Ha Yun Anna Yoon
ME 249 Project 1 Report

For Project 1, I worked on all of the tasks by myself.

Task 1

In task 1, I was given CodeP1.1 from the class that contains data array for the project. The log-log plot of heat flux vs wall superheat for the first two groups (g = 0.098 and 9.8 m/s^2) in the Appendix listing is attached in Figure 1.

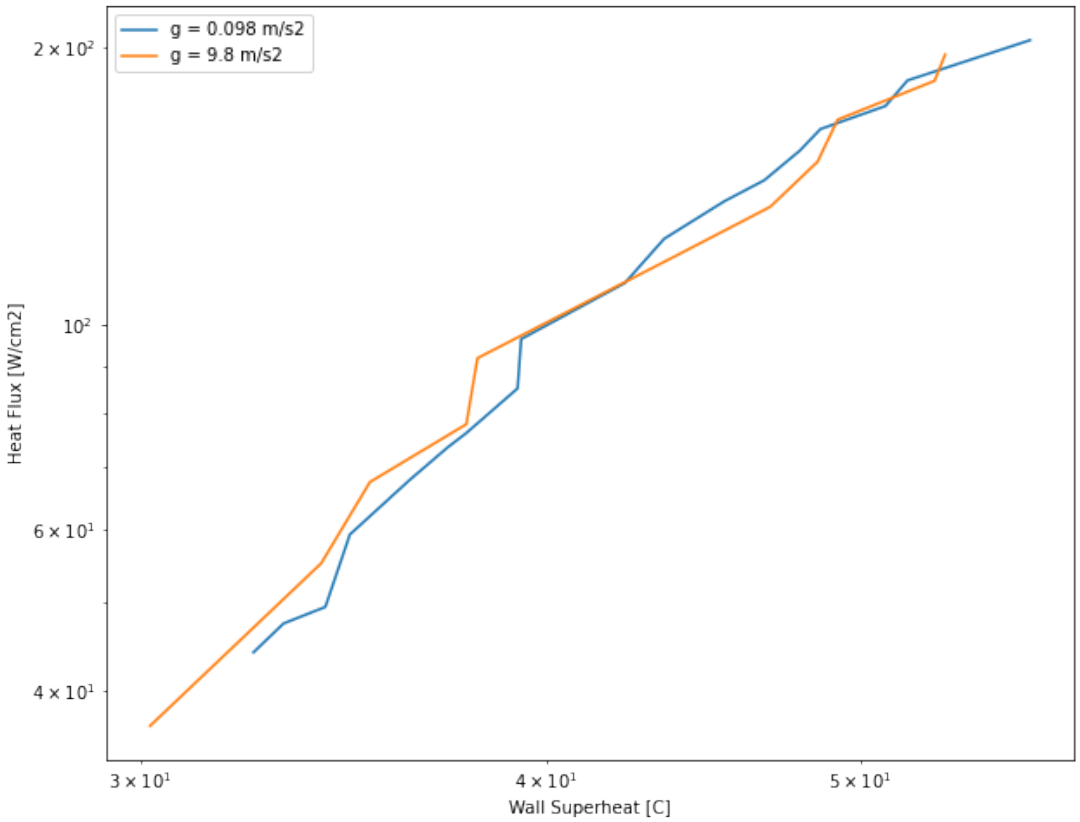


Figure 1. A log-log plot of heat flux vs wall superheat for two gravity levels (g = 0.098 and 9.8 m/s^2).

Task 2

In task 2, I was given CodeP1.2 from the class that contains genetic algorithm. In task 2, genetic algorithm was used to do raw data analysis. The equations 1-4 below were used to create the genetic algorithm. The fully assembled code successfully achieved a minimum error less than 0.04. When different initial guesses were made, the outputs were very sensitive to initial guesses—when I inputted initial guesses farther away from the values, it took double the number of generations to achieve low error (less than 0.03).

$$q'' = n_1(T_w - T_{sat})^{n_2} g^{n_3} \quad (1)$$

$$\ln q'' = \ln n_1 + n_2 \ln(T_w - T_{sat}) + n_3 \ln(g) \quad (2)$$

$$f_{err,i} = -\ln q''_{data,i} + \ln n_1 + n_2 \ln(T_w - T_{sat})_{data,i} + n_3 \ln(g_{data,i}) \quad (3)$$

$$F_{err} = \sum_{i=1}^{N_D} \left| f_{err,i} \right| / \left| \ln q''_{data,i} \right| \quad (4)$$

Task 3

In task 3, the given CodeP1.1 and CodeP1.2 were modified to train a five constant model that includes variation of pressure and the surface tension parameter (γ). From CodeP1.1 the rest of the commented data were uncommented to include the additional data into our dataset for training. Therefore, the database size increased to 77, so the ND and NS in our codes were modified to 77 in order to reflect the size change. Since five constant model is being used, codes were changed to include the n_4 and n_5 values in mating, constant output write ups, and resulting plots. Also, F_{err} , $F_{err\ avgn}$, q''_{pred} lines were modified to take into account n_4 and n_5 value inclusions based on the following equations:

$$q'' = n_1 (T_w - T_{sat})^{n_2} (g + n_4 g_{en} \gamma)^{n_3} P^{n_5} \quad (5)$$

$$\ln q'' = \ln n_1 + n_2 \ln (T_w - T_{sat}) + n_3 \ln (g + n_4 g_{en} \gamma) + n_5 \ln (P) \quad (6)$$

$$f_{err,i} = -\ln q''_{data,i} + \ln n_1 + n_2 \ln (T_w - T_{sat})_{data,i} + n_3 \ln (g_{data,i} + n_4 g_{en} \gamma_{data,i}) + n_5 \ln P_{data,i} \quad (7)$$

$$F_{err} = \sum_{i=1}^{N_D} \left| f_{err,i} \right| / \left| \ln q''_{data,i} \right| \quad (8)$$

Initial guesses were also changed to be n_{0i} = -1.0, n_{1i} = 0.00476, n_{2i} = 3.028, n_{3i} = 0.2249, n_{4i} = 1.054, and n_{5i} = 0.217. The new model was run to determine the constants for best fit with the data. Table 1 below summarizes the constants n_1 through n_5 for best fit to the data. Resulting curve-fit equation was used to create a surface plot (Fig. 2) of $q''/(T_w - T_{sat})^{n_2}$ vs g and γ for $1.0 < g < 20\ m/s^2$ and $0.001 < \gamma < 2$ at P of 10kPa. Also, the CodeP1.2 created two plots—constants and their errors per generation (Fig. 3) and measured vs predicted heat flux values based on genetic algorithm using raw data analysis (Fig. 4).

Table 1. Constants n_1 through n_5 for best fit with data. Constants given for three different conditions—minimum, population average, and time average. Constant values are within the range as follows: n_1 = 0.00047~0.00049, n_2 = 2.93~3.01, n_3 = 0.23~0.33, n_4 = 1.03~1.31, and n_5 = 0.21~0.27.

	n1	n2	n3	n4	n5
Minimum	0.00047260142357352343	2.92585532908119537	0.327086662724091	1.3146212382723095	0.21460273511360908
Population Average	0.00049252442842515 29	3.006557669679141	0.232103204194760 44	1.0368073348301452	0.2537862400534099
Time Average	0.00047741001338118 28	2.968653502936508	0.24873206603322 39	1.2648352921801513	0.2651268426862002

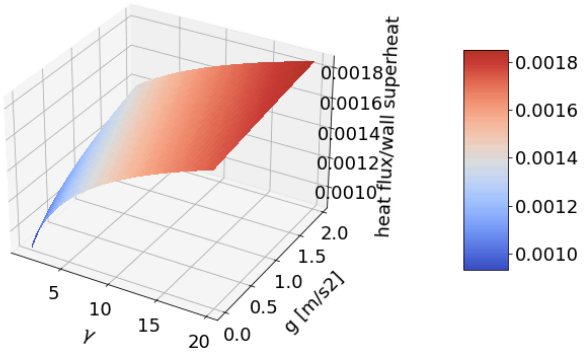


Figure 2. Surface plot of $q"/(T_w - T_{sat})^{n_2}$ vs g and γ for $1.0 < g < 20\ m/s^2$ and $0.001 < \gamma < 2$ at P of 10kPa.

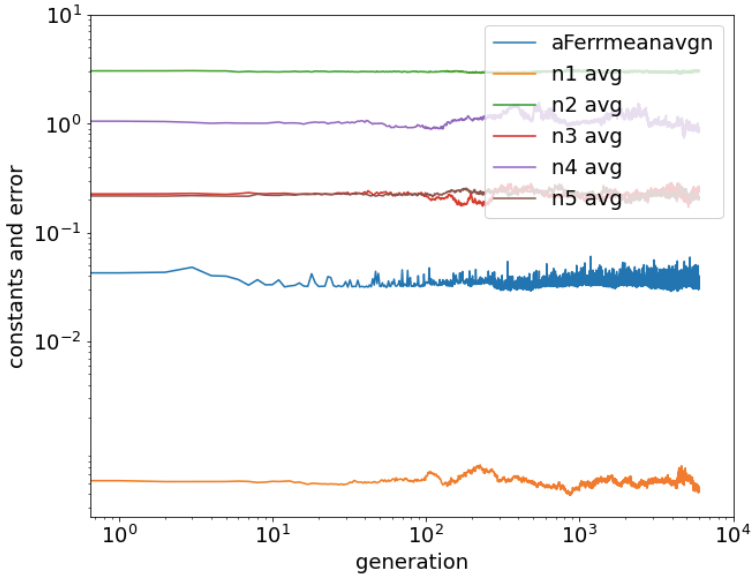


Figure 3. The five constants and their errors as the generation increases. As the generation increases, the constants converge.

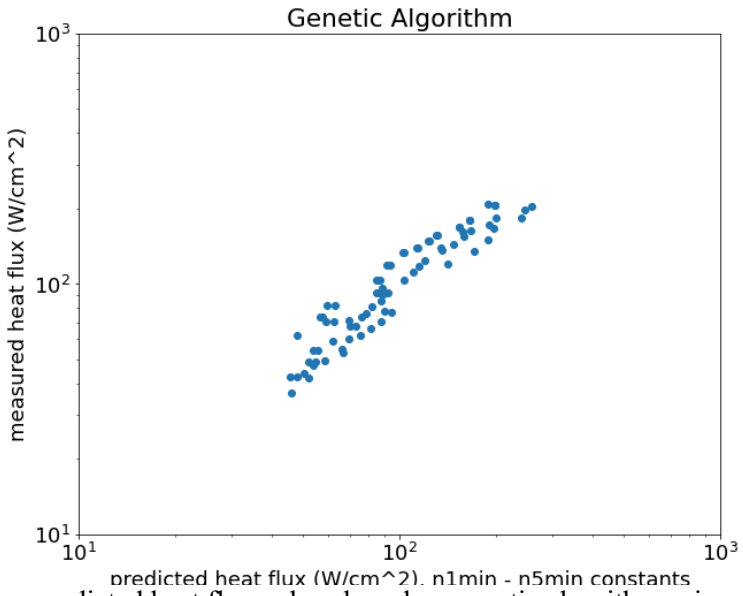


Figure 4. Measured vs predicted heat flux values based on genetic algorithm using raw data analysis. It is clear that there is a linear log-log trend between the measured and the predicted heat flux.

Task 4

In task 4, the given data array in CodeP1.1 were modified to give 5 non-dimensional parameters suggested by the well-known Rohsenow correlation for nucleate boiling. Therefore, the given ydata array from CodeP1.1 were modified in code based on the equations below to give a new non-dimensional parameter array consisting of [Q_s, Ja_s, g/g_{en}, γ, Pr_l]. Also, the table of properties given (Table 2) were used to accurately calculate the non-dimensional parameters (table of values available in the Appendix).

$$Q_s = 10\frac{q''}{\mu_l h_{lv}}\sqrt{\frac{\sigma}{g_{en}(\rho_l - \rho_v)}} \tag{9}$$

$$Ja_s = 100\frac{c_{pl}(T_w - T_{sat})}{h_{lv}} \tag{10}$$

$$Pr_l \tag{11}$$

$$\frac{g}{g_{en}} \tag{12}$$

$$\gamma = \frac{\sigma_w - \sigma_{mix}}{\sigma_{mix}} \tag{13}$$

Table 2. Low pressure water properties given to determine dimensionless parameters.

	P = 5.5 kPa	P = 7.0 kPa	P = 9.5 kPa
T_{sat} (°C)	34.9	38.0	45.0
c_{pl} (kJ/kg°C)	4.18	4.18	4.18
h_{lv} (kJ/kg°C)	2418	2406	2394
μ_l (Ns/m^2)	7.19×10^{-6}	6.53×10^{-6}	5.96×10^{-6}
Pr_l	4.83	4.54	3.91
ρ_l (kg/m^3)	994	993	990
ρ_v (kg/m^3)	0.0397	0.0476	0.182
σ (N/m)	0.0706	0.0692	0.0688

With the modified dimensionless groups, the postulated relation for the interrelationship among these parameters were as indicated in equations 14-17. Thus, the lines that computed the appropriate error and Q_s quantities for the five non-dimensional model were modified accordingly.

$$Q_s = n_1 Ja_s^{n_2} \left(\frac{g}{g_{en}} + n_3\gamma\right)^{n_4} Pr_l^{-n_5} \tag{14}$$

$$\ln Q_s = \ln n_1 + n_2 \ln(Ja_s) + n_4 \ln\left(\frac{g}{g_{en}} + n_3\gamma\right) - n_5 \ln(Pr_l) \tag{15}$$

$$f_{err,i} = -\ln Q_{s_{data,i}} + \ln n_1 + n_2 \ln(Ja_s)_{data,i} + n_4 \ln\left(\frac{g_{data,i}}{g_{en}} + n_3\gamma_{data,i}\right) - n_5 \ln Pr_{l\ data,i} \tag{16}$$

$$F_{err} = \sum_{i=1}^{N_D} \left|f_{err,i}\right| \Big/ \left|\ln Q_{s_{data,i}}\right| \tag{17}$$

Task 5

In task 5, the code from Task 3 was modified to train a five dimensionless model developed in Task 4. F_{err} , $F_{err\ avgn}$, Q_s lines were modified to take into account changes based on equations 14-17. Initial guesses were also changed to be $n_{0i} = 1.0$, $n_{1i} = 1.0$, $n_{2i} = 2.9$, $n_{3i} = 0.35$, $n_{4i} = 0.2$, and $n_{5i} = 2.0$. The new model was run to determine the constants for best fit with the dimensionless data. Table 3 below summarizes the constants n_1 through n_5 for best fit to the data. Resulting curve-fit equation was used to create a surface plot (Fig. 5) of $Q_s Pr_l^{n_5} / Ja_s^{n_2}$ vs g and γ for $0.01 \le g/g_{en} \le 2$ and $0.001 \le \gamma \le 2$. In addition, two plots were created—constants and their errors per generation (Fig. 6) and measured vs predicted heat flux values based on genetic algorithm using raw data analysis (Fig. 7).

Table 3. Constants n_1 through n_5 for best fit with data. Constants given for three different conditions—minimum, population average, and time average. Constant values are within the range as follows: n_1 = 0.998~1.00, n_2 = 2.92~2.99, n_3 = 0.347~0.35, n_4 = 0.200~0.201, and n_5 = 1.99~2.0.

	n1	n2	n3	n4	n5
Minimum	1.002821339355131	2.9897854303734923	0.34928119094999954	0.20166686010748558	1.98536497616805
Population Average	0.9984136508146513	2.9191938094867087	0.34665698315125326	0.2001410709028095	1.9855927292663735
Time Average	1.0001198575643	2.934973182156045	0.35002714281341796	0.20004694099636128	1.9997018657847132

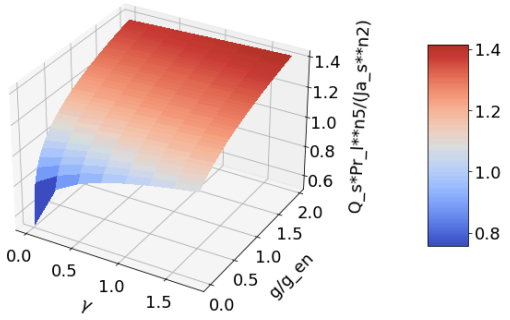


Figure 5. Surface plot of $Q_s Pr_l^{n_5} / Ja_s^{n_2}$ vs g and γ for $0.01 \le g/g_{en} \le 2$ and $0.001 \le \gamma \le 2$.

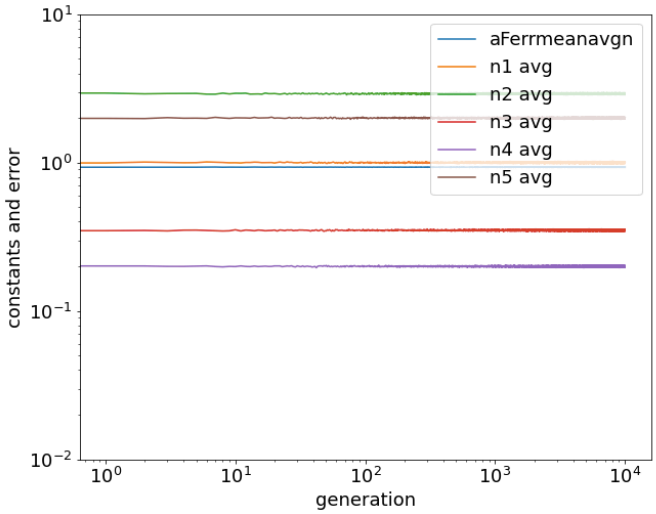


Figure 6. The five constants and their errors as the generation increases. As the generation increases, the constants converge.

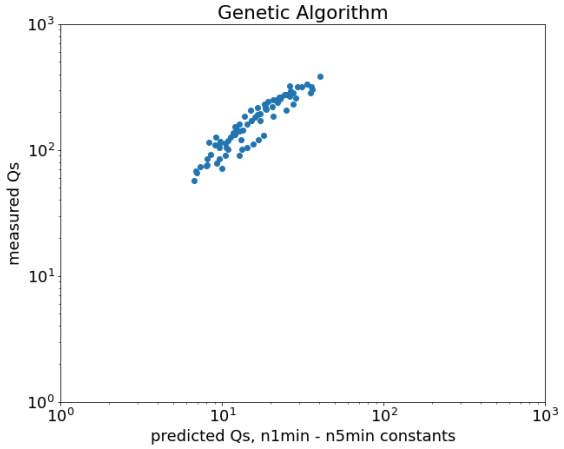


Figure 7. Measured vs predicted Q_s values based on genetic algorithm using dimensionless parameter analysis. It is clear that there is a linear log-log trend between the measured and the predicted Q_s.

Overall, this project has shown that initial guesses of model constants are critical because it not only affects the stability of the algorithm, but also affects the speed of convergence to a satisfactory fit (in our case, error < 0.03). For example, if the initial guesses are farther away from the actual value, it takes even double or triple the generation to converge to acceptable limit. Also, the stability of algorithm hugely varied based on the closeness of the model constants. Different initial guesses of model constants would widely vary the measured vs predicted values of heat flux (or Q_s in the nondimensional set)—for example, initial guesses close to best fit would give data scattered closely together along the linear trend that would give a high R^2 values; however, initial guesses far away from best fit would give data scattered in groups (not clumped together to show as one linear trend) that would look like several linear trend that are parallel.

As for advantages and disadvantages of raw data analysis versus dimensionless data analysis for this heat transport process varies. It takes less time to do raw data analysis since there is no need to convert to dimensionless parameters. It also took fewer generations to find

constant values of best fit with the raw data analysis. However, dimensionless data gave constant values of best fit that were close to each other—i.e., minimum, time average, and population average constant values were much closer to each other. Also, it was easier to use dimensionless data because it allowed for simplicity—reducing dimensions--in this complex heat transfer problem.

Appendix

Task 4 Dimensionless parameters

ndydata: [Q_s, Ja_s, g/g_{en}, γ, Pr_l]

```
[[68.3, 5.62, 0.01, 1.79, 4.83],
[73.4, 5.74, 0.01, 1.79, 4.83],
[76.5, 5.91, 0.01, 1.79, 4.83],
[91.7, 6.02, 0.01, 1.79, 4.83],
[105.0, 6.28, 0.01, 1.79, 4.83],
[114.0, 6.45, 0.01, 1.79, 4.83],
[118.2, 6.53, 0.01, 1.79, 4.83],
[132.1, 6.78, 0.01, 1.79, 4.83],
[149.4, 6.79, 0.01, 1.79, 4.83],
[171.9, 7.31, 0.01, 1.79, 4.83],
[192.0, 7.52, 0.01, 1.79, 4.83],
[210.9, 7.85, 0.01, 1.79, 4.83],
[222.2, 8.07, 0.01, 1.79, 4.83],
[239.4, 8.28, 0.01, 1.79, 4.83],
[252.6, 8.40, 0.01, 1.79, 4.83],
[267.6, 8.80, 0.01, 1.79, 4.83],
[285.2, 8.94, 0.01, 1.79, 4.83],
[315.4, 9.75, 0.01, 1.79, 4.83],
[56.8, 5.22, 1.0, 1.79, 4.83],
[85.3, 5.89, 1.0, 1.79, 4.83],
[104.5, 6.10, 1.0, 1.79, 4.83],
[120.8, 6.53, 1.0, 1.79, 4.83],
[142.5, 6.59, 1.0, 1.79, 4.83],
[185.8, 7.62, 1.0, 1.79, 4.83],
[208.0, 8.11, 1.0, 1.79, 4.83],
[232.7, 8.38, 1.0, 1.79, 4.83],
[258.6, 8.51, 1.0, 1.79, 4.83],
[284.9, 9.11, 1.0, 1.79, 4.83],
[304.3, 9.18, 1.0, 1.79, 4.83],
[79.1, 4.89, 2.0, 1.79, 3.91],
[90.9, 5.12, 2.0, 1.79, 3.91],
[101.7, 5.17, 2.0, 1.79, 3.91],
[115.9, 4.98, 2.0, 1.79, 3.91],
[132.1, 5.33, 2.0, 1.79, 3.91],
[137.6, 5.29, 2.0, 1.79, 3.91],
[152.7, 5.34, 2.0, 1.79, 3.91],
[171.5, 6.02, 2.0, 1.79, 3.91],
[193.9, 6.02, 2.0, 1.79, 3.91],
[222.3, 6.18, 2.0, 1.79, 3.91],
[249.6, 6.43, 2.0, 1.79, 3.91],
[261.1, 6.65, 2.0, 1.79, 3.91],
[276.8, 6.83, 2.0, 1.79, 3.91],
[293.0, 6.98, 2.0, 1.79, 3.91],
[315.6, 7.37, 2.0, 1.79, 3.91],
[334.5, 7.54, 2.0, 1.79, 3.91],
[382.6, 8.03, 2.0, 1.79, 3.91],
[65.7, 5.13, 2.0, 1.79, 4.83],
[75.4, 5.36, 2.0, 1.79, 4.83],
[84.4, 5.39, 2.0, 1.79, 4.83],
[109.6, 5.60, 2.0, 1.79, 4.83],
[114.1, 5.43, 2.0, 1.79, 4.83],
[126.7, 5.62, 2.0, 1.79, 4.83],
[142.3, 6.28, 2.0, 1.79, 4.83],
[160.9, 6.28, 2.0, 1.79, 4.83],
[184.4, 6.43, 2.0, 1.79, 4.83],
[207.0, 6.64, 2.0, 1.79, 4.83],
[216.6, 6.86, 2.0, 1.79, 4.83],
[229.6, 7.07, 2.0, 1.79, 4.83],
[243.1, 7.19, 2.0, 1.79, 4.83],
[261.9, 7.59, 2.0, 1.79, 4.83],
[277.5, 7.78, 2.0, 1.79, 4.83],
[317.4, 8.28, 2.0, 1.79, 4.83],
[130.7, 7.21, 1.0, 0.0, 4.54],
[120.5, 7.04, 1.0, 0.0, 4.54],
[112.0, 6.86, 1.0, 0.0, 4.54],
[105.2, 6.69, 1.0, 0.0, 4.54],
[71.3, 5.91, 1.0, 0.0, 4.54],
[101.8, 6.51, 1.0, 0.0, 4.54],
[90.0, 6.43, 1.0, 0.0, 4.54],
[111.0, 6.29, 0.01, 1.71, 4.83],
[126.2, 6.66, 0.01, 1.71, 4.83],
[140.5, 6.83, 0.01, 1.71, 4.83],
[160.0, 7.19, 0.01, 1.71, 4.83],
[181.2, 7.45, 0.01, 1.71, 4.83],
[214.6, 7.85, 0.01, 1.71, 4.83],
[250.4, 8.28, 0.01, 1.71, 4.83],
[321.3, 8.80, 0.01, 1.71, 4.83]]
```

Codes

In [1]:
```
'''>>>>> start CodeP1.1
    V.P. Carey ME249, Spring 2021'''
#import math and numpy packages
import math
import numpy as np

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

#Parameters for Evolution Loop
#create arrays - SWITCH n3 and n4
ydata = []
lydata = []

#set data parameters
ND = 77      #number of data vectors in array, original 45
DI = 5        #number of data items in vector
NS = 77       #total number of DNA strands, original 45

# 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, surf

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])
```

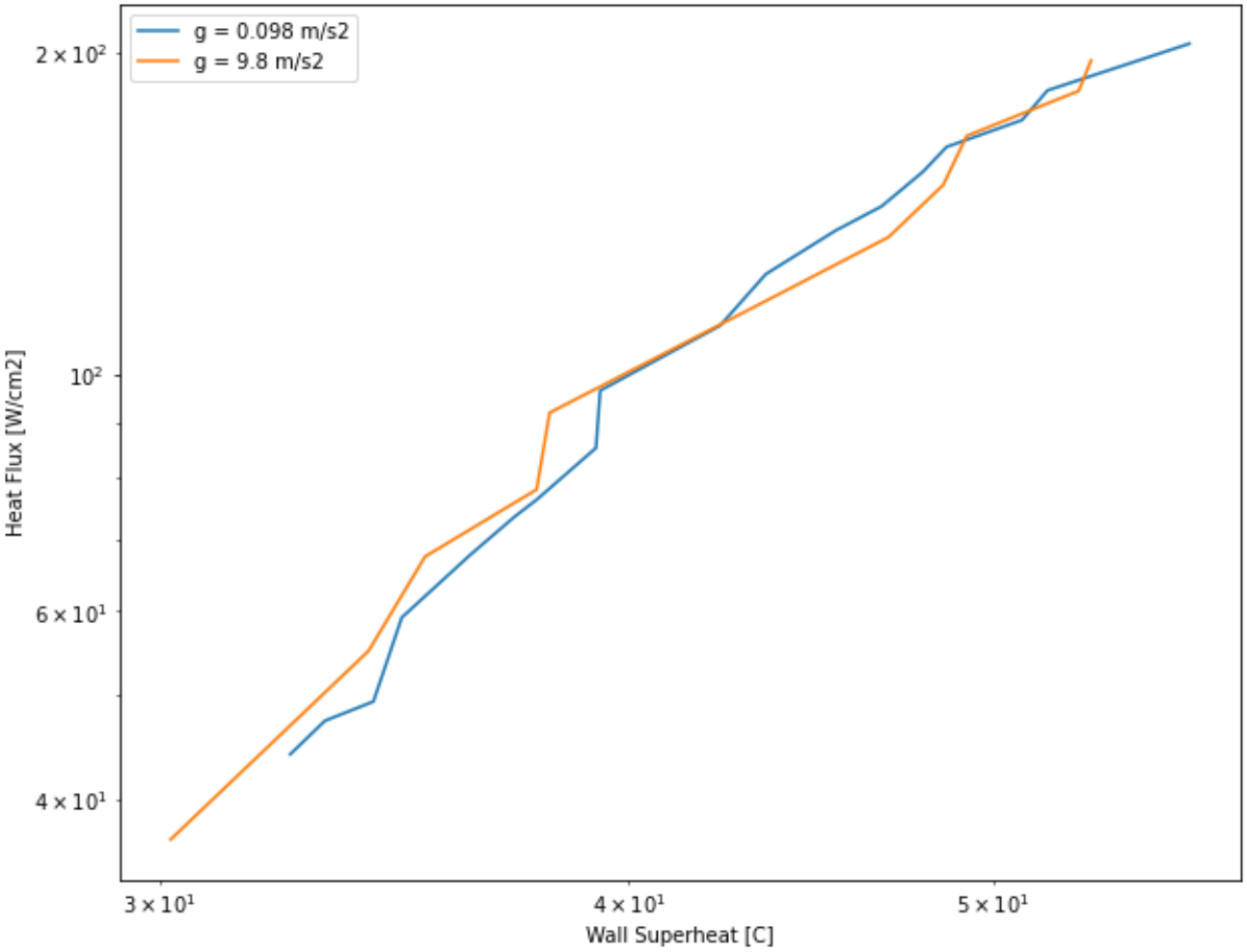
```
ydata.append([179.2, 45.0, 19.6, 1.79, 5.5])
ydata.append([205.0, 47.9, 19.6, 1.79, 5.5])

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


# 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 CodeP1.1 '''
```

```
ydata = [[44.1, 32.5, 0.098, 1.79, 5.5], [47.4, 33.2, 0.098, 1.79, 5.5], [49.4
, 34.2, 0.098, 1.79, 5.5], [59.2, 34.8, 0.098, 1.79, 5.5], [67.8, 36.3, 0.098,
1.79, 5.5], [73.6, 37.3, 0.098, 1.79, 5.5], [76.3, 37.8, 0.098, 1.79, 5.5], [8
5.3, 39.2, 0.098, 1.79, 5.5], [96.5, 39.3, 0.098, 1.79, 5.5], [111.0, 42.3, 0.
098, 1.79, 5.5], [124.0, 43.5, 0.098, 1.79, 5.5], [136.2, 45.4, 0.098, 1.79, 5
.5], [143.5, 46.7, 0.098, 1.79, 5.5], [154.6, 47.9, 0.098, 1.79, 5.5], [163.1,
48.6, 0.098, 1.79, 5.5], [172.8, 50.9, 0.098, 1.79, 5.5], [184.2, 51.7, 0.098,
1.79, 5.5], [203.7, 56.4, 0.098, 1.79, 5.5], [36.7, 30.2, 9.8, 1.79, 5.5], [55
.1, 34.1, 9.8, 1.79, 5.5], [67.5, 35.3, 9.8, 1.79, 5.5], [78.0, 37.8, 9.8, 1.7
9, 5.5], [92.0, 38.1, 9.8, 1.79, 5.5], [120.0, 44.1, 9.8, 1.79, 5.5], [134.3,
46.9, 9.8, 1.79, 5.5], [150.3, 48.5, 9.8, 1.79, 5.5], [167.0, 49.2, 9.8, 1.79,
5.5], [184.0, 52.7, 9.8, 1.79, 5.5], [196.5, 53.1, 9.8, 1.79, 5.5], [42.4, 28.
0, 19.6, 1.79, 9.5], [48.7, 29.3, 19.6, 1.79, 9.5], [54.5, 29.6, 19.6, 1.79, 9
.5], [62.1, 28.5, 19.6, 1.79, 9.5], [70.8, 30.5, 19.6, 1.79, 9.5], [73.7, 30.3
, 19.6, 1.79, 9.5], [81.8, 30.6, 19.6, 1.79, 9.5], [91.9, 34.5, 19.6, 1.79, 9.
5], [103.9, 34.5, 19.6, 1.79, 9.5], [119.1, 35.4, 19.6, 1.79, 9.5], [133.7, 36
.8, 19.6, 1.79, 9.5], [139.9, 38.1, 19.6, 1.79, 9.5], [148.3, 39.1, 19.6, 1.79
, 9.5], [157.0, 40.0, 19.6, 1.79, 9.5], [169.1, 42.2, 19.6, 1.79, 9.5], [179.2
, 43.2, 19.6, 1.79, 9.5], [205.0, 46.0, 19.6, 1.79, 9.5], [42.4, 29.7, 19.6, 1
.79, 5.5], [48.7, 31.0, 19.6, 1.79, 5.5], [54.5, 31.2, 19.6, 1.79, 5.5], [70.8
, 32.4, 19.6, 1.79, 5.5], [73.7, 31.4, 19.6, 1.79, 5.5], [81.8, 32.5, 19.6, 1.
79, 5.5], [91.9, 36.3, 19.6, 1.79, 5.5], [103.9, 36.3, 19.6, 1.79, 5.5], [119.
1, 37.2, 19.6, 1.79, 5.5], [133.7, 38.4, 19.6, 1.79, 5.5], [139.9, 39.7, 19.6,
1.79, 5.5], [148.3, 40.9, 19.6, 1.79, 5.5], [157.0, 41.6, 19.6, 1.79, 5.5], [1
69.1, 43.9, 19.6, 1.79, 5.5], [179.2, 45.0, 19.6, 1.79, 5.5], [205.0, 47.9, 19
.6, 1.79, 5.5], [77.0, 41.5, 9.8, 0.0, 7.0], [71.0, 40.5, 9.8, 0.0, 7.0], [66.
0, 39.5, 9.8, 0.0, 7.0], [62.0, 38.5, 9.8, 0.0, 7.0], [42.0, 34.0, 9.8, 0.0, 7
.0], [60.0, 37.5, 9.8, 0.0, 7.0], [53.0, 37.0, 9.8, 0.0, 7.0], [71.7, 36.4, 0.
098, 1.71, 5.5], [81.5, 38.5, 0.098, 1.71, 5.5], [90.7, 39.5, 0.098, 1.71, 5.5
], [103.3, 41.6, 0.098, 1.71, 5.5], [117.0, 43.1, 0.098, 1.71, 5.5], [138.6, 4
5.4, 0.098, 1.71, 5.5], [161.7, 47.9, 0.098, 1.71, 5.5], [207.5, 50.9, 0.098,
1.71, 5.5]]
```

Out[1]: '>>>>> end CodeP1.1 '

In [2]:

```
'''Anna Yoon Task 1
    ME 249, Jan 2021'''
#group data
x1=[]
x2=[]
y1=[]
y2=[]
for x in range(len(ydata)):
    if ydata[x][2]==0.098:
        x1.append(ydata[x][1])
        y1.append(ydata[x][0])
    elif ydata[x][2]==9.8:
        x2.append(ydata[x][1])
        y2.append(ydata[x][0])

#plot log-log heat flux vs wall superheat
import matplotlib.pyplot as plt
plt.loglog(x1, y1)
plt.loglog(x2,y2)
plt.xlabel('Wall Superheat [C]')
plt.ylabel('Heat Flux [W/cm2]')
plt.legend(["g = 0.098 m/s2","g = 9.8 m/s2"])
plt.show()
#plt.savefig("Part1b.png")
```



In [2]:

```
'''>>>>> start CodeP1.2
    V.P. Carey ME249, Spring 2021'''

import numpy

'''INITIALIZING PARAMETERS'''
n = []
ntemp = []
gen=[0]
g_en = 9.8 #m/s2 earth gravitational
n1avg = [0.0]
n2avg = [0.0]
n3avg = [0.0]
n4avg = [0.0]
n5avg = [0.0]
meanAFerr=[0.0]
aFerrmeanavgn=[0.0]

#set program parameters
NGEN = 6000   #number of generations (steps)
'''Test Case
NGEN = 10000'''
```

```
MFRAC = 0.5   # faction of median threshold

# here the number of data vectors equals the number of DNA strands (or organi
# they can be different if they are randomly paired to compute Ferr (survivab
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 =  -1.0
n1i = 0.000476
n2i = 3.028
n3i = 0.2249
n4i = 1.054
n5i = 0.217

'''Original Case v2
n0i =  -1.0
n1i = 0.00020
n2i = 3.4
n3i = 0.05
n4i = 1.325
n5i = 0.165
'''

''' Test Case
n0i =  -1.0
n1i = 0.00030
n2i = 4.4
n3i = 0.07
n4i = 1.325
n5i = 0.165 '''


#- 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 p
n =  [[-1., n1i+0.001*random(), n2i+0.1*random(), n3i+0.0001*random(), n4i+0.
for i in range(ND):
    n.append([-1., n1i+0.0001*random(), n2i+0.001*random(), n3i+0.0001*random
#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])
#print (Ferr)

aFerrmeanavgnMin=1000000000.0
# these store the  n values for minimum population average error durng NGEN g
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 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 GENERATIONS COMPUTED
for k in range(NGEN):

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


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

    # Here we calculate error Ferr and absolute error aFerr for each data poi
    # for specified n(i), and calculate (mean aFerr) = aFerrmean
    # and (median aFerr) = aFerrmedian for the data collection and specified
```

```
# Note that the number data points ND equals the number of solutions (org
#=============================================================================
for i in range(ND):
    ''' Original
    Ferr[i] = n[i][0]*lydata[i][0] + math.log(n[i][1]) + n[i][2]*lydata[i
    Ferr[i] = Ferr[i] + n[i][3]*math.log( ydata[i][2] ) '''

    Ferr[i] = n[i][0]*lydata[i][0] + math.log(n[i][1]) + n[i][2]*lydata[i
    Ferr[i] = Ferr[i] + n[i][3]*math.log(ydata[i][2] + g_en*n[i][4]*ydata

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

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

# now check each organism/solution to see if aFerr is less than cut of li
#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 populatio
    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 to
#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 o
#=============================================================
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 fixe
    #if (numpy.random.rand() < 0.5)
```

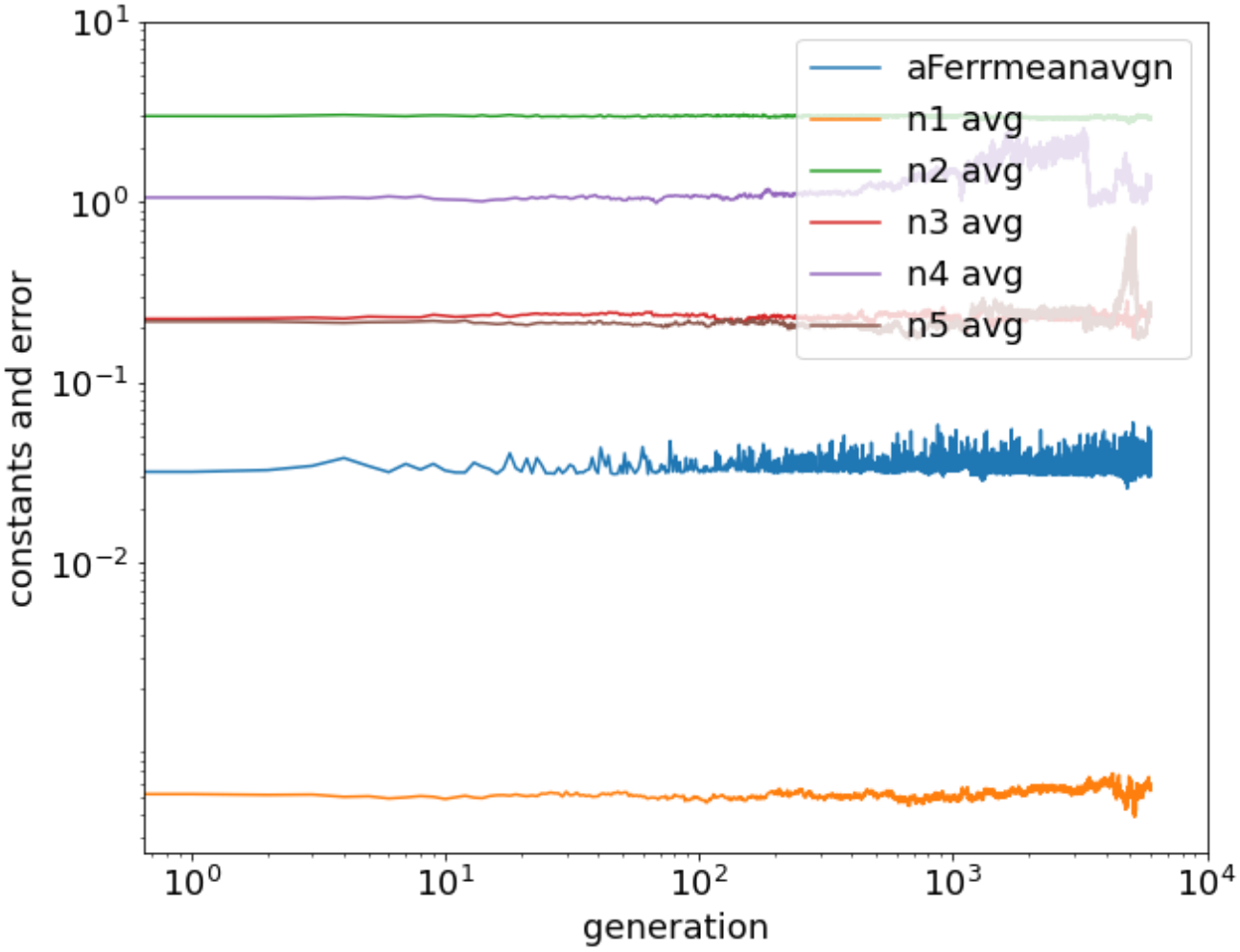
```
        #    ntemp[nkeep+j+1][0] = n[nmate1][0]
        #else
        #    ntemp[nkeep+j+1][0] = n[nmate2][0]

        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][1] = n[nmate1][1]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][1] = n[nmate2][1]*(1.+0.09*2.*(0.5-numpy.random.

        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][2] = n[nmate1][2]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][2] = n[nmate2][2]*(1.+0.09*2.*(0.5-numpy.random.

        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][3] = n[nmate1][3]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][3] = n[nmate2][3]*(1.+0.09*2.*(0.5-numpy.random.

        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][4] = n[nmate1][4]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][4] = n[nmate2][4]*(1.+0.09*2.*(0.5-numpy.random.

        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][5] = n[nmate1][5]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][5] = n[nmate2][5]*(1.+0.09*2.*(0.5-numpy.random.
    #=============================================
    n = deepcopy(ntemp)   # save ntemp as n for use in next generation (next


    '''AVERAGING OVER POPULATION AND OVER TIME, FINDING MINIMUM ERROR SET OF
    # [6] calculate n1avg[k], etc., which are average n values for population
    # at this generation k
    #=============================================
    #initialoze average n's to zero and sum contribution of each member of th
    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 u
    # for this solutions generation k
    # aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of o
    #
```

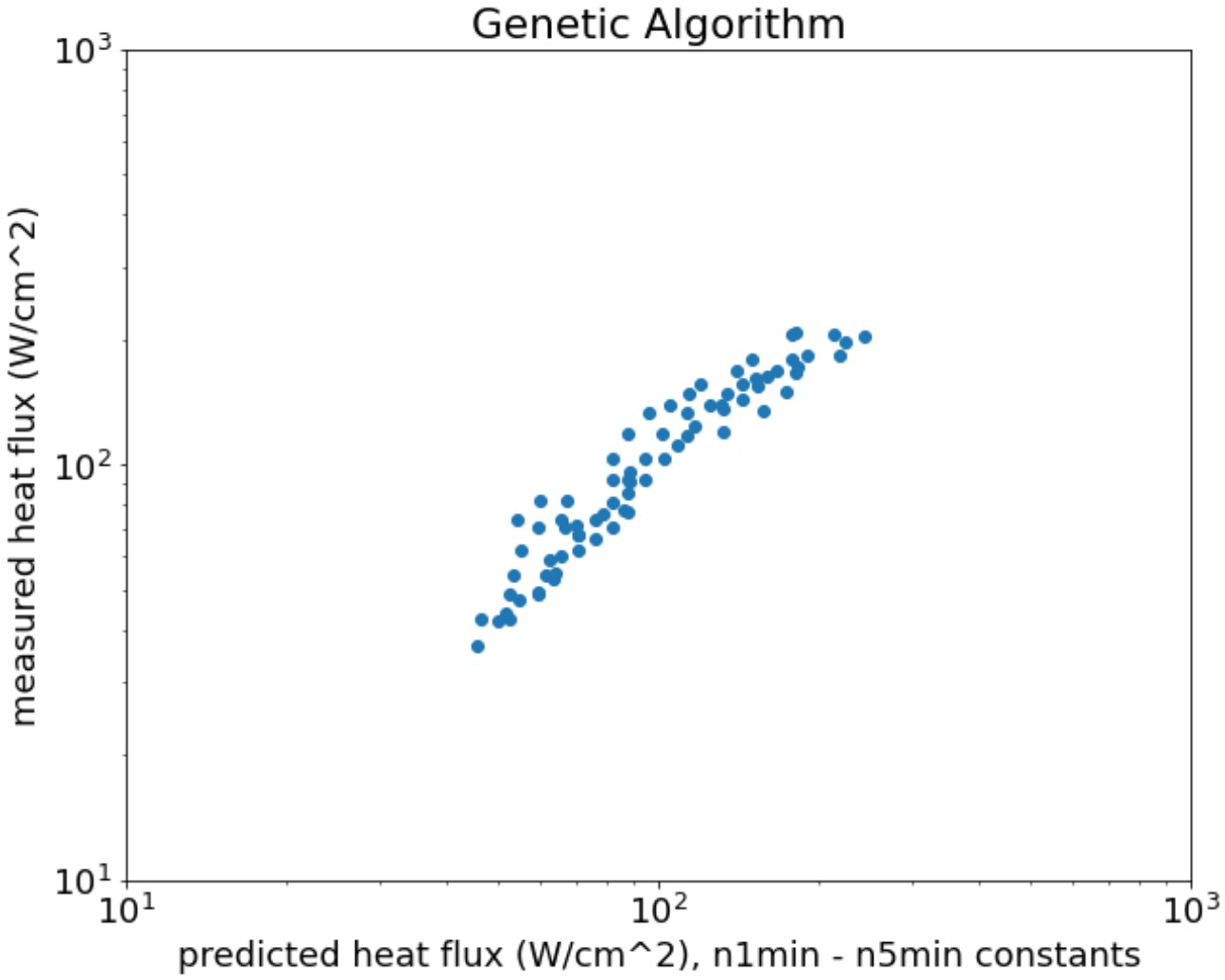
```
    #============================================
    for i in range(ND):
        '''Original
        Ferravgn[i] = -1.*lydata[i][0] + math.log(n1avg[k]) + n2avg[k]*lydata
        Ferravgn[i] = Ferravgn[i] + n3avg[k]*math.log( ydata[i][2] ) '''

        Ferravgn[i] = -1.*lydata[i][0] + math.log(n1avg[k]) + n2avg[k]*lydata
        Ferravgn[i] = Ferravgn[i] + n3avg[k]*math.log( ydata[i][2] + n4avg[k]

        #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 (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 is
    #============================================
    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], aFerrmeanavg
    #print ('kvalue =', k)
    '''end of evolution loop'''
    # ---------------------------------------------------------------
    # ---------------------------------------------------------------


# ---------------------------------------------------------------
#final print and plot of results
# ---------------------------------------------------------------
print('ENDING: pop. avg n1-n5,aFerrmean:', n1avg[k], n2avg[k], n3avg[k], n4av
print('MINUMUM:  avg n1-n5,aFerrmeanMin:', n1min, n2min, n3min, n4min, n5min,
print('TIME AVG:    avg n1-n5,aFerrmean:', n1ta, n2ta, n3ta, n4ta, n5ta, aFer

#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):
    '''Original
    qpppred[i] = n1min*(ydata[i][1]**n2min) * ((ydata[i][2])**n3min) '''
    qpppred[i] = n1min*(ydata[i][1]**n2min) * ((ydata[i][2]+n4min*g_en*ydata[
    qppdata[i] = ydata[i][0]

#========

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

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

# aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of organ
# 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 avg'

# 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()

# data vs. predicted heat flux plot
plt.scatter(qpppred, qppdata)
plt.title('Genetic Algorithm')
plt.xlabel('predicted heat flux (W/cm^2), n1min - n5min constants')
plt.ylabel('measured heat flux (W/cm^2)')
plt.loglog()
plt.xlim(xmax = 1000, xmin = 10)
plt.ylim(ymax = 1000, ymin = 10)

plt.show()
```

```
'''>>>>> end CodeP1.2 '''
```

```
ENDING: pop. avg n1-n5,aFerrmean: 0.0005912342656710523 2.902526860643639 0.24
99213859510798 1.3210530580237352 0.251526803918152 0.031447922257221865
MINUMUM:  avg n1-n5,aFerrmeanMin: 0.00044597762948339403 2.822960294110226 0.2
784948722421603 1.5656334899147812 0.5308561550364231 0.025951911534647862
TIME AVG:    avg n1-n5,aFerrmean: 0.0005510264944817697 2.939144852927464 0.23
026152042264725 1.5227328156847322 0.2609591412531581 0.033707442556654345
```





Out[2]: '>>>>> end CodeP1.2 '

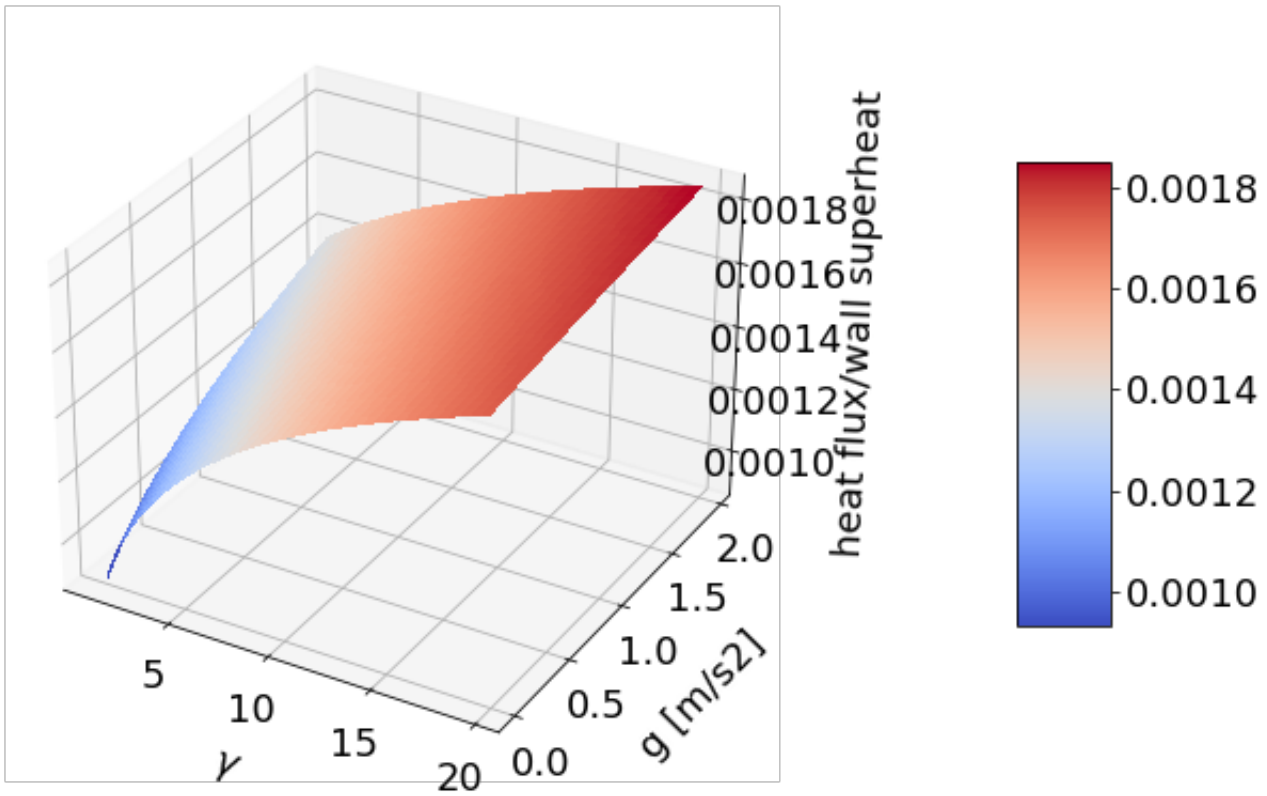
In [20]:
```
#Task 3 Part 1
import matplotlib.pyplot as plt

data = [[n1min, n2min, n3min, n4min, n5min],
        [n1avg[k], n2avg[k], n3avg[k], n4avg[k], n5avg[k]],
        [n1ta, n2ta, n3ta, n4ta, n5ta]]
rows = ('Minimum', 'Population Average', 'Time Average')
columns = ('n1', 'n2', 'n3', 'n4', 'n5')

fig, ax = plt.subplots(1,1)
ax.axis('tight')
ax.axis('off')
table = ax.table(cellText=data,
                 rowLabels=rows,
                 colLabels=columns,
                 rowColours =["palegreen"] * 10,
                 colColours =["palegreen"] * 10,
                 loc="center")

plt.show()
```

	n1	n2	n3	n4	n5
Minimum	0.0004726014235735234 3	2.9258532908119537	0.32708666272409 1	1.3146212382725095	0.21460273511360 98
Population Average	0.0004925244284251529	3.006557669679141	0.23210320419476044	1.0368073348301452	0.25378624005340 99
Time Average	0.0004774100133811828	2.968653502936308	0.2487320660352239	1.2648352921801513	0.26512684268620 02

In [18]:
```
#Task 3 Part 2: plot log-log heat flux/wall superheat vs g vs surface tension

import numpy, scipy, scipy.optimize
import matplotlib
```

```
from mpl_toolkits.mplot3d import  Axes3D
from matplotlib import cm # to colormap 3D surfaces from blue to red
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import matplotlib.pyplot as plt

graphWidth = 800 # units are pixels
graphHeight = 600 # units are pixels

def SurfacePlot(func, data, fittedParameters):
    f = plt.figure()
    axes = f.gca(projection='3d')

    # Make data mesh with appropriate ranges.
    x_data = data[0]
    y_data = data[1]
    z_data = data[2]

    X = np.arange(1, 20, 0.25)
    Y = np.arange(0.001, 2, 0.05)
    X, Y = np.meshgrid(X, Y)
    Z = func(numpy.array([X, Y]), *fittedParameters)

    surf = axes.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                         linewidth=0, antialiased=False)
    f.colorbar(surf, shrink=0.5, aspect=5, pad=0.2)
    #axes.scatter(x_data, y_data, z_data) # show data along with plotted surf
    axes.set_zlabel('heat flux/wall superheat', rotation=60)
    axes.set_ylabel('g [m/s2]')
    axes.set_xlabel(r'$\gamma$', rotation=150)
    axes.xaxis.labelpad=15
    axes.yaxis.labelpad=15
    axes.zaxis.labelpad=15

    plt.show()

def func(data, a, b, c):
    x = data[0]
    y = data[1]
    return a * (y + b*x)**c

xData = []
yData = []
zData = []

for x in range(len(lydata)):
    if 1 <= ydata[x][2] <= 20:
        if 0.001 <= ydata[x][3] <= 2:
            xData.append(ydata[x][3])
            yData.append(ydata[x][2])
            zData.append(qpppred[x] / (ydata[x][1]**n2avg[k]))

data = [xData, yData, zData]
```

```
const = n1avg[k] * (10**n5avg[k]) #P=10kPa
initialParameters = [const, n4avg[k]*g_en, n3avg[k]]
fittedParameters, covariance = scipy.optimize.curve_fit(func, np.asarray([xDa
SurfacePlot(func, data, fittedParameters)
```



In [3]:
```
# Task 4
from copy import copy, deepcopy
from __future__ import print_function

ndydata = deepcopy(ydata)

P = [5.5, 7.0, 9.5] #kPa
Tsat = [34.9, 38.0, 45.0] #Celsius
c_pl = [4.18, 4.18, 4.18] #kJ/kg*C
h_lv = [2418, 2406, 2394] #kJ/kg*C
mu_l = [7.19e-6, 6.53e-6, 5.96e-6] #Ns/m2
Pr_l = [4.83, 4.54, 3.91]
rho_l = [994, 993, 990] #kg/m3
rho_v = [0.0397, 0.0476, 0.182] #kg/m3
sigma = [0.0706, 0.0692, 0.0688] #N/m

for i in range(ND):
    if ydata[i][4] == 5.5:
        k=0
    elif ydata[i][4] == 7.0:
        k=1
    else: #ydata[i][4] == 9.5
        k=2
    ndydata[i][0] = 10*ydata[i][0]/(mu_l[k]*h_lv[k])*math.sqrt(sigma[k]/(g_en
    ndydata[i][1] = 100*c_pl[k]*ydata[i][1]/h_lv[k]
    ndydata[i][2] = ydata[i][2]/g_en
    ndydata[i][3] = ydata[i][3]
    ndydata[i][4] = Pr_l[k]

print ('dimensionless data =', ndydata)
```

```
dimensionless data = [[68.29017033869965, 5.618279569892473, 0.01, 1.79, 4.83]
, [73.40031913955472, 5.739288668320927, 0.01, 1.79, 4.83], [76.49737901886084
, 5.912158808933003, 0.01, 1.79, 4.83], [91.67297242746076, 6.015880893300248,
0.01, 1.79, 4.83], [104.99032990847702, 6.2751861042183625, 0.01, 1.79, 4.83],
[113.97180355846473, 6.448056244830438, 0.01, 1.79, 4.83], [118.15283439552798
, 6.534491315136476, 0.01, 1.79, 4.83], [132.08960385240545, 6.776509511993384
, 0.01, 1.79, 4.83], [149.43313917651966, 6.79379652605459, 0.01, 1.79, 4.83],
[171.88682330148893, 7.312406947890818, 0.01, 1.79, 4.83], [192.01771251697863
, 7.51985111662531, 0.01, 1.79, 4.83], [210.90977778074588, 7.848304383788255,
0.01, 1.79, 4.83], [222.21404634021314, 8.073035566583954, 0.01, 1.79, 4.83],
[239.40272867036205, 8.280479735318446, 0.01, 1.79, 4.83], [252.565233157413,
8.401488833746898, 0.01, 1.79, 4.83], [267.58597357204764, 8.799090157154673,
0.01, 1.79, 4.83], [285.23921488409246, 8.937386269644335, 0.01, 1.79, 4.83],
[315.435548707327, 9.749875930521092, 0.01, 1.79, 4.83], [56.83104878526706, 5
.220678246484698, 1.0, 1.79, 4.83], [85.32399967488324, 5.894871794871795, 1.0
, 1.79, 4.83], [104.5257709265811, 6.102315963606286, 1.0, 1.79, 4.83], [120.7
8533529293816, 6.534491315136476, 1.0, 1.79, 4.83], [142.4647544480809, 6.5863
523573200995, 1.0, 1.79, 4.83], [185.8235927583664, 7.623573200992555, 1.0, 1.
79, 4.83], [207.96757089540506, 8.10760959470637, 1.0, 1.79, 4.83], [232.74404
992985393, 8.384201819685691, 1.0, 1.79, 4.83], [258.6044999220599, 8.50521091
8114145, 1.0, 1.79, 4.83], [284.9295088961618, 9.110256410256412, 1.0, 1.79, 4
.83], [304.28613314182496, 9.179404466501241, 1.0, 1.79, 4.83], [79.1405307353
4394, 4.888888888888889, 2.0, 1.79, 3.91], [90.89961902856722, 5.1158730158730
```

```
155, 2.0, 1.79, 3.91], [101.72544634613786, 5.1682539682539685, 2.0, 1.79, 3.9
1], [115.911013176058, 4.976190476190476, 2.0, 1.79, 3.91], [132.1497541524139
2, 5.325396825396825, 2.0, 1.79, 3.91], [137.56266781119925, 5.29047619047619,
2.0, 1.79, 3.91], [152.68149561677205, 5.342857142857143, 2.0, 1.79, 3.91], [1
71.53336732495538, 6.023809523809524, 2.0, 1.79, 3.91], [193.93163074061874, 6
.023809523809524, 2.0, 1.79, 3.91], [222.30276440045904, 6.18095238095238, 2.0
, 1.79, 3.91], [249.55398488951616, 6.425396825396826, 2.0, 1.79, 3.91], [261.
1264209876089, 6.652380952380953, 2.0, 1.79, 3.91], [276.8052053785733, 6.8269
84126984128, 2.0, 1.79, 3.91], [293.04394635492923, 6.984126984126984, 2.0, 1.
79, 3.91], [315.6288619657231, 7.368253968253969, 2.0, 1.79, 3.91], [334.48073
36739065, 7.542857142857144, 2.0, 1.79, 3.91], [382.63700001758275, 8.03174603
1746032, 2.0, 1.79, 3.91], [65.65766944128947, 5.13424317617866, 2.0, 1.79, 4.
83], [75.41340806110371, 5.358974358974359, 2.0, 1.79, 4.83], [84.394881711091
41, 5.393548387096774, 2.0, 1.79, 4.83], [109.63591972743619, 5.60099255583126
5, 2.0, 1.79, 4.83], [114.12665655243005, 5.428122415219189, 2.0, 1.79, 4.83],
[126.66974906361976, 5.618279569892473, 2.0, 1.79, 4.83], [142.3099014541156,
6.2751861042183625, 2.0, 1.79, 4.83], [160.89226072995226, 6.2751861042183625,
2.0, 1.79, 4.83], [184.42991581267864, 6.430769230769231, 2.0, 1.79, 4.83], [2
07.03845293161322, 6.638213399503722, 2.0, 1.79, 4.83], [216.63933855746217, 6
.862944582299422, 2.0, 1.79, 4.83], [229.6469900505478, 7.070388751033913, 2.0
, 1.79, 4.83], [243.11920052552938, 7.191397849462366, 2.0, 1.79, 4.83], [261.
8564127953313, 7.588999172870141, 2.0, 1.79, 4.83], [277.4965651858272, 7.7791
56327543424, 2.0, 1.79, 4.83], [317.44863762887593, 8.280479735318446, 2.0, 1.
79, 4.83], [130.6946215008316, 7.209891936824605, 1.0, 0.0, 4.54], [120.510625
02024733, 7.0361596009975065, 1.0, 0.0, 4.54], [112.0239612864271, 6.862427265
170408, 1.0, 0.0, 4.54], [105.23463029937092, 6.688694929343308, 1.0, 0.0, 4.5
4], [71.28797536408997, 5.906899418121363, 1.0, 0.0, 4.54], [101.8399648058428
1, 6.514962593516209, 1.0, 0.0, 4.54], [89.95863557849448, 6.42809642560266, 1
.0, 0.0, 4.54], [111.02959667312393, 6.2924731182795695, 0.01, 1.71, 4.83], [1
26.20519008172386, 6.65550041356493, 0.01, 1.71, 4.83], [140.45166552653194, 6
.828370554177006, 0.01, 1.71, 4.83], [159.96314276616042, 7.191397849462366, 0
.01, 1.71, 4.83], [181.17800293940724, 7.450703060380479, 0.01, 1.71, 4.83], [
214.6262496359132, 7.848304383788255, 0.01, 1.71, 4.83], [250.39729124189876,
8.280479735318446, 0.01, 1.71, 4.83], [321.31996247800856, 8.799090157154673,
0.01, 1.71, 4.83]]
```

In [10]:

```
'''Initialize Parameters'''
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]

#set program parameters
NGEN = 10000   #number of generations (steps)
MFRAC = 0.5   # faction of median threshold

# here the number of data vectors equals the number of DNA strands (or organi
# they can be different if they are randomly paired to compute Ferr (survivab
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 = 1
n1i = 3
n2i = 1
n3i = 1
n4i = 0.3
n5i = 0.3'''
'''n0i = 1
n1i = 3
n2i = 1
n3i = 10
n4i = 0.3
n5i = 0.3'''
n0i = 1
n1i = 1
n2i = 2.9
n3i = 0.35
n4i = 0.2
n5i = 2

#- 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 p
n =  [[-1., n1i+0.001*random(), n2i+0.1*random(), n3i+0.0001*random(), n4i+0.
for i in range(ND):
    n.append([-1., n1i+0.0001*random(), n2i+0.001*random(), n3i+0.0001*random
#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])
#print (Ferr)

aFerrmeanavgnMin=1000000000.0
# these store the  n values for minimum population average error durng NGEN g
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 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 GENERATIONS COMPUTED
for k in range(NGEN):


    '''CALCULATING ERROR (FITNESS).'''

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

        Ferr[i] = n[i][0]*ndydata[i][0] + math.log(n[i][1]) + n[i][2]*ndydata
        Ferr[i] = Ferr[i] + n[i][4]*math.log(ndydata[i][2] + n[i][3]*ndydata[

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

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

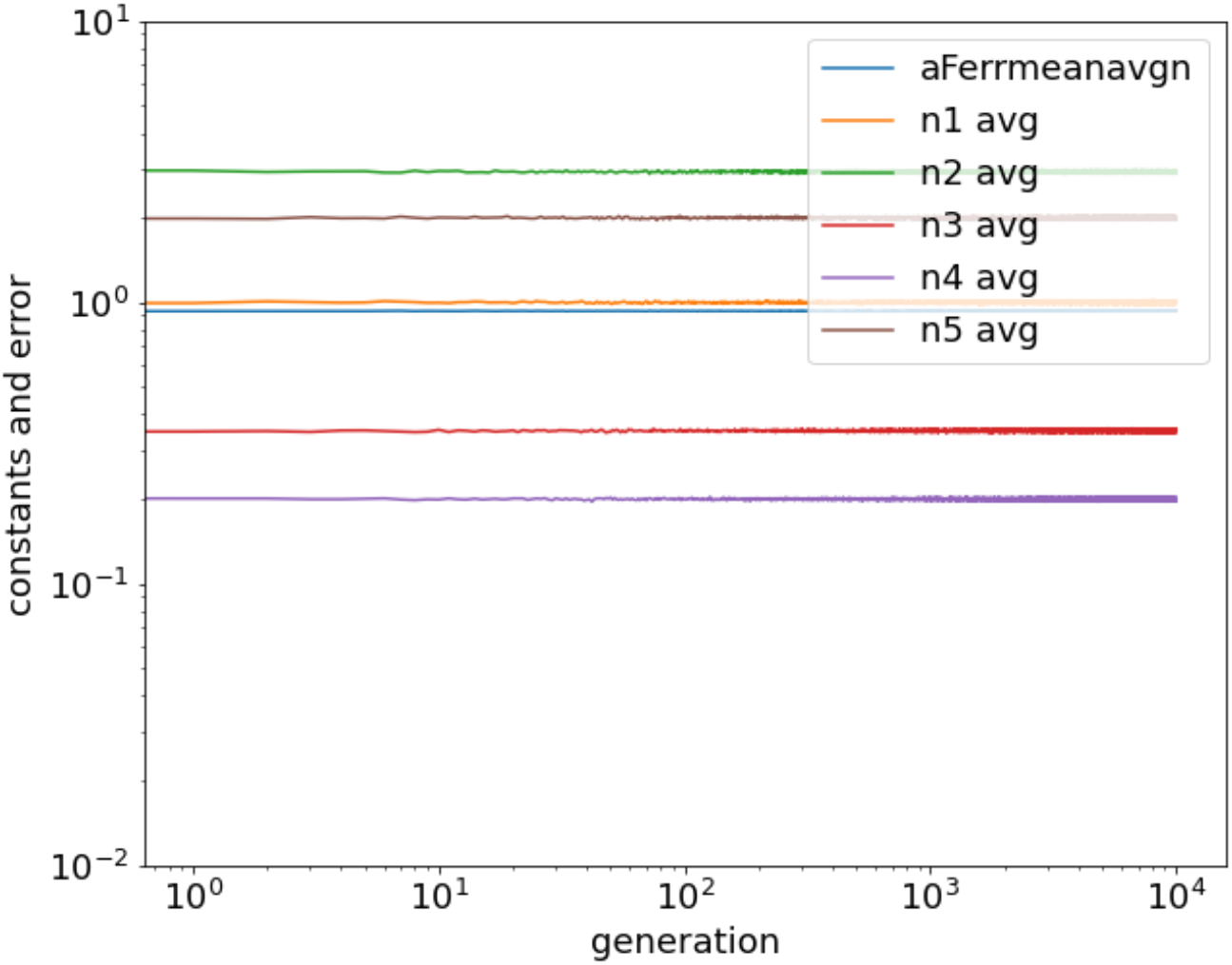
```
# now check each organism/solution to see if aFerr is less than cut of li
#if yes, store n for next generation population in ntemp, at end nkeep =
#and number of new offspring = NS-nkeep
#============================================
for j in range(NS):  # NS Ferr values, one for each solution in populatio
    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 to
#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 o
#============================================================
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 fixe
    #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.+0.09*2.*(0.5-numpy.random.
    else:
        ntemp[nkeep+j+1][1] = n[nmate2][1]*(1.+0.09*2.*(0.5-numpy.random.

    if (numpy.random.rand() < 0.5):
        ntemp[nkeep+j+1][2] = n[nmate1][2]*(1.+0.09*2.*(0.5-numpy.random.
    else:
        ntemp[nkeep+j+1][2] = n[nmate2][2]*(1.+0.09*2.*(0.5-numpy.random.

    if (numpy.random.rand() < 0.5):
        ntemp[nkeep+j+1][3] = n[nmate1][3]*(1.+0.09*2.*(0.5-numpy.random.
    else:
        ntemp[nkeep+j+1][3] = n[nmate2][3]*(1.+0.09*2.*(0.5-numpy.random.

    if (numpy.random.rand() < 0.5):
        ntemp[nkeep+j+1][4] = n[nmate1][4]*(1.+0.09*2.*(0.5-numpy.random.
    else:
```

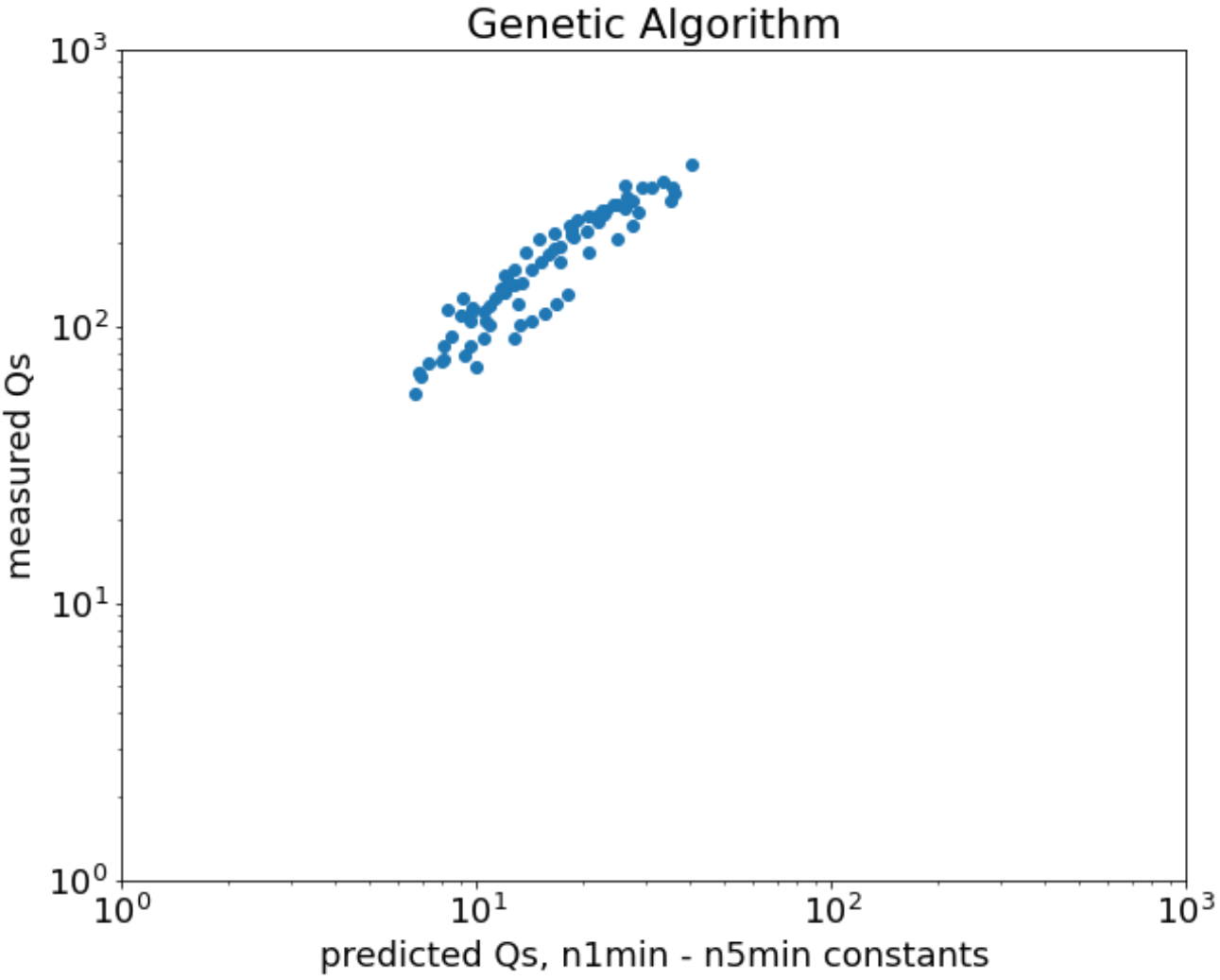
```
            ntemp[nkeep+j+1][4] = n[nmate2][4]*(1.+0.09*2.*(0.5-numpy.random.
        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][5] = n[nmate1][5]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][5] = n[nmate2][5]*(1.+0.09*2.*(0.5-numpy.random.
    #============================================
    n = deepcopy(ntemp)   # save ntemp as n for use in next generation (next


    '''AVERAGING OVER POPULATION AND OVER TIME, FINDING MINIMUM ERROR SET OF
    #============================================
    #initialoze average n's to zero and sum contribution of each member of th
    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;

    # aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of o
    #
    #============================================
    for i in range(ND):

        Ferravgn[i] = -1.*ndydata[i][0] + math.log(n1avg[k]) + n2avg[k]*ndyda
        Ferravgn[i] = Ferravgn[i] + n4avg[k]*math.log( ndydata[i][2] + n3avg[

        #aFerravgn[i] = abs(Ferr[i])/abs(ndydata[i][0])
        aFerravgn[i] = abs(Ferravgn[i])/abs(ndydata[i][0])
    #-------------
    aFerrmeanavgn[k] = np.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 is
    #============================================
```

```
    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], aFerrmeanavg
    #print ('kvalue =', k)
    '''end of evolution loop'''
    # ------------------------------------------------------------------
    # ------------------------------------------------------------------


# ------------------------------------------------------------------
#final print and plot of results
# ------------------------------------------------------------------
print('ENDING: pop. avg n1-n5,aFerrmean:', n1avg[k], n2avg[k], n3avg[k], n4av
print('MINUMUM:  avg n1-n5,aFerrmeanMin:', n1min, n2min, n3min, n4min, n5min,
print('TIME AVG:    avg n1-n5,aFerrmean:', n1ta, n2ta, n3ta, n4ta, n5ta, aFer

#Task 4 PLOTS

#========
#initialize values
qspred =  [[0.0]]
qsdata =  [[0.0]]

for i in range(ND-1):
    qspred.append([0.0])
    qsdata.append([0.0])

#calculate predicted and data values to plot
for i in range(ND):
    qspred[i] = n1min*(ndydata[i][1]**n2min) * ((ndydata[i][2]+n3min*ndydata[
    qsdata[i] = ndydata[i][0]

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

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

# aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of organ
# 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 avg'

# 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()

# data vs. predicted heat flux plot
plt.scatter(qspred, qsdata)
plt.title('Genetic Algorithm')
plt.xlabel('predicted Qs, n1min - n5min constants')
plt.ylabel('measured Qs')
plt.loglog()
plt.xlim(xmax = 1000, xmin = 1)
plt.ylim(ymax = 1000, ymin = 1)

plt.show()
```

```
ENDING: pop. avg n1-n5,aFerrmean: 1.0008865229084032 2.9052876283355134 0.3506
4345011020903 0.20083089416544694 2.0003284945030866 0.933346068513857
MINUMUM:  avg n1-n5,aFerrmeanMin: 0.9975315184812517 2.9800243408816467 0.3514
134388180167 0.19968005222876514 1.9823317095454989 0.9293996450214281
TIME AVG:    avg n1-n5,aFerrmean: 1.0002118973075826 2.915345319624403 0.35004
235360785496 0.20008150503668837 1.9997466857273827 0.9328725372026878
```


aFerrmeanavgn
n1 avg
n2 avg
n3 avg
n4 avg
n5 avg
constants and error
generation
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}

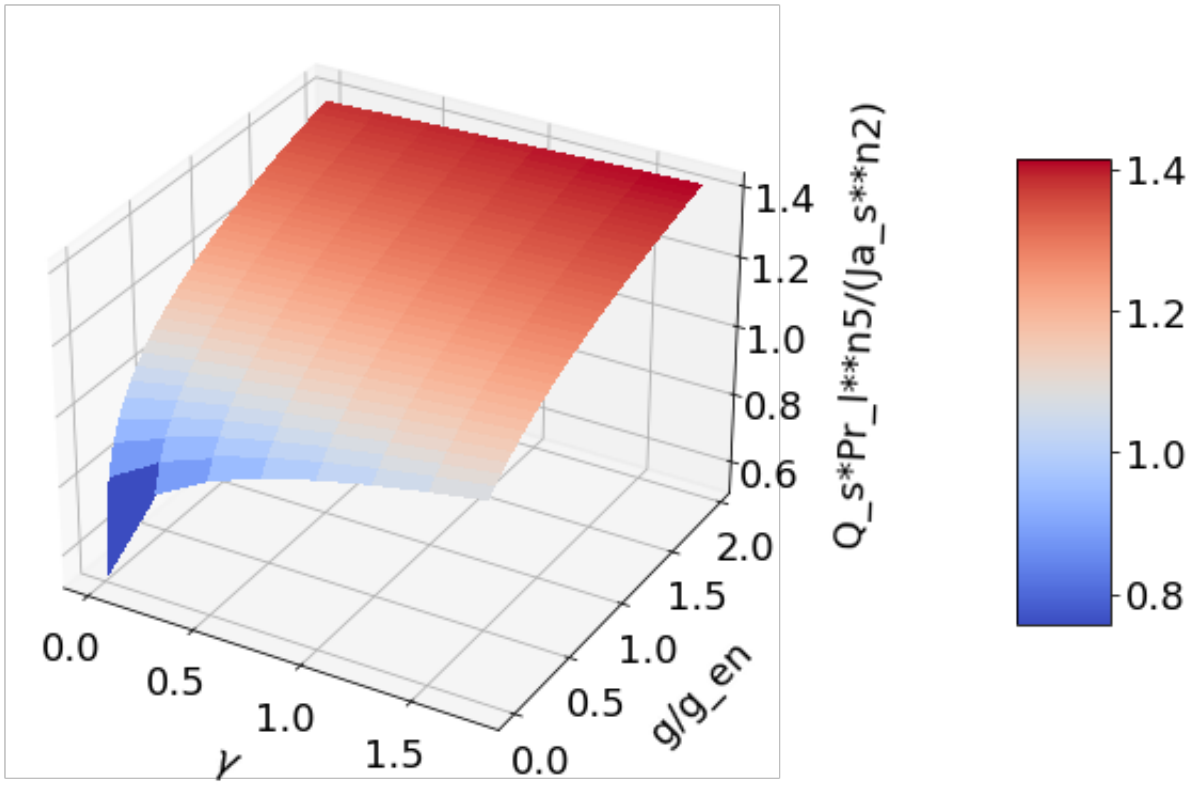


In [8]:
```
#Task 5 Part 1
import matplotlib.pyplot as plt

data = [[n1min, n2min, n3min, n4min, n5min],
        [n1avg[k], n2avg[k], n3avg[k], n4avg[k], n5avg[k]],
        [n1ta, n2ta, n3ta, n4ta, n5ta]]
rows = ('Minimum', 'Population Average', 'Time Average')
columns = ('n1', 'n2', 'n3', 'n4', 'n5')

fig, ax = plt.subplots()
ax.set_axis_off()
table = ax.table(cellText=data,
                 rowLabels=rows,
                 colLabels=columns,
                 rowColours =["palegreen"] * 10,
                 colColours =["palegreen"] * 10,
                 loc ='center')

plt.show()
```

	n1	n2	n3	n4	n5
Minimum	1.002821339355131	2.9897854303734923	0.34928119094999954	0.20166686010748558	1.98536497616805
Population Average	0.9984136508146513	2.9191938094867087	0.34665698315125326	0.20014107090280095	1.9855927292663735
Time Average	1.00011985756 43	2.934973182156045	0.35002714281341796	0.20004694099636128	1.9997018657847132

In [11]:
```
#Task 5 Part 2: surface plot
import numpy, scipy, scipy.optimize
import matplotlib
from mpl_toolkits.mplot3d import  Axes3D
from matplotlib import cm # to colormap 3D surfaces from blue to red
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import matplotlib.pyplot as plt

ndlydata = deepcopy(ndydata)
for j in range(DI):
    for i in range(ND):
        ndlydata[i][j]=math.log(ndydata[i][j]+0.000000000010)

def SurfacePlot(func, data, fittedParameters):
    f = plt.figure()
    axes = f.gca(projection='3d')

    # Make data mesh with appropriate ranges.
    x_data = data[0]
    y_data = data[1]
    z_data = data[2]

    X = np.arange(0.01, 2, 0.25)
    Y = np.arange(0.001, 2, 0.05)
    X, Y = np.meshgrid(X, Y)
```

```
    Z = func(numpy.array([X, Y]), *fittedParameters)

    surf = axes.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                             linewidth=0, antialiased=False)
    f.colorbar(surf, shrink=0.5, aspect=5, pad=0.2)
    #axes.scatter(x_data, y_data, z_data) # show data along with plotted surf
    axes.set_xlabel(r'$\gamma$', rotation=150)
    axes.set_ylabel('g/g_en')
    axes.set_zlabel('Q_s*Pr_l**n5/(Ja_s**n2)', rotation=60)
    axes.xaxis.labelpad=15
    axes.yaxis.labelpad=15
    axes.zaxis.labelpad=15

    plt.show()

def func(data, a, b, c):
    x = data[0]
    y = data[1]
    return a * (y + b*x)**c

xData = []
yData = []
zData = []

for x in range(len(lydata)):
    if 0.01 <= ndydata[x][2] <= 2:
        if 0.001 <= ndydata[x][3] <= 2:
            xData.append(ndydata[x][3])
            yData.append(ndydata[x][2])
            zData.append(qspred[x] * (ndydata[x][4]**n5avg[k]) / (ndydata[x][

data = [xData, yData, zData]

initialParameters = [n1avg[k], n3avg[k], n4avg[k]]
fittedParameters, covariance = scipy.optimize.curve_fit(func, np.asarray([xDa
SurfacePlot(func, data, fittedParameters)
```


1.4
1.2
1.0
0.8
0.6
Q_s*Pr_l**n5/(Ja_s**n2)
0.0
0.5
1.0
1.5
γ
0.0
0.5
1.0
1.5
2.0
g/g_en
1.4
1.2
1.0
0.8

In [ ]: