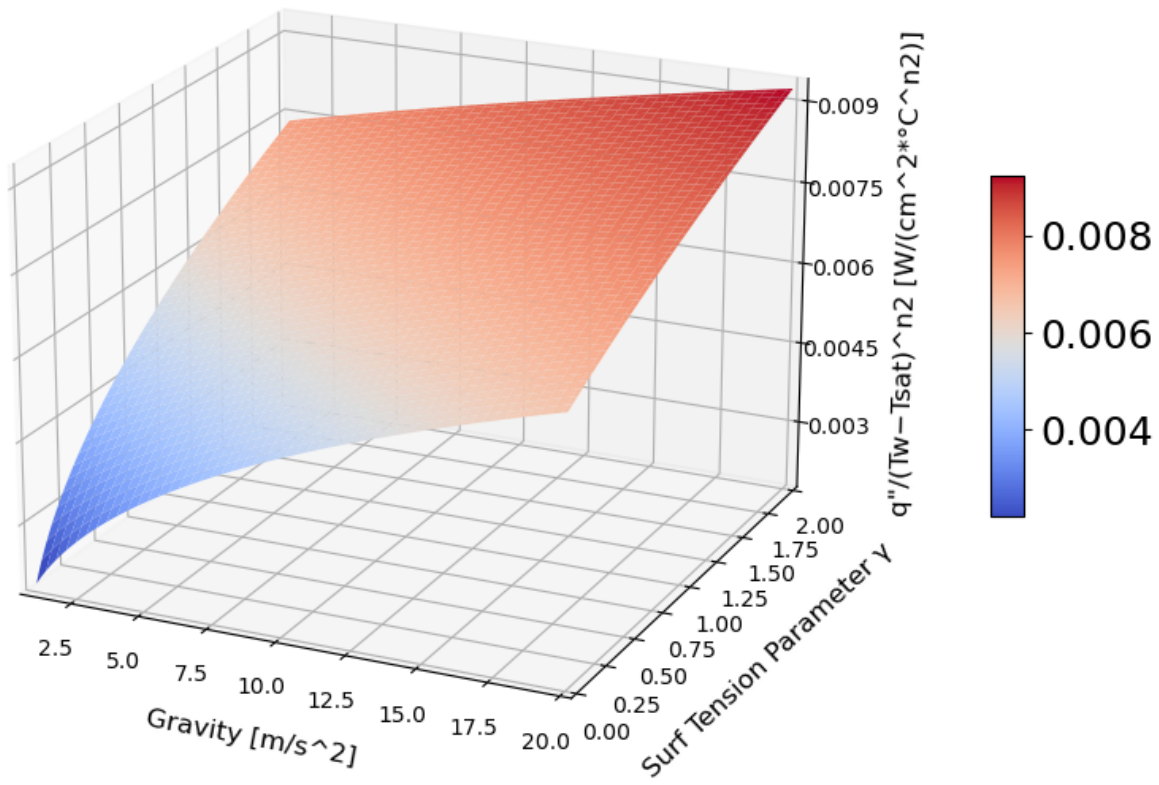



```
Results Table 1:
Set       n1i       n2i       n3i       n4i       n5i       p         NGEN
RMSE      MAE
1         0.00110   2.500     0.400     1.160     0.400     0.09      6000
13.79     0.0237
Results Table 2:
Set       n1min     n2min     n3min     n4min     n5min     p         NGEN
RMSE      MAE
1         0.00093   2.624     0.416     1.269     0.310     0.09      6000
13.79     0.0237
Results Table 3:
Set       n1avg[k]  n2avg[k]  n3avg[k]  n4avg[k]  n5avg[k]  p         NGEN
RMSE      MAE
1         0.00109   2.632     0.341     1.387     0.323     0.09      6000
13.79     0.0237
```

In [246…

```python
# Import necessary libraries
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator, FuncFormatter, MaxNLocator

# Define a custom formatting function to round to a specific number of signific
def format_z(value, _):
    # Specify the number of significant figures you want (e.g., 3)
    num_significant_figures = 3
    return f"{value:.{num_significant_figures}g}"

# Define the range of X and Y values
X = numpy.linspace(1,20,100) # Gravity
```

```
Y = numpy.linspace(0,2,100)  # Surface tension parameter
X, Y = numpy.meshgrid(X, Y)

# Calculate Z using your equation
Z = n1min * ((X + n4min * 9.8 * Y) ** n3min) * (10 ** n5min)

# Create a 3D figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot the 3D surface
surf = ax.plot_surface(X, Y, Z, cmap='coolwarm')
fig.colorbar(surf, shrink=0.35, aspect=10, pad=0.1)

# Set labels for the axes
ax.set_xlabel('Gravity [m/s^2]', fontsize=12, labelpad=10)
ax.set_ylabel('Surf Tension Parameter γ', fontsize=12, labelpad=10)
ax.set_zlabel('q"/(Tw-Tsat)^n2 [W/(cm^2*°C^n2)]', fontsize=12, labelpad=10)

# Set limits for the axes
ax.set_xlim(1.0, 20.0)
ax.set_ylim(0.0, 2.0)
# ax.set_zlim(0.001, 0.003)

# ax.zaxis.set_major_locator(LinearLocator(10))
# Reduce the number of ticks on the z-axis
ax.zaxis.set_major_locator(MaxNLocator(nbins=5))

# Customize tick formatting for z-axis using the custom formatting function
ax.zaxis.set_major_formatter(FuncFormatter(format_z))

# Customize tick formatting for z-axis
ax.tick_params(axis='both', which='major', labelsize=10)

# Set the view perspective to orient the plot
ax.view_init(elev=20, azim=-65)  # Adjust the angles as needed

# Show the plot
plt.show()
```


0.009
0.0075
0.006
0.0045
0.003
q"/(Tw−Tsat)^n2 [W/(cm^2*°C^n2)]
0.008
0.006
0.004
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Gravity [m/s^2]
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
Surf Tension Parameter γ

In [ ]:

Task 1.3 (extra)

The code was generate using the provided code with the only goal to SEARCH THROUGH A LINEAR SPACE
of each of the variables (n1, n2, n3, n4, n5) to obtain the lowest values of RMSE and MAE. 16,807
combinations were tested. Below is an example for running this cells for 243 combinations.

In [3]:
```
'''>>>>> start CodeP1.1F23
    V.P. Carey ME249, Fall 2023'''

# version 3 print function
from __future__ import print_function
# seed the pseudorandom number generator
from random import random
from random import seed
# seed random number generator
seed(1)

#import math and numpy packages
import math
import numpy as np
import numpy as numpy

%matplotlib inline
# importing the required module
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 8] # for square canvas

#import copy
from copy import copy, deepcopy

#create arrays
ydata = []
lydata = []

#Parameters for Evolution Loop
#set data parameters
ND = 45       #number of data vectors in array
DI = 5        #number of data items in vector
NS = 45       #total number of DNA strands

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
# this is an array that is essentially a list of lists

#assembling data array
#store array where rows are data vectors
#[heat flux, superheat, gravity, surface tension parameter, pressure]

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, 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])

''' need deepcopy to create an array of the same size as ydata,
#   since this array is a list(rows) of lists (column entries)  '''
lydata = deepcopy(ydata)  # create array to store ln of data values

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
#now store log values for data
for j in range(DI):
    for i in range(ND):
        lydata[i][j]=math.log(ydata[i][j]+0.000000000010)

#OK now have stored array of log values for data
```

To automate finding the initial values (n1i, n2i, n3i, n4i, n5i) for the lowest RMS deviation percent, we implement
a search algorithm. The approach is to use a grid search where it systematically explore

different
combinations of initial values within certain ranges and then select the combination that results in an
RMS deviation closest to the desire RMS deviation.

Added section:

- Define linspace for each variable at a range
- For Loop for running through the multiple sets
- Calculation of RMSE an segregate top results
- Calculation of MAE an segregate top results
- Print both tables

In [4]:
```
# Define search ranges for n1i, n2i, and n3i
n1i_range = np.linspace(0.0001, 0.1, 3)  # Adjust the range as needed
n2i_range = np.linspace(2.0, 10.0, 3)     # Adjust the range as needed
n3i_range = np.linspace(0.01, 1.0, 3)     # Adjust the range as needed
n4i_range = np.linspace(0.1, 5.0, 3)     # Adjust the range as needed
n5i_range = np.linspace(0.01, 5.0, 3)     # Adjust the range as needed

# Initialize variables to store the best values and minimum RMS deviation
best_n1i = 0.0
best_n2i = 0.0
best_n3i = 0.0
best_n4i = 0.0
best_n5i = 0.0
min_rms_deviation = float('inf')  # Initialize with a large value
min_MAE = float('inf') # Initialize with a large value

# Define a list to store the top 5 results
top_results = []     # RMSE
top_results_MAE = [] # MAE

# Create an empty list to store summaries
results = []

# Iterate through the search ranges
for n1i in n1i_range:
    for n2i in n2i_range:
        for n3i in n3i_range:
            for n4i in n4i_range:
                for n5i in n5i_range:
                    '''INITIALIZING PARAMETERS'''
                    # Define a list of n values, perturbation, and NGEN to eva
                    n_values_to_evaluate = [
                        [(-1, n1i, n2i, n3i, n4i, n5i), 0.09, 3000]]  # Given

                    for n_values, perturbation, NGEN in n_values_to_evaluate:

                        n = []
                        ntemp = []
                        gen=[0]
                        n1avg = [0.0]
                        n2avg = [0.0]
                        n3avg = [0.0]
                        n4avg = [0.0]
```

```
n5avg = [0.0]
meanAFerr=[0.0]
aFerrmeanavgn=[0.0]
rms_dev = [0.0] # Define rms_dev

# Set program parameters
NGEN = NGEN      # number of generations (steps)
MFRAC = 0.5   # faction of median threshold
perturbation = perturbation # pertubation value

# here the number of data vectors equals the number of
# they can be different if they are randomly paired to
for k in range(NGEN-1):
    gen.append(k+1)   # generation array stores the
    meanAFerr.append(0.0)
    aFerrmeanavgn.append(0.0)
    n1avg.append(0.0)
    n2avg.append(0.0)
    n3avg.append(0.0)
    n4avg.append(0.0)
    n5avg.append(0.0)

'''guesses for initial solution population'''
n0i, n1i, n2i, n3i, n4i, n5i = n_values

#- initialize arrays before start of evolution loop EL
#then - create array of DNA strands n[i] and ntemp[i] w

#i initialize array where rows are dna vectors [n0i,n1i
n =  [[-1., n1i+0.001*random(), n2i+0.1*random(), n3i+(
for i in range(ND):
    n.append([-1., n1i+0.0001*random(), n2i+0.001*rando
#print (n) # uncomment command to print array so it can

# store also in wtemp
ntemp = deepcopy(n)

#initialize Ferr values an dother loop parameters
#define arrays of Ferr (error) functions
#individual solution error and absoute error
Ferr =  [[0.0]]
#population average solution error and absoute error
Ferravgn =  [[0.0]]
aFerr =  [[0.0]]
aFerravgn =  [[0.0]]

#store zeros in ND genes
for i in range(ND-1):
    #individual solution error and absoute error
    Ferr.append([0.0])
    aFerr.append([0.0])
    #population average solution error and absoute erro
    Ferravgn.append([0.0])
    aFerravgn.append([0.0])
    rms_dev.append([0.0])
#print (Ferr)

aFerrmeanavgnMin = 1000000000.0
# these store the  n values for minimum population aver
n1min = 0.0
```

```
n2min = 0.0
n3min = 0.0
n4min = 0.0
n5min = 0.0
aFerrta = 0.0
# these store the time averaged n values durng from ge
n1min = 0.0
n1ta = 0.0
n2ta = 0.0
n3ta = 0.0
n4ta = 0.0
n5ta = 0.0

'''START OF EVOLUTION LOOP'''
# ---------------------------------------
# k is generation number, NGEN IS TOTAL NUMBER OF GENE
for k in range(NGEN):

    '''In this program , the number of organisms (solu
    number of data points ND so for each generation, e
    data point and all the data is compared in each ge
    that holds the solution constants is constantly ch
    is random.'''

    '''CALCULATING ERROR (FITNESS)
    In this program, the absolute error in the logrith
    used to evaluate fitness.'''

    # Here we calculate error Ferr and absolute error
    # for specified n(i), and calculate (mean aFerr) =
    # and (median aFerr) = aFerrmedian for the data co
    # Note that the number data points ND equals the n
    #================================================
    '''CALCULATING ERROR (FITNESS)'''
    for i in range(ND):

        # ============================================
        # New function error equation to accomodate fo
        Ferr[i] = n[i][0]*lydata[i][0] + math.log(n[i]
        Ferr[i] = Ferr[i] + n[i][3] * math.log( ydata[
        Ferr[i] = Ferr[i] + n[i][5] * lydata[i][4]
        # ============================================
        aFerr[i] = abs(Ferr[i])/abs(lydata[i][0])  #- a
    #-------------
    aFerrmean = numpy.mean(aFerr) #mean error for popu
    meanAFerr[k]=aFerrmean  #store aFerrmean for this
    aFerrmedian = numpy.median(aFerr) #median error fo

    '''SELECTION'''
    #pick survivors
    #[2] calculate survival cutoff, set number kept = n
    #============================================
    clim = MFRAC*aFerrmedian  #cut off limit is a frac
    nkeep = 0

    # now check each organism/solution to see if aFerr
    #if yes, store n for next generation population in
    #and number of new offspring = NS-nkeep
    #============================================
    for j in range(NS):  # NS Ferr values, one for eac
```

```
            if (aFerr[j] < clim):
                nkeep = nkeep + 1
                #ntemp[nkeep][0] = n[j][0] = -1 so it is u
                ntemp[nkeep-1][1] = n[j][1];
                ntemp[nkeep-1][2] = n[j][2];
                ntemp[nkeep-1][3] = n[j][3];
                ntemp[nkeep-1][4] = n[j][4];
                ntemp[nkeep-1][5] = n[j][5];
        #now have survivors in leading entries in list of 
        #compute number to be added by mating
        nnew = NS - nkeep

        '''MATING'''
        #[4] for nnew new organisms/solutions,
        # randomly pick two survivors, randomly pick DNA (
        #================================================
        for j in range(nnew):
            # pick two survivors randomly
            nmate1 = numpy.random.randint(low=0, high=nkeep
            nmate2 = numpy.random.randint(low=0, high=nkeep

            #then randomly pick DNA from parents for offsp

            '''here, do not change property ntemp[nkeep+j+1
            #if (numpy.random.rand() < 0.5)
            #    ntemp[nkeep+j+1][0] = n[nmate1][0]
            #else
            #    ntemp[nkeep+j+1][0] = n[nmate2][0]

            # ============================================
            # Change to original code. 0.09 substitute by
            # ============================================

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][1] = n[nmate1][1]*(1.+pert
            else:
                ntemp[nkeep+j+1][1] = n[nmate2][1]*(1.+pert

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][2] = n[nmate1][2]*(1.+pert
            else:
                ntemp[nkeep+j+1][2] = n[nmate2][2]*(1.+pert

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][3] = n[nmate1][3]*(1.+pert
            else:
                ntemp[nkeep+j+1][3] = n[nmate2][3]*(1.+pert

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][4] = n[nmate1][4]*(1.+pert
            else:
                ntemp[nkeep+j+1][4] = n[nmate2][4]*(1.+pert

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][5] = n[nmate1][5]*(1.+pert
            else:
                ntemp[nkeep+j+1][5] = n[nmate2][5]*(1.+pert

        #============================================
        n = deepcopy(ntemp)   # save ntemp as n for use in
```

```
            '''AVERAGING OVER POPULATION AND OVER TIME, FINDIN
            # [6] calculate n1avg[k], etc., which are average
            # at this generation k
            #=========================================
            #initialize average n's to zero and sum contributi
            n1avg[k] = 0.0;
            n2avg[k] = 0.0;
            n3avg[k] = 0.0;
            n4avg[k] = 0.0;
            n5avg[k] = 0.0;
            for j in range(NS):
                n1avg[k] = n1avg[k] + n[j][1]/NS;
                n2avg[k] = n2avg[k] + n[j][2]/NS;
                n3avg[k] = n3avg[k] + n[j][3]/NS;
                n4avg[k] = n4avg[k] + n[j][4]/NS;
                n5avg[k] = n5avg[k] + n[j][5]/NS;

            # Here we compute aFerravgn[i] = absolute Ferr of
            # for this solutions generation k
            # aFerrmeanavgn[k] is the mean of the Ferravgn[i]
            #
            #=========================================
            ''' CALCULATING MEAN ERROR FOR POPULATION'''
            for i in range(ND):
                # ========================================
                # New average function error equation to accom
                Ferravgn[i] = -1.*lydata[i][0] + math.log(n1av
                Ferravgn[i] = Ferravgn[i] + n3avg[k] * math.lo
                Ferravgn[i] = Ferravgn[i] + n5avg[k] * math.lo
                # ========================================

                aFerravgn[i] = abs(Ferravgn[i])/abs(lydata[i][
            #-------------
            aFerrmeanavgn[k] = numpy.mean(aFerravgn)

            # next, update time average of n valaues in popula
            # for generations = k > 800 up to total NGEN
            #=========================================
            aFerrta = aFerrta + aFerrmeanavgn[k]/NGEN
            if (k > 800):
                n1ta = n1ta + n1avg[k]/(NGEN-800)
                n2ta = n2ta + n2avg[k]/(NGEN-800)
                n3ta = n3ta + n3avg[k]/(NGEN-800)
                n4ta = n4ta + n4avg[k]/(NGEN-800)
                n5ta = n5ta + n5avg[k]/(NGEN-800)

            # compare aFerrmeanavgn[k] to previous minimum valu
            # it and corresponding n(i) values if the value fo
            #=========================================
            if (aFerrmeanavgn[k] < aFerrmeanavgnMin):
                aFerrmeanavgnMin = aFerrmeanavgn[k]
                n1min = n1avg[k]
                n2min = n2avg[k]
                n3min = n3avg[k]
                n4min = n4avg[k]
                n5min = n5avg[k]

            #print('avg n1-n4:', n1avg[k], n2avg[k], n3avg[k],
            #print ('kvalue =', k)
```

```
            '''end of evolution loop'''
            # ------------------------------------------------
            # ------------------------------------------------

        # ------------------------------------------------
        #final print and plot of results
        # ------------------------------------------------
        print('Initial Values:', n1i, n2i, n3i, n4i, n5i)

        #SETTING UP PLOTS
        #========
        #initialize values
        qpppred =  [[0.0]]
        qppdata =  [[0.0]]
        for i in range(ND-1):
            qpppred.append([0.0])
            qppdata.append([0.0])
        #calculate predicted and data values to plot
        for i in range(ND):
            qpppred[i] = n1min * (ydata[i][1]**n2min) * ((ydata
            qppdata[i] = ydata[i][0]
        #Calculationg RMS btw qppdata and qpppred
        for i in range(ND):
            rms_dev[i] = (numpy.array(qppdata[i]) - numpy.array
        rms_dev = numpy.sqrt(numpy.sum(rms_dev) / ND)
        print('RMS error: ', rms_dev)
        print('MAE error: ', aFerrmeanavgnMin)

        # After calculating n1min, n2min, n3min, and rms_dev, c
        iteration_result = {
            'Set': len(results) + 1,
            'n1i': n1i,
            'n2i': n2i,
            'n3i': n3i,
            'n4i': n4i,
            'n5i': n5i,
            'n1min': n1min,
            'n2min': n2min,
            'n3min': n3min,
            'n4min': n4min,
            'n5min': n5min,
            'p': perturbation,
            'NGEN': NGEN,
            'rms_dev': rms_dev,
            'aFerrmeanavgnMin': aFerrmeanavgnMin,
            # Add more values as needed
        }

        results.append(iteration_result)  # Append the summary

        # Check if the current combination results in a lower l
        if abs(rms_dev - 1.0) < abs(min_rms_deviation - 1.0):
            # Update the best values
            best_n1i = n1i
            best_n2i = n2i
            best_n3i = n3i
            best_n4i = n4i
            best_n5i = n5i
            min_rms_deviation = rms_dev
        # Check if the current combination results in a lower l
```

```
                        if abs(aFerrmeanavgnMin - 1.0) < abs(min_MAE - 1.0):
                            # Update the best values
                            best_n1i = n1i
                            best_n2i = n2i
                            best_n3i = n3i
                            best_n4i = n4i
                            best_n5i = n5i
                            min_MAE = aFerrmeanavgnMin

                        # Check if this result is among the top 5 results with
                        if len(top_results) < 5:
                            top_results.append(iteration_result)
                            top_results.sort(key=lambda x: x['rms_dev'])  # So
                        elif rms_dev < top_results[-1]['rms_dev']:
                            top_results.pop()  # Remove the result with the hig
                            top_results.append(iteration_result)
                            top_results.sort(key=lambda x: x['rms_dev'])  # So

                        # Check if this result is among the top 5 results with
                        if len(top_results_MAE) < 5:
                            top_results_MAE.append(iteration_result)
                            top_results_MAE.sort(key=lambda x: x['aFerrmeanavgr
                        elif aFerrmeanavgnMin < top_results_MAE[-1]['aFerrmeana
                            top_results_MAE.pop()  # Remove the result with the
                            top_results_MAE.append(iteration_result)
                            top_results_MAE.sort(key=lambda x: x['aFerrmeanavgr

# ====================================================
# Long version Table (adds info of minimum n values)
# ====================================================
# # Print the best values and minimum RMS deviation
# print("Best n1i:", best_n1i)
# print("Best n2i:", best_n2i)
# print("Best n3i:", best_n3i)
# print("Minimum RMS Deviation:", min_rms_deviation)

# # Print the top 5 results
# print("\nTop 5 Results:")
# print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'n1min':<10}{'n2min':<1
# for result in top_results:
#     print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{r

# # Print the top 5 results
# print("\nTop 5 Results:")
# print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'n1min':<10}{'n2min':<1
# for result in top_results_MAE:
#     print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{r

# ====================================================
# Short version Table (ommits info of minimum n values)
# ====================================================
# Print the best values and minimum RMS deviation
# print("Best n1i:", best_n1i)
# print("Best n2i:", best_n2i)
# print("Best n3i:", best_n3i)
# print("Best n4i:", best_n4i)
# print("Best n5i:", best_n5i)
# print("Minimum RMS Deviation:", min_rms_deviation)

# Print the top 5 results
```

```
print("\nTop 5 RMSE Results:")
print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'n4i':<10}{'n5i':<10}{'p'
for result in top_results:
    print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{res

# Print the top 5 results
print("\nTop 5 MAE Results:")
print(f"{'Set':<10}{'n1i':<10}{'n2i':<10}{'n3i':<10}{'n4i':<10}{'n5i':<10}{'p'
for result in top_results_MAE:
    print(f"{result['Set']:<10}{result['n1i']:<10.5f}{result['n2i']:<10.3f}{res
```

```
Initial Values: 0.0001 2.0 0.01 0.1 0.01
RMS error:  122.9490154941606
MAE error:  1.0970670039838033
Initial Values: 0.0001 2.0 0.01 0.1 2.505
RMS error:  22.097409165635376
MAE error:  0.034227580364359626
Initial Values: 0.0001 2.0 0.01 0.1 5.0
RMS error:  22.076483595224445
MAE error:  0.035810959913122636
Initial Values: 0.0001 2.0 0.01 2.5500000000000003 0.01
RMS error:  121.43251534311666
MAE error:  0.8546030453203479
Initial Values: 0.0001 2.0 0.01 2.5500000000000003 2.505
RMS error:  21.016322196784856
MAE error:  0.036213822216188726
Initial Values: 0.0001 2.0 0.01 2.5500000000000003 5.0
RMS error:  21.66593567962583
MAE error:  0.035483520087791515
Initial Values: 0.0001 2.0 0.01 5.0 0.01
RMS error:  121.76597714455136
MAE error:  0.8897862990451905
Initial Values: 0.0001 2.0 0.01 5.0 2.505
RMS error:  22.852386118010248
MAE error:  0.03342313059485806
Initial Values: 0.0001 2.0 0.01 5.0 5.0
RMS error:  24.740750200766016
MAE error:  0.04603709543002062
Initial Values: 0.0001 2.0 0.505 0.1 0.01
RMS error:  121.33792517702412
MAE error:  0.8651555368712388
Initial Values: 0.0001 2.0 0.505 0.1 2.505
RMS error:  17.168528411409874
MAE error:  0.0310572640518339
Initial Values: 0.0001 2.0 0.505 0.1 5.0
RMS error:  19.3782488546151
MAE error:  0.02711727267144029
Initial Values: 0.0001 2.0 0.505 2.5500000000000003 0.01
RMS error:  14.62375348680055
MAE error:  0.02648634187985586
Initial Values: 0.0001 2.0 0.505 2.5500000000000003 2.505
RMS error:  17.723778789237134
MAE error:  0.03232237391536138
Initial Values: 0.0001 2.0 0.505 2.5500000000000003 5.0
RMS error:  43561.25424492056
MAE error:  1.3091453162522755
Initial Values: 0.0001 2.0 0.505 5.0 0.01
RMS error:  19.148446958856493
MAE error:  0.02850772027070464
Initial Values: 0.0001 2.0 0.505 5.0 2.505
RMS error:  17.96153717002705
MAE error:  0.031710843456417476
Initial Values: 0.0001 2.0 0.505 5.0 5.0
RMS error:  69839.33865023212
MAE error:  1.411657599427664
Initial Values: 0.0001 2.0 1.0 0.1 0.01
RMS error:  21.41540115336809
MAE error:  0.02653431846021187
Initial Values: 0.0001 2.0 1.0 0.1 2.505
RMS error:  71.60720829964805
MAE error:  0.18702722157182686
```

```
Initial Values: 0.0001 2.0 1.0 0.1 5.0
RMS error:  93.97833777606226
MAE error:  0.14066483844350494
Initial Values: 0.0001 2.0 1.0 2.5500000000000003 0.01
RMS error:  15.159178249786622
MAE error:  0.027449382798151255
Initial Values: 0.0001 2.0 1.0 2.5500000000000003 2.505
RMS error:  18.228944335124833
MAE error:  0.03464328047409908
Initial Values: 0.0001 2.0 1.0 2.5500000000000003 5.0
RMS error:  345776.4289853231
MAE error:  1.761760337060895
Initial Values: 0.0001 2.0 1.0 5.0 0.01
RMS error:  16.1944802750774
MAE error:  0.029707702991710265
Initial Values: 0.0001 2.0 1.0 5.0 2.505
RMS error:  5329.852490148624
MAE error:  0.8548997372909871
Initial Values: 0.0001 2.0 1.0 5.0 5.0
RMS error:  385818.06588629575
MAE error:  1.7861321771116785
Initial Values: 0.0001 6.0 0.01 0.1 0.01
RMS error:  2511406.8098698393
MAE error:  2.0241852308486195
Initial Values: 0.0001 6.0 0.01 0.1 2.505
RMS error:  343786698.0025551
MAE error:  3.1003667862559863
Initial Values: 0.0001 6.0 0.01 0.1 5.0
RMS error:  29975696310.86473
MAE error:  4.0757070180849695
Initial Values: 0.0001 6.0 0.01 2.5500000000000003 0.01
RMS error:  6602939.495139219
MAE error:  2.2323419977853596
Initial Values: 0.0001 6.0 0.01 2.5500000000000003 2.505
RMS error:  244249460.867562
MAE error:  3.0225465812787875
Initial Values: 0.0001 6.0 0.01 2.5500000000000003 5.0
RMS error:  13174855633.214622
MAE error:  3.895730027874998
Initial Values: 0.0001 6.0 0.01 5.0 0.01
RMS error:  2354300.4610080584
MAE error:  2.009089497065458
Initial Values: 0.0001 6.0 0.01 5.0 2.505
RMS error:  265766629.6629819
MAE error:  3.0423065106730123
Initial Values: 0.0001 6.0 0.01 5.0 5.0
RMS error:  29371390258.730648
MAE error:  4.074327222412628
Initial Values: 0.0001 6.0 0.505 0.1 0.01
RMS error:  4140778.1022312003
MAE error:  2.128498046221474
Initial Values: 0.0001 6.0 0.505 0.1 2.505
RMS error:  694229214.4991943
MAE error:  3.2470782574112396
Initial Values: 0.0001 6.0 0.505 0.1 5.0
RMS error:  26500339709.92957
MAE error:  4.046509046176113
Initial Values: 0.0001 6.0 0.505 2.5500000000000003 0.01
RMS error:  42566054.49275608
MAE error:  2.6476712496730763
```

```
Initial Values: 0.0001 6.0 0.505 2.5500000000000003 2.505
RMS error:  517378696.68888766
MAE error:  3.194807635332414
Initial Values: 0.0001 6.0 0.505 2.5500000000000003 5.0
RMS error:  71758639513.18503
MAE error:  4.273059681068089
Initial Values: 0.0001 6.0 0.505 5.0 0.01
RMS error:  57659163.54054641
MAE error:  2.7110390325039666
Initial Values: 0.0001 6.0 0.505 5.0 2.505
RMS error:  4547607630.284076
MAE error:  3.666805406627988
Initial Values: 0.0001 6.0 0.505 5.0 5.0
RMS error:  296833073980.2275
MAE error:  4.583277073073437
Initial Values: 0.0001 6.0 1.0 0.1 0.01
RMS error:  30360483.89485538
MAE error:  2.499337010457253
Initial Values: 0.0001 6.0 1.0 0.1 2.505
RMS error:  1235103583.8442228
MAE error:  3.30944458110137
Initial Values: 0.0001 6.0 1.0 0.1 5.0
RMS error:  280796383417.8809
MAE error:  4.500211895911299
Initial Values: 0.0001 6.0 1.0 2.5500000000000003 0.01
RMS error:  130204324.0983177
MAE error:  2.8994940043045863
Initial Values: 0.0001 6.0 1.0 2.5500000000000003 2.505
RMS error:  14055743177.035833
MAE error:  3.9258240602420504
Initial Values: 0.0001 6.0 1.0 2.5500000000000003 5.0
RMS error:  1596339013326.5251
MAE error:  4.956045479719608
Initial Values: 0.0001 6.0 1.0 5.0 0.01
RMS error:  291306870.9700975
MAE error:  3.0737705423805615
Initial Values: 0.0001 6.0 1.0 5.0 2.505
RMS error:  17984662726.953117
MAE error:  3.97439420816353
Initial Values: 0.0001 6.0 1.0 5.0 5.0
RMS error:  2123645800103.6082
MAE error:  5.0209539217341055
Initial Values: 0.0001 10.0 0.01 0.1 0.01
RMS error:  27742980939478.56
MAE error:  5.335314614104841
Initial Values: 0.0001 10.0 0.01 0.1 2.505
RMS error:  394468999250020.44
MAE error:  5.916639747280272
Initial Values: 0.0001 10.0 0.01 0.1 5.0
RMS error:  9.434588867323826e+16
MAE error:  7.115723923883242
Initial Values: 0.0001 10.0 0.01 2.5500000000000003 0.01
RMS error:  6132404640822.123
MAE error:  5.0025215871864726
Initial Values: 0.0001 10.0 0.01 2.5500000000000003 2.505
RMS error:  404762148349781.2
MAE error:  5.920396796593887
Initial Values: 0.0001 10.0 0.01 2.5500000000000003 5.0
RMS error:  1.0151986866296429e+17
MAE error:  7.129371781125149
```

```
Initial Values: 0.0001 10.0 0.01 5.0 0.01
RMS error:  19496624976954.082
MAE error:  5.2560956581069425
Initial Values: 0.0001 10.0 0.01 5.0 2.505
RMS error:  2184704238309112.5
MAE error:  6.285680393624615
Initial Values: 0.0001 10.0 0.01 5.0 5.0
RMS error:  7.260766281963994e+16
MAE error:  7.058637160851641
Initial Values: 0.0001 10.0 0.505 0.1 0.01
RMS error:  83800313232422.62
MAE error:  5.596329997587519
Initial Values: 0.0001 10.0 0.505 0.1 2.505
RMS error:  986253005328679.5
MAE error:  6.136994184096675
Initial Values: 0.0001 10.0 0.505 0.1 5.0
RMS error:  5.280970216599597e+16
MAE error:  7.012382286317724
Initial Values: 0.0001 10.0 0.505 2.5500000000000003 0.01
RMS error:  225719648719113.5
MAE error:  5.8007604826401415
Initial Values: 0.0001 10.0 0.505 2.5500000000000003 2.505
RMS error:  9574955432082176.0
MAE error:  6.621638871666846
Initial Values: 0.0001 10.0 0.505 2.5500000000000003 5.0
RMS error:  5.4451303017806586e+17
MAE error:  7.51123771592953
Initial Values: 0.0001 10.0 0.505 5.0 0.01
RMS error:  32629236111733.57
MAE error:  5.373222398442501
Initial Values: 0.0001 10.0 0.505 5.0 2.505
RMS error:  5373233119168184.0
MAE error:  6.489936426025363
Initial Values: 0.0001 10.0 0.505 5.0 5.0
RMS error:  5.998788194996442e+17
MAE error:  7.52270495976749
Initial Values: 0.0001 10.0 1.0 0.1 0.01
RMS error:  212231141461640.4
MAE error:  5.754504137546827
Initial Values: 0.0001 10.0 1.0 0.1 2.505
RMS error:  2.1789148897631016e+16
MAE error:  6.766874138330629
Initial Values: 0.0001 10.0 1.0 0.1 5.0
RMS error:  4.545245662193342e+17
MAE error:  7.437471819420062
Initial Values: 0.0001 10.0 1.0 2.5500000000000003 0.01
RMS error:  1200858925528245.5
MAE error:  6.172727913143989
Initial Values: 0.0001 10.0 1.0 2.5500000000000003 2.505
RMS error:  5.933426705274233e+16
MAE error:  7.034546402081207
Initial Values: 0.0001 10.0 1.0 2.5500000000000003 5.0
RMS error:  1.434957178774562e+18
MAE error:  7.732160723248024
Initial Values: 0.0001 10.0 1.0 5.0 0.01
RMS error:  1923563826434554.5
MAE error:  6.273842151204909
Initial Values: 0.0001 10.0 1.0 5.0 2.505
RMS error:  3.5829629006447196e+16
MAE error:  6.912082719747738
```

```
Initial Values: 0.0001 10.0 1.0 5.0 5.0
RMS error:  1.2392485342279428e+19
MAE error:  8.191125042510075
Initial Values: 0.050050000000000004 2.0 0.01 0.1 0.01
RMS error:  20.806268798739872
MAE error:  0.036011030874359966
Initial Values: 0.050050000000000004 2.0 0.01 0.1 2.505
RMS error:  7135.325117402975
MAE error:  0.9139521497807783
Initial Values: 0.050050000000000004 2.0 0.01 0.1 5.0
RMS error:  349285.52719591476
MAE error:  1.7646315690239134
Initial Values: 0.050050000000000004 2.0 0.01 2.5500000000000003 0.01
RMS error:  22.408850157431445
MAE error:  0.039024808048373875
Initial Values: 0.050050000000000004 2.0 0.01 2.5500000000000003 2.505
RMS error:  6333.756702096748
MAE error:  0.8889907366920785
Initial Values: 0.050050000000000004 2.0 0.01 2.5500000000000003 5.0
RMS error:  444173.25892357575
MAE error:  1.8153265987972105
Initial Values: 0.050050000000000004 2.0 0.01 5.0 0.01
RMS error:  21.511740098131014
MAE error:  0.03691687466197201
Initial Values: 0.050050000000000004 2.0 0.01 5.0 2.505
RMS error:  7242.748206048893
MAE error:  0.9167763761879472
Initial Values: 0.050050000000000004 2.0 0.01 5.0 5.0
RMS error:  351390.10500373895
MAE error:  1.7659069295256982
Initial Values: 0.050050000000000004 2.0 0.505 0.1 0.01
RMS error:  19.438454272609853
MAE error:  0.03731925453835871
Initial Values: 0.050050000000000004 2.0 0.505 0.1 2.505
RMS error:  18753.854029051316
MAE error:  1.0882062499310339
Initial Values: 0.050050000000000004 2.0 0.505 0.1 5.0
RMS error:  1156511.026696419
MAE error:  1.990200645764889
Initial Values: 0.050050000000000004 2.0 0.505 2.5500000000000003 0.01
RMS error:  23.963240396453322
MAE error:  0.046849819375666346
Initial Values: 0.050050000000000004 2.0 0.505 2.5500000000000003 2.505
RMS error:  45059.47359464657
MAE error:  1.3168730967991458
Initial Values: 0.050050000000000004 2.0 0.505 2.5500000000000003 5.0
RMS error:  2713243.728184087
MAE error:  2.215319395168548
Initial Values: 0.050050000000000004 2.0 0.505 5.0 0.01
RMS error:  23.156691891241795
MAE error:  0.04502062242878595
Initial Values: 0.050050000000000004 2.0 0.505 5.0 2.505
RMS error:  57958.98944600724
MAE error:  1.3721591012103478
Initial Values: 0.050050000000000004 2.0 0.505 5.0 5.0
RMS error:  4263256.661841901
MAE error:  2.3124692164990788
Initial Values: 0.050050000000000004 2.0 1.0 0.1 0.01
RMS error:  144.3510815514602
MAE error:  0.1902011889160834
```

```
Initial Values: 0.050050000000000004 2.0 1.0 0.1 2.505
RMS error:  88620.45701469232
MAE error:  1.335925832925667
Initial Values: 0.050050000000000004 2.0 1.0 0.1 5.0
RMS error:  4739208.25044559
MAE error:  2.206626212205463
Initial Values: 0.050050000000000004 2.0 1.0 2.5500000000000003 0.01
RMS error:  4498.519552250507
MAE error:  0.8169158416159564
Initial Values: 0.050050000000000004 2.0 1.0 2.5500000000000003 2.505
RMS error:  261097.60512504145
MAE error:  1.7025810246075177
Initial Values: 0.050050000000000004 2.0 1.0 2.5500000000000003 5.0
RMS error:  18462793.37560359
MAE error:  2.6349412533634076
Initial Values: 0.050050000000000004 2.0 1.0 5.0 0.01
RMS error:  7536.269948231299
MAE error:  0.9290267541056624
Initial Values: 0.050050000000000004 2.0 1.0 5.0 2.505
RMS error:  607849.1560781524
MAE error:  1.8862949215201723
Initial Values: 0.050050000000000004 2.0 1.0 5.0 5.0
RMS error:  34474976.52100823
MAE error:  2.7720812568629616
Initial Values: 0.050050000000000004 6.0 0.01 0.1 0.01
RMS error:  396959995.0121275
MAE error:  3.1288837077656737
Initial Values: 0.050050000000000004 6.0 0.01 0.1 2.505
RMS error:  20410456180.30774
MAE error:  3.996306750740247
Initial Values: 0.050050000000000004 6.0 0.01 0.1 5.0
RMS error:  1672136369385.7441
MAE error:  4.9583474784215715
Initial Values: 0.050050000000000004 6.0 0.01 2.5500000000000003 0.01
RMS error:  321050604.20186865
MAE error:  3.084871251952402
Initial Values: 0.050050000000000004 6.0 0.01 2.5500000000000003 2.505
RMS error:  28727123941.783913
MAE error:  4.066085958981495
Initial Values: 0.050050000000000004 6.0 0.01 2.5500000000000003 5.0
RMS error:  1317686261152.9565
MAE error:  4.909152598557161
Initial Values: 0.050050000000000004 6.0 0.01 5.0 0.01
RMS error:  362426342.9638644
MAE error:  3.109595721704148
Initial Values: 0.050050000000000004 6.0 0.01 5.0 2.505
RMS error:  17614016195.821037
MAE error:  3.9650463127729374
Initial Values: 0.050050000000000004 6.0 0.01 5.0 5.0
RMS error:  1372098715844.6929
MAE error:  4.919628654757824
Initial Values: 0.050050000000000004 6.0 0.505 0.1 0.01
RMS error:  926915352.2567861
MAE error:  3.3117965496705355
Initial Values: 0.050050000000000004 6.0 0.505 0.1 2.505
RMS error:  51480278087.85948
MAE error:  4.195195549696829
Initial Values: 0.050050000000000004 6.0 0.505 0.1 5.0
RMS error:  4458462862656.687
MAE error:  5.16937059607671
```

```
Initial Values: 0.050050000000000004 6.0 0.505 2.5500000000000003 0.01
RMS error:  2386879587.9361925
MAE error:  3.5306836112736826
Initial Values: 0.050050000000000004 6.0 0.505 2.5500000000000003 2.505
RMS error:  174991005672.89563
MAE error:  4.4711355556207275
Initial Values: 0.050050000000000004 6.0 0.505 2.5500000000000003 5.0
RMS error:  10019629412220.248
MAE error:  5.359542266328616
Initial Values: 0.050050000000000004 6.0 0.505 5.0 0.01
RMS error:  3302013063.1381073
MAE error:  3.5981538979514247
Initial Values: 0.050050000000000004 6.0 0.505 5.0 2.505
RMS error:  211134037756.2951
MAE error:  4.510640907619779
Initial Values: 0.050050000000000004 6.0 0.505 5.0 5.0
RMS error:  15123398272040.959
MAE error:  5.44433023334516
Initial Values: 0.050050000000000004 6.0 1.0 0.1 0.01
RMS error:  3111781129.0541444
MAE error:  3.5090892749561142
Initial Values: 0.050050000000000004 6.0 1.0 0.1 2.505
RMS error:  224337182462.763
MAE error:  4.449799827099682
Initial Values: 0.050050000000000004 6.0 1.0 0.1 5.0
RMS error:  15096329044851.16
MAE error:  5.370378717281621
Initial Values: 0.050050000000000004 6.0 1.0 2.5500000000000003 0.01
RMS error:  15847761090.397543
MAE error:  3.9507188687374057
Initial Values: 0.050050000000000004 6.0 1.0 2.5500000000000003 2.505
RMS error:  1269765136711.2407
MAE error:  4.908716669912385
Initial Values: 0.050050000000000004 6.0 1.0 2.5500000000000003 5.0
RMS error:  61214451613248.42
MAE error:  5.7623848256927594
Initial Values: 0.050050000000000004 6.0 1.0 5.0 0.01
RMS error:  31799750281.413593
MAE error:  4.0976905035210915
Initial Values: 0.050050000000000004 6.0 1.0 5.0 2.505
RMS error:  2411506537241.457
MAE error:  5.044116268097772
Initial Values: 0.050050000000000004 6.0 1.0 5.0 5.0
RMS error:  134918787257389.14
MAE error:  5.929807064437801
Initial Values: 0.050050000000000004 10.0 0.01 0.1 0.01
RMS error:  1318748062372349.2
MAE error:  6.1827573717170115
Initial Values: 0.050050000000000004 10.0 0.01 0.1 2.505
RMS error:  1.1191111369322027e+17
MAE error:  7.153097987901699
Initial Values: 0.050050000000000004 10.0 0.01 0.1 5.0
RMS error:  7.34307117894344e+18
MAE error:  8.06996258731159
Initial Values: 0.050050000000000004 10.0 0.01 2.5500000000000003 0.01
RMS error:  1777717645245489.8
MAE error:  6.242597526155764
Initial Values: 0.050050000000000004 10.0 0.01 2.5500000000000003 2.505
RMS error:  1.0190514022795416e+17
MAE error:  7.132834579477878
```

```
Initial Values: 0.050050000000000004 10.0 0.01 2.5500000000000003 5.0
RMS error:  1.031537072120843e+19
MAE error:  8.140686567488526
Initial Values: 0.050050000000000004 10.0 0.01 5.0 0.01
RMS error:  1519521234618931.5
MAE error:  6.211025829634644
Initial Values: 0.050050000000000004 10.0 0.01 5.0 2.505
RMS error:  1.185564035228397e+17
MAE error:  7.164135130999174
Initial Values: 0.050050000000000004 10.0 0.01 5.0 5.0
RMS error:  6.133096043410829e+18
MAE error:  8.03238327738611
Initial Values: 0.050050000000000004 10.0 0.505 0.1 0.01
RMS error:  3656981655039604.5
MAE error:  6.425977009312778
Initial Values: 0.050050000000000004 10.0 0.505 0.1 2.505
RMS error:  2.8124218869173613e+17
MAE error:  7.375913867903706
Initial Values: 0.050050000000000004 10.0 0.505 0.1 5.0
RMS error:  1.6348973084800823e+19
MAE error:  8.26647598081898
Initial Values: 0.050050000000000004 10.0 0.505 2.5500000000000003 0.01
RMS error:  1.2446541810456514e+16
MAE error:  6.678773772721829
Initial Values: 0.050050000000000004 10.0 0.505 2.5500000000000003 2.505
RMS error:  9.25122735738012e+17
MAE error:  7.621970019510306
Initial Values: 0.050050000000000004 10.0 0.505 2.5500000000000003 5.0
RMS error:  5.2851568674489524e+19
MAE error:  8.510947618714823
Initial Values: 0.050050000000000004 10.0 0.505 5.0 0.01
RMS error:  1.29774686053434e+16
MAE error:  6.688540670414365
Initial Values: 0.050050000000000004 10.0 0.505 5.0 2.505
RMS error:  1.002695025661618e+18
MAE error:  7.639334265288689
Initial Values: 0.050050000000000004 10.0 0.505 5.0 5.0
RMS error:  7.115444559938036e+19
MAE error:  8.572790833735612
Initial Values: 0.050050000000000004 10.0 1.0 0.1 0.01
RMS error:  1.0323919103953334e+16
MAE error:  6.6082445437696595
Initial Values: 0.050050000000000004 10.0 1.0 0.1 2.505
RMS error:  7.711970548750002e+17
MAE error:  7.551564947512825
Initial Values: 0.050050000000000004 10.0 1.0 0.1 5.0
RMS error:  6.415650260590101e+19
MAE error:  8.516586122092138
Initial Values: 0.050050000000000004 10.0 1.0 2.5500000000000003 0.01
RMS error:  8.136349263596864e+16
MAE error:  7.10065306954263
Initial Values: 0.050050000000000004 10.0 1.0 2.5500000000000003 2.505
RMS error:  5.502233325420639e+18
MAE error:  8.023908011681772
Initial Values: 0.050050000000000004 10.0 1.0 2.5500000000000003 5.0
RMS error:  3.875690693269923e+20
MAE error:  8.954956193977202
Initial Values: 0.050050000000000004 10.0 1.0 5.0 0.01
RMS error:  1.5036985039230563e+17
MAE error:  7.22760109934372
```

```
Initial Values: 0.050050000000000004 10.0 1.0 5.0 2.505
RMS error:  9.291084986268424e+18
MAE error:  8.133361264689428
Initial Values: 0.050050000000000004 10.0 1.0 5.0 5.0
RMS error:  6.419265948426336e+20
MAE error:  9.060684509510654
Initial Values: 0.1 2.0 0.01 0.1 0.01
RMS error:  21.26931728150601
MAE error:  0.041117229950281296
Initial Values: 0.1 2.0 0.01 0.1 2.505
RMS error:  13878.720216285315
MAE error:  1.0583344349230077
Initial Values: 0.1 2.0 0.01 0.1 5.0
RMS error:  926136.4288220983
MAE error:  1.975823966732342
Initial Values: 0.1 2.0 0.01 2.5500000000000003 0.01
RMS error:  21.723815967658954
MAE error:  0.041622658204274965
Initial Values: 0.1 2.0 0.01 2.5500000000000003 2.505
RMS error:  14105.973756528709
MAE error:  1.0615196279444779
Initial Values: 0.1 2.0 0.01 2.5500000000000003 5.0
RMS error:  996208.302542011
MAE error:  1.991774380859214
Initial Values: 0.1 2.0 0.01 5.0 0.01
RMS error:  22.378135439576265
MAE error:  0.04250939533728469
Initial Values: 0.1 2.0 0.01 5.0 2.505
RMS error:  14804.940819690375
MAE error:  1.0713988672875323
Initial Values: 0.1 2.0 0.01 5.0 5.0
RMS error:  712629.8761208903
MAE error:  1.920620284221312
Initial Values: 0.1 2.0 0.505 0.1 0.01
RMS error:  22.046048796972194
MAE error:  0.040461145713433344
Initial Values: 0.1 2.0 0.505 0.1 2.505
RMS error:  39600.038811565464
MAE error:  1.251609627712056
Initial Values: 0.1 2.0 0.505 0.1 5.0
RMS error:  2750538.7617269163
MAE error:  2.1787733460704284
Initial Values: 0.1 2.0 0.505 2.5500000000000003 0.01
RMS error:  25.523182639168844
MAE error:  0.04502624605364218
Initial Values: 0.1 2.0 0.505 2.5500000000000003 2.505
RMS error:  107384.31805501576
MAE error:  1.506070171681517
Initial Values: 0.1 2.0 0.505 2.5500000000000003 5.0
RMS error:  5399000.652712354
MAE error:  2.3656446911379776
Initial Values: 0.1 2.0 0.505 5.0 0.01
RMS error:  1711.0700657498992
MAE error:  0.6145926506447507
Initial Values: 0.1 2.0 0.505 5.0 2.505
RMS error:  130916.82222759789
MAE error:  1.5493743779570404
Initial Values: 0.1 2.0 0.505 5.0 5.0
RMS error:  6559540.666642306
MAE error:  2.408959457035186
```

```
Initial Values: 0.1 2.0 1.0 0.1 0.01
RMS error:  85.59149331048181
MAE error:  0.14359157695320987
Initial Values: 0.1 2.0 1.0 0.1 2.505
RMS error:  148530.91117151067
MAE error:  1.4534495569148818
Initial Values: 0.1 2.0 1.0 0.1 5.0
RMS error:  9870149.437143194
MAE error:  2.3687654006823804
Initial Values: 0.1 2.0 1.0 2.5500000000000003 0.01
RMS error:  10448.826880637362
MAE error:  0.9981331382649435
Initial Values: 0.1 2.0 1.0 2.5500000000000003 2.505
RMS error:  573205.9346066385
MAE error:  1.8742954194157697
Initial Values: 0.1 2.0 1.0 2.5500000000000003 5.0
RMS error:  43394766.8417291
MAE error:  2.8211251435576576
Initial Values: 0.1 2.0 1.0 5.0 0.01
RMS error:  14999.107735201393
MAE error:  1.0783043631970255
Initial Values: 0.1 2.0 1.0 5.0 2.505
RMS error:  1150270.13999211
MAE error:  2.026743294557982
Initial Values: 0.1 2.0 1.0 5.0 5.0
RMS error:  76950423.18822078
MAE error:  2.9473505536391826
Initial Values: 0.1 6.0 0.01 0.1 0.01
RMS error:  611042947.1811149
MAE error:  3.2269717734777834
Initial Values: 0.1 6.0 0.01 0.1 2.505
RMS error:  37496468867.394295
MAE error:  4.129959877286396
Initial Values: 0.1 6.0 0.01 0.1 5.0
RMS error:  3490842804328.334
MAE error:  5.119735143163064
Initial Values: 0.1 6.0 0.01 2.5500000000000003 0.01
RMS error:  586057122.8079185
MAE error:  3.2180109726039783
Initial Values: 0.1 6.0 0.01 2.5500000000000003 2.505
RMS error:  57551375028.57188
MAE error:  4.218183371458044
Initial Values: 0.1 6.0 0.01 2.5500000000000003 5.0
RMS error:  2860670640018.8394
MAE error:  5.077165924135671
Initial Values: 0.1 6.0 0.01 5.0 0.01
RMS error:  606411503.2592015
MAE error:  3.2248626082476024
Initial Values: 0.1 6.0 0.01 5.0 2.505
RMS error:  45868067468.39552
MAE error:  4.17114858093589
Initial Values: 0.1 6.0 0.01 5.0 5.0
RMS error:  2827238771809.6177
MAE error:  5.074933112857462
Initial Values: 0.1 6.0 0.505 0.1 0.01
RMS error:  1654571649.346833
MAE error:  3.440630117006358
Initial Values: 0.1 6.0 0.505 0.1 2.505
RMS error:  112093223073.46657
MAE error:  4.363842286895302
```

```
Initial Values: 0.1 6.0 0.505 0.1 5.0
RMS error:  8802964529483.244
MAE error:  5.317894069555686
Initial Values: 0.1 6.0 0.505 2.5500000000000003 0.01
RMS error:  4953300061.550335
MAE error:  3.6896349609160146
Initial Values: 0.1 6.0 0.505 2.5500000000000003 2.505
RMS error:  338553951189.74664
MAE error:  4.615317512373374
Initial Values: 0.1 6.0 0.505 2.5500000000000003 5.0
RMS error:  18573880281998.05
MAE error:  5.497655076087361
Initial Values: 0.1 6.0 0.505 5.0 0.01
RMS error:  7673373423.328399
MAE error:  3.7807550893072306
Initial Values: 0.1 6.0 0.505 5.0 2.505
RMS error:  417778629221.2956
MAE error:  4.661164366520735
Initial Values: 0.1 6.0 0.505 5.0 5.0
RMS error:  25633289246865.082
MAE error:  5.563028650894158
Initial Values: 0.1 6.0 1.0 0.1 0.01
RMS error:  5555215368.952954
MAE error:  3.639980336081046
Initial Values: 0.1 6.0 1.0 0.1 2.505
RMS error:  473988588178.81366
MAE error:  4.610400676765747
Initial Values: 0.1 6.0 1.0 0.1 5.0
RMS error:  25420286508143.8
MAE error:  5.487480843151936
Initial Values: 0.1 6.0 1.0 2.5500000000000003 0.01
RMS error:  27621320409.10719
MAE error:  4.074417488924203
Initial Values: 0.1 6.0 1.0 2.5500000000000003 2.505
RMS error:  1973831901370.0007
MAE error:  5.010445432689376
Initial Values: 0.1 6.0 1.0 2.5500000000000003 5.0
RMS error:  150930562022536.94
MAE error:  5.959591405399256
Initial Values: 0.1 6.0 1.0 5.0 0.01
RMS error:  64884604774.81384
MAE error:  4.253682164699096
Initial Values: 0.1 6.0 1.0 5.0 2.505
RMS error:  3412489009479.6655
MAE error:  5.126547050424079
Initial Values: 0.1 6.0 1.0 5.0 5.0
RMS error:  248887681661571.44
MAE error:  6.065393189923931
Initial Values: 0.1 10.0 0.01 0.1 0.01
RMS error:  2445513928551966.5
MAE error:  6.319673710574861
Initial Values: 0.1 10.0 0.01 0.1 2.505
RMS error:  2.0038508618125926e+17
MAE error:  7.2826295735270525
Initial Values: 0.1 10.0 0.01 0.1 5.0
RMS error:  1.2737580192811426e+19
MAE error:  8.19112166578485
Initial Values: 0.1 10.0 0.01 2.5500000000000003 0.01
RMS error:  2881551204660823.5
MAE error:  6.3518335058204265
```

```
Initial Values: 0.1 10.0 0.01 2.5500000000000003 2.505
RMS error:  2.6609835340287072e+17
MAE error:  7.339003967899555
Initial Values: 0.1 10.0 0.01 2.5500000000000003 5.0
RMS error:  1.462443994026959e+19
MAE error:  8.22056616367393
Initial Values: 0.1 10.0 0.01 5.0 0.01
RMS error:  3772932085807046.5
MAE error:  6.4066763598815495
Initial Values: 0.1 10.0 0.01 5.0 2.505
RMS error:  2.1563927685984982e+17
MAE error:  7.296334051244379
Initial Values: 0.1 10.0 0.01 5.0 5.0
RMS error:  1.8406790594834622e+19
MAE error:  8.26679380651499
Initial Values: 0.1 10.0 0.505 0.1 0.01
RMS error:  6205867355674967.0
MAE error:  6.5439726797001505
Initial Values: 0.1 10.0 0.505 0.1 2.505
RMS error:  5.799766995492918e+17
MAE error:  7.533345023676305
Initial Values: 0.1 10.0 0.505 0.1 5.0
RMS error:  2.8320291541092524e+19
MAE error:  8.388793819416119
Initial Values: 0.1 10.0 0.505 2.5500000000000003 0.01
RMS error:  1.8434959402842868e+16
MAE error:  6.769918001083191
Initial Values: 0.1 10.0 0.505 2.5500000000000003 2.505
RMS error:  1.5493590184885335e+18
MAE error:  7.738949358595926
Initial Values: 0.1 10.0 0.505 2.5500000000000003 5.0
RMS error:  1.059648658344568e+20
MAE error:  8.663800733282828
Initial Values: 0.1 10.0 0.505 5.0 0.01
RMS error:  2.9558770199614804e+16
MAE error:  6.866689068140594
Initial Values: 0.1 10.0 0.505 5.0 2.505
RMS error:  1.9291964413308895e+18
MAE error:  7.783564264556716
Initial Values: 0.1 10.0 0.505 5.0 5.0
RMS error:  1.785696664543258e+20
MAE error:  8.771063197924187
Initial Values: 0.1 10.0 1.0 0.1 0.01
RMS error:  2.5223291744776812e+16
MAE error:  6.800492553766305
Initial Values: 0.1 10.0 1.0 0.1 2.505
RMS error:  2.0405702133328205e+18
MAE error:  7.761125353857297
Initial Values: 0.1 10.0 1.0 0.1 5.0
RMS error:  1.03134082058723e+20
MAE error:  8.623188973090198
Initial Values: 0.1 10.0 1.0 2.5500000000000003 0.01
RMS error:  1.4093517171720256e+17
MAE error:  7.222417925040678
Initial Values: 0.1 10.0 1.0 2.5500000000000003 2.505
RMS error:  9.060401162874626e+18
MAE error:  8.13700723820551
Initial Values: 0.1 10.0 1.0 2.5500000000000003 5.0
RMS error:  7.221252392694566e+20
MAE error:  9.09448246724651
```

```
Initial Values: 0.1 10.0 1.0 5.0 0.01
RMS error:  2.5032643484888893e+17
MAE error:  7.340642178800743
Initial Values: 0.1 10.0 1.0 5.0 2.505
RMS error:  1.7586406696303473e+19
MAE error:  8.274256015351453
Initial Values: 0.1 10.0 1.0 5.0 5.0
RMS error:  1.2555599499102705e+21
MAE error:  9.207076358691534

Top 5 Results:
Set       n1i       n2i       n3i       n4i       n5i       p         NGEN
RMSE      MAE
13        0.00010   2.000     0.505     2.550     0.010     0.09      3000
14.62     0.0265
22        0.00010   2.000     1.000     2.550     0.010     0.09      3000
15.16     0.0274
25        0.00010   2.000     1.000     5.000     0.010     0.09      3000
16.19     0.0297
11        0.00010   2.000     0.505     0.100     2.505     0.09      3000
17.17     0.0311
14        0.00010   2.000     0.505     2.550     2.505     0.09      3000
17.72     0.0323

Top 5 Results:
Set       n1i       n2i       n3i       n4i       n5i       p         NGEN
RMSE      MAE
13        0.00010   2.000     0.505     2.550     0.010     0.09      3000
14.62     0.0265
19        0.00010   2.000     1.000     0.100     0.010     0.09      3000
21.42     0.0265
12        0.00010   2.000     0.505     0.100     5.000     0.09      3000
19.38     0.0271
22        0.00010   2.000     1.000     2.550     0.010     0.09      3000
15.16     0.0274
16        0.00010   2.000     0.505     5.000     0.010     0.09      3000
19.15     0.0285
```

In [ ]:

Task 2.1

Genetic algorithm to determine the set of (UA)e, (UA)c, and α values that best fit the data. The data was divide into a proper training set (about 80%) and a randomly selected validation
set (about 20%).

In [63]:
```
'''>>>>> start CodeP1.3F23
    V.P. Carey ME249, Fall 2023'''

'''Heat pipe performance data for Part 2 of Project 1
   122 data points'''

# version 3 print function
from __future__ import print_function
# seed the pseudorandom number generator
from random import random
from random import seed
# seed random number generator
seed(1)

#import math and numpy packages
import math
import numpy as numpy

%matplotlib inline
# importing the required module
import matplotlib.pyplot as plt

#import copy
from copy import copy, deepcopy

#create array
ydata = []

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]
# this is an array that is essentially a list of lists

#assembling heat pipe performance data array
#=========================================
#store array where rows are data vectors
#[Ta,in(deg C), Ta,out(deg C), qdot(W)]

ydata =     [[ 27.2,     15.9,      31.2]]
ydata.append([ 29.6,     17.1,      37.0])
ydata.append([ 29.5,     18.1,      34.2])
ydata.append([ 30.4,     19.0,      31.])
ydata.append([ 30.4,     20.0,      28.5])
ydata.append([ 30.2,     21.0,      25.7])
ydata.append([ 30.0,     22.1,      22.8])
ydata.append([ 29.8,     22.9,      20.0])
ydata.append([ 29.9,     24.1,      17.1])
ydata.append([ 29.6,     24.9,      14.3])
ydata.append([ 29.5,     26.1,      11.4])
ydata.append([ 30.3,     26.9,       8.6])
```

```
ydata.append([ 32.5,     16.9,      45.7])
ydata.append([ 32.6,     18.1,      42.8])
ydata.append([ 32.6,     19.1,      40.0])
ydata.append([ 32.7,     20.2,      37.2])
ydata.append([ 32.9,     21.1,      34.3])
ydata.append([ 33.0,     22.0,      31.5])
ydata.append([ 32.5,     23.1,      28.7])
ydata.append([ 32.8,     24.1,      25.8])
ydata.append([ 33.0,     24.9,      23.0])
ydata.append([ 32.9,     26.0,      20.1])
ydata.append([ 33.1,     27.0,      17.2])
ydata.append([ 35.7,     17.0,      54.4])
ydata.append([ 35.6,     18.1,      51.6])
ydata.append([ 36.2,     19.1,      48.7])
ydata.append([ 36.3,     20.2,      45.9])
ydata.append([ 36.4,     21.1,      43.1])
ydata.append([ 35.5,     22.2,      40.2])
ydata.append([ 36.2,     22.9,      37.4])
ydata.append([ 36.2,     24.1,      34.5])
ydata.append([ 35.7,     25.0,      31.7])
ydata.append([ 36.0,     26.1,      28.8])
ydata.append([ 35.8,     27.1,      25.9])
ydata.append([ 38.8,     16.9,      63.1])
ydata.append([ 39.4,     18.1,      60.3])
ydata.append([ 38.9,     19.2,      57.5])
ydata.append([ 39.2,     20.1,      54.7])
ydata.append([ 38.5,     21.2,      51.8])
ydata.append([ 39.0,     22.0,      49.0])
ydata.append([ 39.0,     23.1,      46.1])
ydata.append([ 39.3,     24.0,      43.3])
ydata.append([ 39.3,     24.9,      40.4])
ydata.append([ 39.1,     26.2,      37.6])
ydata.append([ 39.1,     27.2,      34.7])
ydata.append([ 42.2,     17.0,      71.9])
ydata.append([ 42.3,     18.1,      69.1])
ydata.append([ 42.4,     18.9,      66.3])
ydata.append([ 41.9,     20.0,      63.5])
ydata.append([ 41.8,     21.0,      60.6])
ydata.append([ 41.8,     22.1,      57.8])
ydata.append([ 42.2,     22.8,      54.9])
ydata.append([ 41.5,     24.2,      52.1])
ydata.append([ 42.3,     25.3,      49.2])
ydata.append([ 41.8,     25.9,      46.3])
ydata.append([ 42.3,     26.8,      43.5])
ydata.append([ 45.2,     17.1,      80.8])
ydata.append([ 44.5,     18.2,      78.0])
ydata.append([ 44.8,     18.8,      75.1])
ydata.append([ 45.2,     20.0,      72.3])
ydata.append([ 44.9,     21.2,      69.5])
ydata.append([ 45.3,     22.1,      66.6])
ydata.append([ 44.5,     22.8,      63.8])
ydata.append([ 44.9,     24.1,      60.9])
ydata.append([ 44.8,     25.2,      58.1])
ydata.append([ 45.3,     26.2,      55.2])
ydata.append([ 44.9,     26.9,      52.3])
ydata.append([ 48.0,     17.2,      89.7])
ydata.append([ 47.7,     18.1,      86.9])
ydata.append([ 48.2,     18.9,      84.0])
ydata.append([ 48.3,     19.9,      81.2])
ydata.append([ 48.0,     20.0,      78.3])
```

```
ydata.append([ 48.4,     22.8,      75.5])
ydata.append([ 48.2,     23.0,      72.7])
ydata.append([ 47.6,     24.0,      69.8])
ydata.append([ 48.2,     25.1,      66.9])
ydata.append([ 48.2,     25.8,      64.1])
ydata.append([ 47.8,     26.9,      61.2])
ydata.append([ 50.5,     17.0,      98.6])
ydata.append([ 50.6,     18.0,      95.8])
ydata.append([ 50.5,     19.1,      93.0])
ydata.append([ 50.9,     20.1,      90.1])
ydata.append([ 51.0,     20.8,      87.3])
ydata.append([ 51.0,     21.9,      84.4])
ydata.append([ 50.5,     23.0,      81.6])
ydata.append([ 51.2,     24.1,      78.7])
ydata.append([ 50.8,     25.2,      75.9])
ydata.append([ 51.2,     25.9,      73.0])
ydata.append([ 51.0,     26.9,      70.1])
ydata.append([ 54.2,     17.1,     107.6])
ydata.append([ 53.5,     18.1,     104.8])
ydata.append([ 54.1,     19.1,     101.9])
ydata.append([ 53.5,     20.1,      99.1])
ydata.append([ 53.5,     21.2,      96.3])
ydata.append([ 54.2,     21.9,      93.4])
ydata.append([ 54.1,     23.2,      90.6])
ydata.append([ 53.8,     24.1,      87.7])
ydata.append([ 53.6,     25.0,      84.8])
ydata.append([ 53.9,     26.0,      82.0])
ydata.append([ 53.7,     26.8,      79.1])
ydata.append([ 57.2,     17.0,     116.7])
ydata.append([ 56.9,     18.1,     113.8])
ydata.append([ 57.0,     19.1,     111.0])
ydata.append([ 56.7,     19.8,     108.1])
ydata.append([ 57.3,     20.9,     105.3])
ydata.append([ 56.8,     21.9,     102.4])
ydata.append([ 57.4,     23.2,      99.6])
ydata.append([ 57.3,     23.8,      96.7])
ydata.append([ 56.6,     25.2,      93.8])
ydata.append([ 57.0,     25.8,      91.0])
ydata.append([ 57.2,     27.0,      88.1])
ydata.append([ 60.2,     16.9,     125.7])
ydata.append([ 59.9,     17.9,     122.9])
ydata.append([ 59.6,     19.1,     120.0])
ydata.append([ 60.2,     20.1,     117.2])
ydata.append([ 60.2,     20.9,     114.3])
ydata.append([ 60.4,     21.8,     111.5])
ydata.append([ 59.1,     22.9,     108.6])
ydata.append([ 59.1,     24.1,     105.8])
ydata.append([ 60.1,     25.2,     102.9])
ydata.append([ 59.1,     25.8,     100.0])
ydata.append([ 60.3,     26.9,      97.1])


# print the data array
#print ('ydata =', ydata)

# j is column, i is row downward for ydata[i][j] - both start at zero
# so it is:  ydata[row][column]

# print the data array
yarray= numpy.array(ydata)
```

```
# ======================================================================
# This section of the code uses a ramdom shuffle to to separate the
# data into to sets: Training and Validation, 80% and 20%, respectively.
# The section also transforms the units of tempearture from Celcius to
# Kelvin.
# ======================================================================
# Shuffle the data randomly
np.random.shuffle(yarray)
# Calculate the split index for training and validation
split_index = int(0.8 * len(yarray))
# Split the data into training and validation sets
t_data = yarray[:split_index]
t_data_k = yarray[:split_index]
v_data = yarray[split_index:]
v_data_k = yarray[split_index:]
# Convert temperature from Celcius to Kelvins
# for calculation uniformity
for i in range (len(t_data)):
    t_data_k[i][1] =  t_data[i][1] + 273
    t_data_k[i][0] = t_data[i][0] + 273
for i in range (len(v_data)):
    v_data_k[i][1] =  v_data[i][1] + 273
    v_data_k[i][0] = v_data[i][0] + 273

# Now, 'training_data' contains 80% of your data, and 'validation_data' contai

#Parameters for Evolution Loop
#set data parameters
ND = len(t_data)       #number of data vectors in array
DI = 3                 #number of data items in vector
NS = len(t_data)       #total number of DNA strands

#end CodeP1.1F23
```

The objective here is to use machine-learning tools to determine the design and optimization of a heat pipe heat exchanger (HPHE) used for cooling electronic components within a cabinet. While a detailed physics-based model can be constructed to account for various factors such as wick flow, heat transport, and fluid properties, a simplified model can often suffice for well-designed heat pipes

Added section:

- For Loop for running multiple sets
- Calculation of RMSE training & validation
- Calculation of MAE validation
- Plot of q' predicted v. data training & validation data
- Results table at the end

In [73]:
```
'''>>>>> start CodeP1.2F23
    V.P. Carey ME249, Fall 2023'''

'''INITIALIZING PARAMETERS'''
```

```
# ====================================================================
# Chage to original code. Define a list with the goal to evalute
# different initial guesses in a for loop
# ====================================================================
# Define a list of n values, perturbation, and NGEN to evaluate
n_values_to_evaluate = [
#     [(-1, 100.0, 50.00, 10.0, 0.0, 0.0), 0.09, 6000], # original
    [(-1, 3.500, 50.00, 22.0, 0.0, 0.0), 0.09, 6000],
#     [(-1, 10.00, 50.00, 22.0, 0.0, 0.0), 0.09, 6000],
#     [(-1, 3.500, 100.0, 2.00, 0.0, 0.0), 0.09, 6000],
#     [(-1, 3.500, 3.000, 0.00, 0.0, 0.0), 0.09, 6000],
]

# Create an empty list to store summaries
results = []

# ====================================================================
# Change to original code. Section added to initiate For Loop to
# iterate through n number of initial guessses and define their
# parameters
# ====================================================================

# ====================================================================
# Create Variables for the Bolztmann constt
# & the reference convective heat transfer
# coefficient for the fin section
# ====================================================================
Bolt = 5.67 * 10 **-8
hc = 70

# For Loop initiation
for n_values, perturbation, NGEN in n_values_to_evaluate:

    n = []
    ntemp = []
    gen=[0]
    n1avg = [0.0]
    n2avg = [0.0]
    n3avg = [0.0]
    n4avg = [0.0]
    n5avg = [0.0]
    meanAFerr=[0.0]
    aFerrmeanavgn=[0.0]
    rms_dev = [0.0] # Initialize rms_dev
    rms_dev_val = [0.0] # Initialize rms_dev_val

    # Set program parameters
    NGEN = NGEN      # number of generations (steps)
    MFRAC = 0.5   # faction of median threshold
    perturbation = perturbation # pertubation value

    # here the number of data vectors equals the number of DNA strands (or orga
    # they can be different if they are randomly paired to compute Ferr (survi
    for k in range(NGEN-1):
        gen.append(k+1)   # generation array stores the
        meanAFerr.append(0.0)
        aFerrmeanavgn.append(0.0)
        n1avg.append(0.0)
        n2avg.append(0.0)
```

```
        n3avg.append(0.0)
        n4avg.append(0.0)
        n5avg.append(0.0)

    '''guesses for initial solution population'''
    n0i, n1i, n2i, n3i, n4i, n5i = n_values # Assign values from the array

    #- initialize arrays before start of evolution loop EL
    #then - create array of DNA strands n[i] and ntemp[i] with dimesnion NS = 5

    #i initialize array where rows are dna vectors [n0i,n1i,...n5i] with random
    n =  [[-1., n1i+0.001*random(), n2i+0.1*random(), n3i+0.0001*random(), n4i+
    for i in range(ND):
        n.append([-1., n1i+0.0001*random(), n2i+0.001*random(), n3i+0.0001*rand
    #print (n) # uncomment command to print array so it can be checked

    # store also in wtemp
    ntemp = deepcopy(n)

    #initialize Ferr values an dother loop parameters
    #define arrays of Ferr (error) functions
    #individual solution error and absoute error
    Ferr =  [[0.0]]
    #population average solution error and absoute error
    Ferravgn =  [[0.0]]
    aFerr =  [[0.0]]
    aFerravgn =  [[0.0]]

    #store zeros in ND genes
    for i in range(ND-1):
        #individual solution error and absoute error
        Ferr.append([0.0])
        aFerr.append([0.0])
        #population average solution error and absoute error
        Ferravgn.append([0.0])
        aFerravgn.append([0.0])
        #====================================
        # New array rms_dev training data
        rms_dev.append([0.0])
        # New array rms_dev Validation data
        rms_dev_val.append([0.0])
        #====================================

    aFerrmeanavgnMin = 1000000000.0
    # these store the  n values for minimum population average error durng NGEN
    n1min = 0.0
    n2min = 0.0
    n3min = 0.0
    n4min = 0.0
    n5min = 0.0
    aFerrta = 0.0
    # these store the time averaged n values durng from generation 800 to NGEN
    n1min = 0.0
    n1ta = 0.0
    n2ta = 0.0
    n3ta = 0.0
    n4ta = 0.0
    n5ta = 0.0

    '''START OF EVOLUTION LOOP'''
```

```
    # ----------------------------------------
    # k is generation number, NGEN IS TOTAL NUMBER OF GENERATIONS COMPUTED
    for k in range(NGEN):

        '''In this program , the number of organisms (solutions) NS is taken to
        number of data points ND so for each generation, each solution can be c
        data point and all the data is compared in each generation.  The order
        that holds the solution constants is constantly changing due to mating
        is random.'''


        '''CALCULATING ERROR (FITNESS)
        In this program, the absolute error in the logrithm of the physical hea
        used to evaluate fitness.'''

        # Here we calculate error Ferr and absolute error aFerr for each data p
        # for specified n(i), and calculate (mean aFerr) = aFerrmean
        # and (median aFerr) = aFerrmedian for the data collection and specifie
        # Note that the number data points ND equals the number of solutions (c
        #=========================================================================
        '''CALCULATING ERROR (FITNESS)'''
        for i in range(ND):
            # ================================================================
            # New function error equation to accomodate HPHE error function
            # ================================================================
            # Calculate the temperature difference
            diff_t = (t_data[i][0] - t_data[i][1])
            # Calculate the temperature sum
            sum_t = (t_data_k[i][0] + t_data_k[i][1])
            # Calculate the sum of the temperature to the power 3
            t_p3 = (t_data_k[i][0])**3 + (t_data_k[i][1])**3
            # Calculate hr
            hr = n[i][2] * (1+ (1/hc)*n[i][3]* (Bolt)*((sum_t))*(t_p3))
            # Compute error function
            Ferr[i] = t_data[i][2] - (hr * (n[i][1] * diff_t)/(n[i][1] + hr))
            # ================================================================
            # New absolute fractional error equation to accomodate HPHE error f
            # ================================================================
            aFerr[i] = abs(Ferr[i])/abs(t_data[i][2])  #- absolute fractional e

        #-------------
        aFerrmean = numpy.mean(aFerr) #mean error for population for this genei
        meanAFerr[k]= aFerrmean  #store aFerrmean for this generation gen[k]=k
        aFerrmedian = numpy.median(aFerr) #median error for population for this

        '''SELECTION'''
        #pick survivors
        #[2] calculate survival cutoff, set number kept = nkeep = 0
        #=============================================
        clim = MFRAC*aFerrmedian  #cut off limit is a fraction/multiplier MFRAC
        nkeep = 0

        # now check each organism/solution to see if aFerr is less than cut of
        #if yes, store n for next generation population in ntemp, at end nkeep
        #and number of new offspring = NS-nkeep
        #=============================================
        for j in range(NS):  # NS Ferr values, one for each solution in populat
            if (aFerr[j] < clim):
                nkeep = nkeep + 1
                #ntemp[nkeep][0] = n[j][0] = -1 so it is unchanged;
```

```
            ntemp[nkeep-1][1] = n[j][1];
            ntemp[nkeep-1][2] = n[j][2];
            ntemp[nkeep-1][3] = n[j][3];
            ntemp[nkeep-1][4] = n[j][4];
            ntemp[nkeep-1][5] = n[j][5];
        #now have survivors in leading entries in list of ntemp vectors from 1
        #compute number to be added by mating
        nnew = NS - nkeep

        '''MATING'''
        #[4] for nnew new organisms/solutions,
        # randomly pick two survivors, randomly pick DNA (n) from pair for eacl
        #=============================================================
        for j in range(nnew):
            # pick two survivors randomly
            nmate1 = np.random.randint(low=0, high=nkeep+1)
            nmate2 = np.random.randint(low=0, high=nkeep+1)

            #then randomly pick DNA from parents for offspring

            '''here, do not change property ntemp[nkeep+j+1][0], it's always fi
            #if (numpy.random.rand() < 0.5)
            #    ntemp[nkeep+j+1][0] = n[nmate1][0]
            #else
            #    ntemp[nkeep+j+1][0] = n[nmate2][0]

            # ================================================================
            # Change to original code. 0.09 substitute by "perturbation" varial
            # ================================================================

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][1] = n[nmate1][1]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][1] = n[nmate2][1]*(1.+perturbation*2.*(0.5-nur

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][2] = n[nmate1][2]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][2] = n[nmate2][2]*(1.+perturbation*2.*(0.5-nur

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][3] = n[nmate1][3]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][3] = n[nmate2][3]*(1.+perturbation*2.*(0.5-nur
            '''
            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][4] = n[nmate1][4]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][4] = n[nmate2][4]*(1.+perturbation*2.*(0.5-nur

            if (numpy.random.rand() < 0.5):
                ntemp[nkeep+j+1][5] = n[nmate1][5]*(1.+perturbation*2.*(0.5-nur
            else:
                ntemp[nkeep+j+1][5] = n[nmate2][5]*(1.+perturbation*2.*(0.5-nur
            '''
        #============================================
        n = deepcopy(ntemp)    # save ntemp as n for use in next generation (nex


        '''AVERAGING OVER POPULATION AND OVER TIME, FINDING MINIMUM ERROR SET (
```

```
        # [6] calculate n1avg[k], etc., which are average n values for populat
        # at this generation k
        #=========================================
        #initialize average n's to zero and sum contribution of each member of
        n1avg[k] = 0.0;
        n2avg[k] = 0.0;
        n3avg[k] = 0.0;
        n4avg[k] = 0.0;
        n5avg[k] = 0.0;
        for j in range(NS):
            n1avg[k] = n1avg[k] + n[j][1]/NS;
            n2avg[k] = n2avg[k] + n[j][2]/NS;
            n3avg[k] = n3avg[k] + n[j][3]/NS;
            n4avg[k] = n4avg[k] + n[j][4]/NS;
            n5avg[k] = n5avg[k] + n[j][5]/NS;

        # Here we compute aFerravgn[i] = absolute Ferr of logrithm data point i
        # for this solutions generation k
        # aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population o
        #
        #=========================================
        ''' CALCULATING MEAN ERROR FOR POPULATION'''
        for i in range(ND):
            # ============================================================
            # New average function error equation to accomodate HPHE error func
            # ============================================================
            # Calculate the temperature difference
            diff_t = (t_data[i][0] - t_data[i][1])
            # Calculate the temperature sum
            sum_t = (t_data_k[i][0] + t_data_k[i][1])
            # Calculate the sum of the temperature to the power 3
            t_p3 = (t_data_k[i][0])**3 + (t_data_k[i][1])**3
            # Calculate hr
            hr_avg = n2avg[k] * ( 1 + (1/hc) * n3avg[k] * (Bolt)*((sum_t))*(t_p
            # Compute average error function
            Ferravgn[i] = t_data[i][2] - (hr_avg * (n1avg[k] * diff_t)/(n1avg[k
            # ============================================================
            # New average absolute error equation to accomodate HPHE error func
            # ============================================================
            aFerravgn[i] = abs(Ferravgn[i])/abs(t_data[i][2])
        #-------------
        aFerrmeanavgn[k] = numpy.mean(aFerravgn)

        # next, update time average of n valaues in population (n1ta[k], etc.)
        # for generations = k > 800 up to total NGEN
        #=========================================
        aFerrta = aFerrta + aFerrmeanavgn[k]/NGEN
        if (k > 800):
            n1ta = n1ta + n1avg[k]/(NGEN-800)
            n2ta = n2ta + n2avg[k]/(NGEN-800)
            n3ta = n3ta + n3avg[k]/(NGEN-800)
            n4ta = n4ta + n4avg[k]/(NGEN-800)
            n5ta = n5ta + n5avg[k]/(NGEN-800)

        # compare aFerrmeanavgn[k] to previous minimum value and save
        # it and corresponding n(i) values if the value for this generation k i
        #=========================================
        if (aFerrmeanavgn[k] < aFerrmeanavgnMin):
            aFerrmeanavgnMin = aFerrmeanavgn[k]
            n1min = n1avg[k]
```

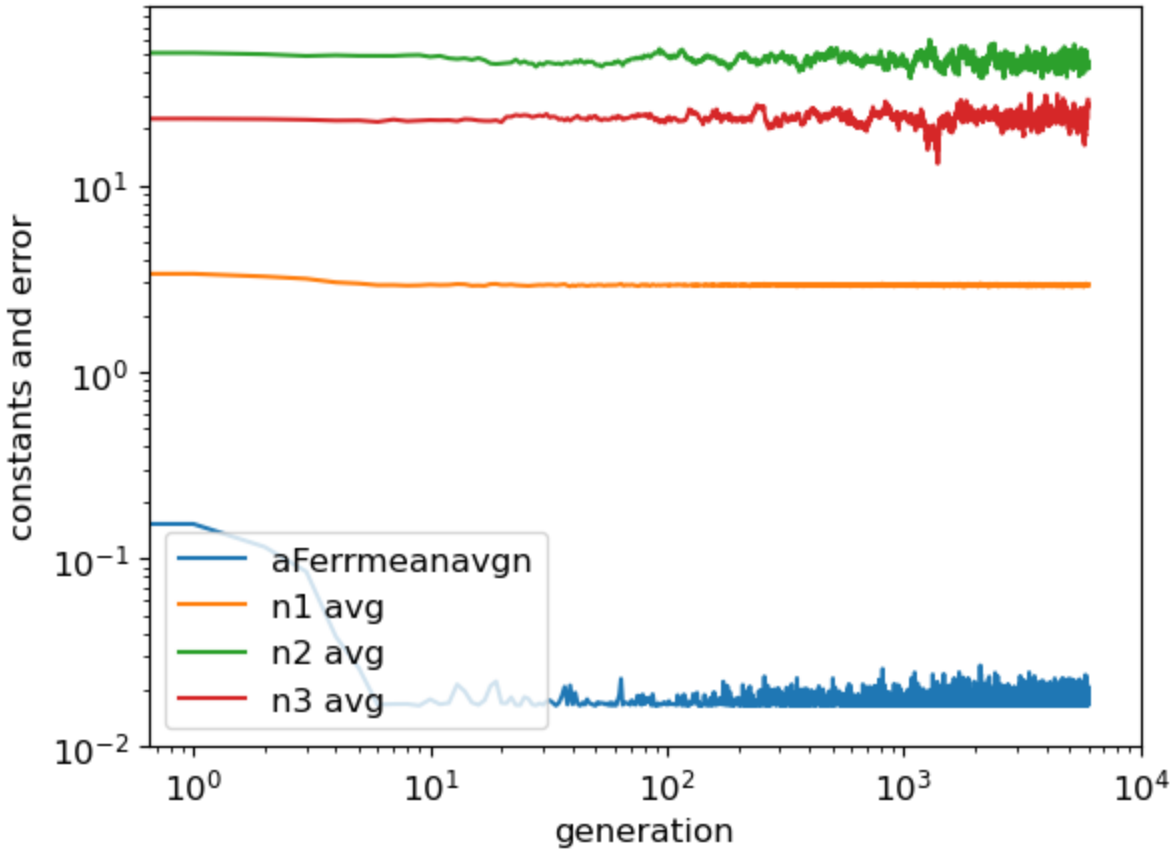
```
            n2min = n2avg[k]
            n3min = n3avg[k]
            n4min = n4avg[k]
            n5min = n5avg[k]

        #print('avg n1-n4:', n1avg[k], n2avg[k], n3avg[k], n4avg[k], aFerrmeana
        #print ('kvalue =', k)
        '''end of evolution loop'''
        # ---------------------------------------------------------------
        # ---------------------------------------------------------------

    # ---------------------------------------------------------------
    #final print and plot of results
    # ---------------------------------------------------------------
#     print('Initial Values:', n1i, n2i, n3i)
#     print('ENDING: pop. avg n1-n3,aFerrmean:', n1avg[k], n2avg[k], n3avg[k],
#     print('MINUMUM:  avg n1-n3,aFerrmeanMin:', n1min, n2min, n3min, aFerrmea
#     print('TIME AVG:    avg n1-n3,aFerrmean:', n1ta, n2ta, n3ta, aFerrta)

    #SETTING UP PLOTS

    # ===================================================================
    # Change to original code. Calculating predicted and RMSE with
    # respect to the data
    # ===================================================================

    # ================================
    #RMS calculation for Traning Data
    # ================================
    #initialize values
    qpppred =  [[0.0]]
    qppdata =  [[0.0]]
    for i in range(ND-1):
        qpppred.append([0.0])
        qppdata.append([0.0])
    # Calculate predicted and data values to plot
    for i in range(ND):
        # Calculate the temperature difference
        diff_t = (t_data[i][0] - t_data[i][1])
        # Calculate the temperature sum
        sum_t = (t_data_k[i][0] + t_data_k[i][1])
        # Calculate the sum of the temperature to the power 2
        t_p3 = (t_data_k[i][0])**3 + (t_data_k[i][1])**3
        # Calculate hr
        hr_min = n2min * (1 + (1/hc) * n3min * (Bolt) * ((sum_t)) * (t_p3))
        # Compute q' predicted
        qpppred[i] = hr_min * n1min * diff_t/(n1min + hr_min)
        qppdata[i] = t_data[i][2]
    #Calculationg RMS btw data and predicted
    for i in range(ND):
        rms_dev[i] = (numpy.array(qppdata[i]) - numpy.array(qpppred[i]))**2
    rms_dev = numpy.sqrt(numpy.sum(rms_dev) / ND)
    print('RMSE training: ', rms_dev)

    # ================================
    #RMS calculation for Validation Data
    # ================================
    #initialize values
    qpppred_val =  [[0.0]]
    qppdata_val =  [[0.0]]
```

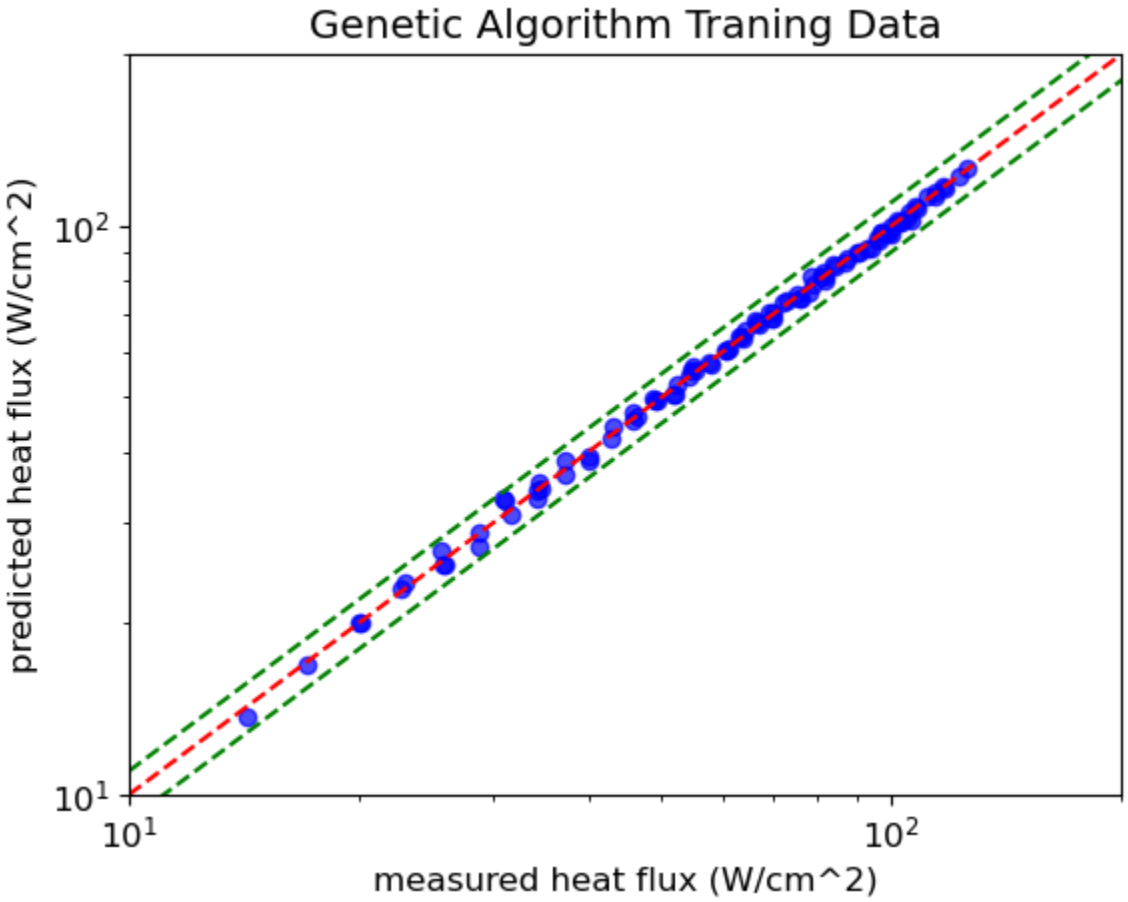
```
    for i in range(len(v_data)-1):
        qpppred_val.append([0.0])
        qppdata_val.append([0.0])
    # Calculate predicted and data values to plot
    for i in range(len(v_data)):
        # Calculate the temperature difference
        diff_t = (v_data[i][0] - v_data[i][1])
        # Calculate the temperature sum
        sum_t = (v_data_k[i][0] + v_data_k[i][1])
        # Calculate the sum of the temperature to the power 2
        t_p3 = (v_data_k[i][0])**3 + (v_data_k[i][1])**3
        # Calculate hr
        hr_min = n2min * (1 + (1/hc) * n3min * (Bolt) * ((sum_t)) * (t_p3))
        # Compute q' predicted
        qpppred_val[i] = hr_min * n1min * diff_t/(n1min + hr_min)
        qppdata_val[i] = v_data[i][2]
    #Calculationg RMS btw data and predicted
    for i in range(len(v_data)):
        rms_dev_val[i] = (numpy.array(qppdata_val[i]) - numpy.array(qpppred_val
    rms_dev_val = numpy.sqrt(numpy.sum(rms_dev_val) / len(v_data))
    print('RMSE Validation: ', rms_dev_val)

    #''' CALCULATING MEAN ERROR FOR POPULATION'''
    Ferravgn_val = [[0.0]]
    aFerravgn_val =  [[0.0]]
    aFerrmeanavgn_val= [[0.0]]
    for i in range(len(v_data)-1):
        Ferravgn_val.append([0.0])
        aFerravgn_val.append([0.0])
        Ferravgn_val.append([0.0])
    for i in range(len(v_data)):
        # Calculate the temperature difference
        diff_t = (v_data[i][0] - v_data[i][1])
        # Calculate the temperature sum
        sum_t = (v_data_k[i][0] + v_data_k[i][1])
        # Calculate the sum of the temperature to the power 3
        t_p3 = (v_data_k[i][0])**3 + (v_data_k[i][1])**3
        # Calculate hr
        hr_avg = n2min * (1 + (1/hc) * n3min * (Bolt) * ((sum_t)) * (t_p3))

        Ferravgn_val[i] = v_data[i][2] - (hr_avg * n1min * diff_t/(n1min + hr_a
        aFerravgn_val[i] = abs(Ferravgn_val[i])/abs(v_data[i][2])
        #-------------
    aFerrmeanavgn_val = numpy.mean(aFerravgn_val)

    # After calculating n1min, n2min, n3min, and rms_dev, create a dictionary t
    iteration_result = {
        'Set': len(results) + 1,
        'n1i': n1i,
        'n2i': n2i,
        'n3i': n3i,
        'n1min': n1min,
        'n2min': n2min,
        'n3min': n3min,
        'n1avg[k]' : n1avg[k],
        'n2avg[k]' : n2avg[k],
        'n3avg[k]' : n3avg[k],
        'p': perturbation,
        'NGEN': NGEN,
        'rms_dev': rms_dev,
```

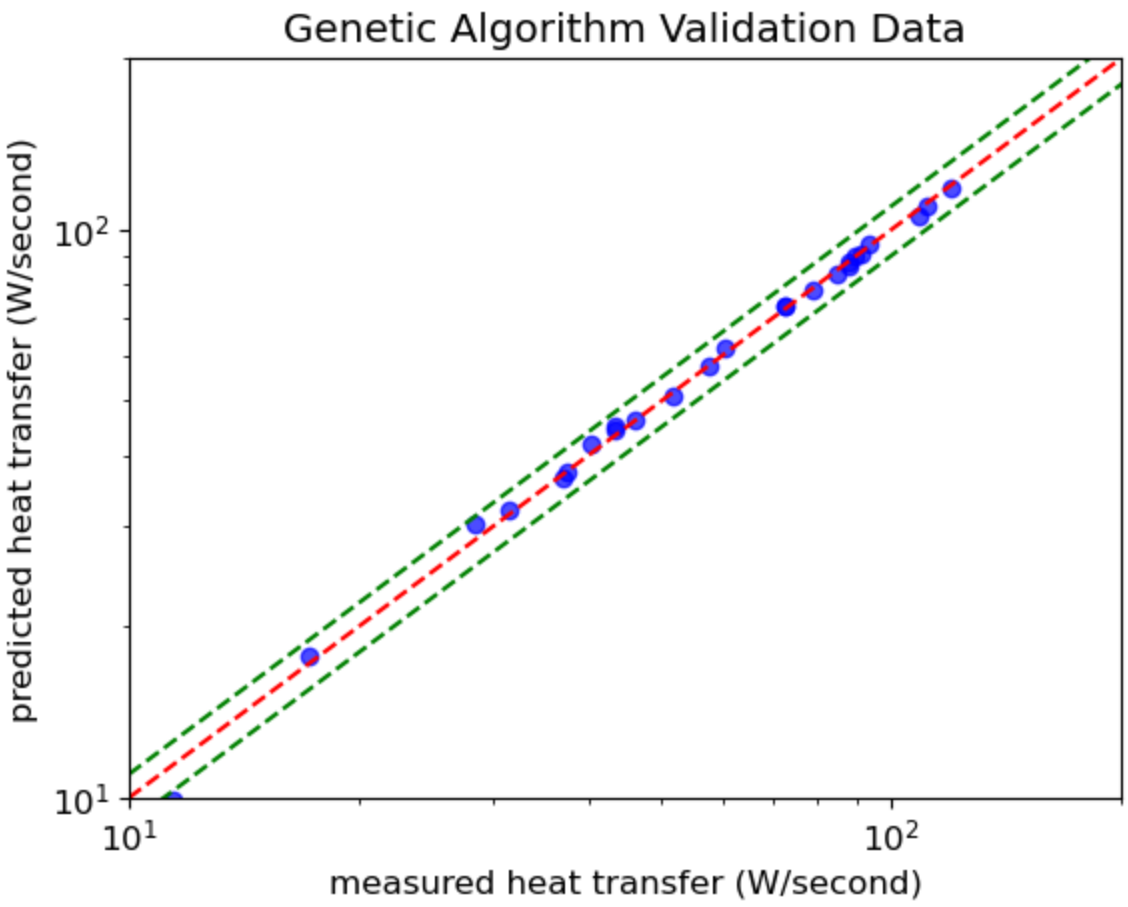
```
        'rms_dev_val': rms_dev_val,
        'MAE_v': aFerrmeanavgn_val,
        'aFerrmeanavgnMin': aFerrmeanavgnMin,
        # Add more values as needed
    }

    results.append(iteration_result)  # Append the summary dictionary to the li
    #========

    # constants evolution plots
    # x axis values are generation number
    # corresponding y axis values are mean absolute population error aFerrmeana
    # plotting the points

    plt.rcParams.update({'font.size': 12})

    # aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of org
    # computed using the mean n values
    plt.plot(gen, aFerrmeanavgn)
    plt.plot(gen, n1avg)
    plt.plot(gen, n2avg)
    plt.plot(gen, n3avg)
    plt.legend(['aFerrmeanavgn', 'n1 avg', 'n2 avg', 'n3 avg'], loc='lower left

    # naming the x axis
    plt.xlabel('generation')
    # naming the y axis
    plt.ylabel('constants and error')
    plt.loglog()
    plt.yticks([0.01,0.1,1.0,10])
    plt.xticks([1,10,100,1000,10000])
    plt.show()


    '''CALCULATE PREDICTED VALUES AND RETRIEVE DATA VALUES TO PLOT'''

    # Adding y=k*x for k=1
    x_values = numpy.logspace(0, 3, 100)   # Adjust the range of x values as ne
    y_values = x_values                     # y = k*x
    y_values1 = x_values + (0.1 * x_values) # y = k*x + 0.1*x (+10% uncertainty
    y_values2 = x_values - (0.1 * x_values) # y = k*x - 0.1*x (-10% uncertainty

    plt.plot(x_values, y_values1, color='green', linestyle='--', label='y = k*x
    plt.plot(x_values, y_values2, color='green', linestyle='--', label='y = k*x
    plt.plot(x_values, y_values, color='red', linestyle='--', label='y = k*x')

    # Plot Heat Flux data vs. predicted
    plt.scatter(qppdata, qpppred, label='Data', color='blue', alpha=0.7)
    plt.title('Genetic Algorithm Traning Data')
    plt.xlabel('measured heat flux (W/cm^2)')
    plt.ylabel('predicted heat flux (W/cm^2)')
    plt.loglog()
    plt.xlim(xmax = 200, xmin = 10)
    plt.ylim(ymax = 200, ymin = 10)

    plt.show()

    # Plotting for Validation Data

    '''CALCULATE PREDICTED VALUES AND RETRIEVE DATA VALUES TO PLOT'''
```

```
    # ================================================================================
    # Change to original code. Adding perfect algorithm prediction reference li
    # and +/-10% uncertainty boundaries
    # ================================================================================
    x_values = numpy.logspace(0, 3, 100)    # Adjust the range of x values as r
    y_values = x_values                     # y = x for a slope of 1
    y_values1 = x_values + (0.1 * x_values)
    y_values2 = x_values - (0.1 * x_values)
    # Plot reference equations
    plt.plot(x_values, y_values1, color='green', linestyle='--', label='Slope 1
    plt.plot(x_values, y_values2, color='green', linestyle='--', label='Slope 1
    plt.plot(x_values, y_values, color='red', linestyle='--', label='Slope 1 Li
    '''CALCULATE PREDICTED VALUES AND RETRIEVE DATA VALUES TO PLOT'''
    # Scatter plot for qppred and qppdata
    plt.scatter(qppdata_val, qpppred_val, label='Data', color='blue', alpha=0.7
    plt.title('Genetic Algorithm Validation Data')
    plt.xlabel('measured heat transfer (W/second)')
    plt.ylabel('predicted heat transfer (W/second)')
    plt.loglog()
    plt.xlim(xmax = 200, xmin = 10)
    plt.ylim(ymax = 200, ymin = 10)

    plt.show()

# Table of Results
print("Initial values for n1, n2, n3:")
print(f"{'n1i':<10}{'n2i':<10}{'n3i':<10}")
for result in results:
    print(f"{result['n1i']:<10.2f}{result['n2i']:<10.2f}{result['n3i']:<10.2f}"

print("Minimum values for n1, n2, n3:")
print(f"{'n1min':<10}{'n2min':<10}{'n3min':<10}")
for result in results:
    print(f"{result['n1min']:<10.2f}{result['n2min']:<10.2f}{result['n3min']:<1

print("Average values for n1, n2, n3:")
print(f"{'n1avg[k]':<10}{'n2avg[k]':<10}{'n3avg[k]':<10}")
for result in results:
    print(f"{result['n1avg[k]']:<10.2f}{result['n2avg[k]']:<10.2f}{result['n3av

print("RMSE & MAE for Training and Validation Data:")
# print(f"{'p':<10}{'NGEN':<10}{'RMSE training':<10}{'MAE trainig':<10}{'RMSE v
print(f"{'p':<10}{'NGEN':<10}{'RMSE t':<10}{'MAE t':<10}{'RMSE v':<10}{'MAE v':
for result in results:
    print(f"{result['p']:<10.2f}{result['NGEN']:<10}{result['rms_dev']:<10.2f}{

# end CodeP1.2F23
```

```
RMSE training:  1.1435633082553414
RMSE Validation:  [1.23139687]
```


constants and error
generation
aFerrmeanavgn
n1 avg
n2 avg
n3 avg


Genetic Algorithm Traning Data
predicted heat flux (W/cm^2)
measured heat flux (W/cm^2)

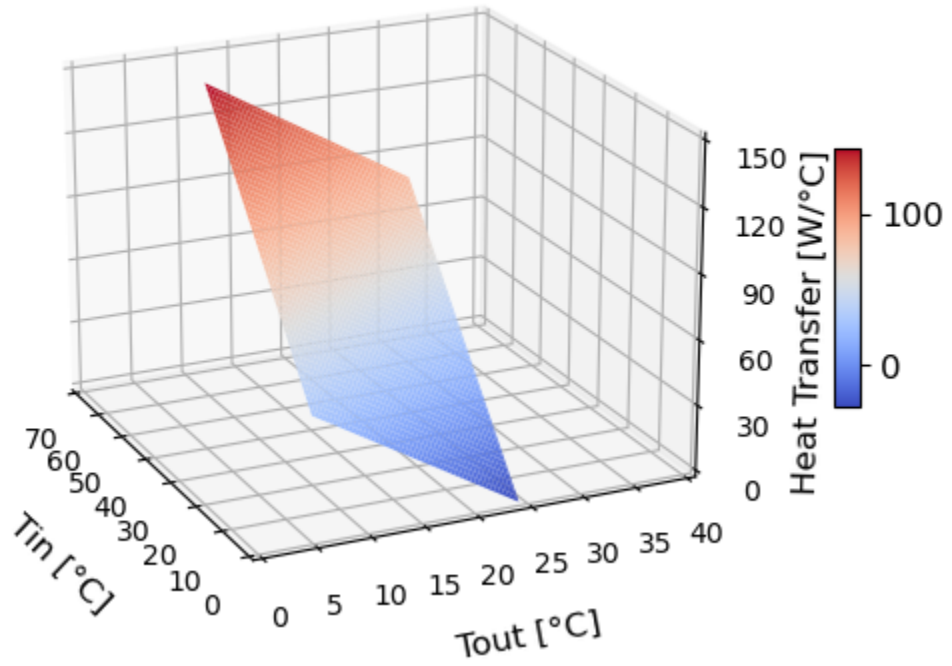



```
Initial values for n1, n2, n3:
n1i       n2i       n3i
3.50      50.00     22.00
Minimum values for n1, n2, n3:
n1min     n2min     n3min
2.91      46.47     18.55
Average values for n1, n2, n3:
n1avg[k]  n2avg[k]  n3avg[k]
2.93      44.68     28.67
RMSE & MAE for Training and Validation Data:
p         NGEN      RMSE t    MAE t     RMSE v    MAE v
0.09      6000      1.14      0.0164
```

In [76]:

```
# Import necessary libraries
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator, FuncFormatter, MaxNLocator

# Define a custom formatting function to round to a specific number of signifi
def format_z(value, _):
    # Specify the number of significant figures you want (e.g., 3)
    num_significant_figures = 3
    return f"{value:.{num_significant_figures}g}"

# Define the range of X and Y values
X = numpy.linspace(20,60,100) # Tin
Y = numpy.linspace(10,30,100) # Tout
X, Y = numpy.meshgrid(X, Y)

# Calculate Z using your equation
hr_min = n2min * (1 + ((1 / hc) * n3min * Bolt * ((X + 273) + (Y + 273) * ((X -
t_dif = X - Y
```

```
Z = hr_min * n1min * t_dif / (n1min + hr_min)

# Create a 3D figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot the 3D surface
surf = ax.plot_surface(X, Y, Z, cmap='coolwarm')
fig.colorbar(surf, shrink=0.35, aspect=10, pad=0.1)

# Set labels for the axes
ax.set_xlabel('Tin [°C]', fontsize=12, labelpad=10)
ax.set_ylabel('Tout [°C]', fontsize=12, labelpad=10)
ax.set_zlabel('Heat Transfer [W/°C]', fontsize=12, labelpad=2)

# Set limits for the axes
ax.set_xlim(70.0, 0.0)
ax.set_ylim(0.0, 40.0)
ax.set_zlim(0, 150)

# ax.zaxis.set_major_locator(LinearLocator(10))
# Reduce the number of ticks on the z-axis
ax.zaxis.set_major_locator(MaxNLocator(nbins=5))

# Customize tick formatting for z-axis using the custom formatting function
ax.zaxis.set_major_formatter(FuncFormatter(format_z))

# Customize tick formatting for z-axis
ax.tick_params(axis='both', which='major', labelsize=10)

# Set the view perspective to orient the plot
ax.view_init(elev=20, azim=-25)  # Adjust the angles as needed

# Show the plot
plt.show()
```



In [ ]: