PHYC40970: Advanced Laboratory II

Newton's Cooling

Author: Lainey Ward
Student ID: 18365881
Supervisor: Dr. Luis Leon Vintro
November 15, 2022

Abstract

Newton's law of cooling states that the rate of temperature loss from a body is proportional to its temperature difference with its environment. Using a series of thermocouples and a heated metal rod, the temporal and spatial evolution of the system towards a non-equilibrium steady-state is investigated for four combinations of initial and boundary conditions. The influence of these conditions on the steady state itself and the rate at which the steady state is reached is discussed. The surface conductance of the rod is found to lie in the range $(3.25 - 3.30 \pm 0.55)\ m^{-1}$, consistent with the findings of Rafois and Ortin (1992). Finally, the temporal evolution of entropy production and transfer are shown to follow a sigmoid function, abiding by Prigogine's theorem of minimum entropy production. The influence of the initial and boundary conditions on each entropic term is also discussed. It is apparent from both the temperature and entropy distributions that the steady state is independent of the initial temperature distribution of the rod.

1 Introduction

In this experiment, Newton's original cooling experiment is reconstructed, however, using Rafois and Ortin's adapted experimental method.

In essence, a rod is heated at one of its ends and is in contact with an external bath elsewhere. Heat flows longitudinally through the rod by means of conduction and through the lateral surface by heat exchange with the bath. Newton's law of cooling states that the normal heat flow through the rod's surface is proportional to its difference in temperature with the external bath. [1]

Thermocouples are positioned along the rod in accordance with Lurié and Wasenberg's discrete hypothesis of local equilibrium. From this, temporal and spatial distributions of temperature are constructed to show the evolution of the system in accordance with Newton's cooling, and towards a non-equilibrium steady-state.

Other fundamental concepts of thermodynamics such as the influence of initial and boundary conditions on the steady-state, the rate at which the steady state is attained, the surface conductance and the temporal evolution of entropy, are examined.

2 Theory

2.1 Newton's Cooling

Newton presented his cooling experiment in 1701, in which he placed different metal rods on a hot iron bar and noted the time it took for them to solidify. He deduced the temperature of the rods from their cooling time. According to Newton: [2]

> "...the air heated by the iron might be always carried off by the wind... thus equal parts of the air heated in equal times and received a degree proportional to the heat of the iron"

Although Newton did not derive Newton's law of cooling, his contributions to thermodynamics was pivotal to its eventual formation by Fourier in 1807. The equation for this law of cooling is often given in its non-dimensional form: [1]

$$T(t) = [T_L - T_R]\, e^{-\beta t} + T_R \tag{1}$$

with $T(t)$, the temperature of the object at a given time, T_L, the heater temperature, T_R, the bath temperature, and β, the heat transfer coefficient.

According to this equation, the temporal distribution of the rod depends only on the boundary conditions imposed at the ends on the conductor and β, which is also defined as follows: [3]

$$\beta = \left[\frac{2H}{\lambda r}\right]^{1/2} \tag{2}$$

with λ, the systems thermal conductivity, r, the radius of the rod, and H, the surface conductance.

Evidently, β itself is dependent on the media constituting the rod and bath, as well as the dimensions of the rod. In the limit where $H \to 0$, the surface is perfectly insulating and for $H \to \infty$, it is perfectly conducting. [3] [4]

2.2 Steady-State

Due to the imposed temperature difference, the system cannot relax to equilibrium. Rather, it tends to a final non-equilibrium stationary state; a time-independent state characterised by a nonzero transport of energy and a non-uniform temperature distribution through the rod. [1]

The length of the rod is significantly larger than its radius, and so it can be considered as one-dimensional. Diaz-Guilera uses the one-dimensional variant of Eq. 1, and forms a second-order differential equation from its steady-state solution. The solution to this equation is the steady-state temperature distribution at a position z: [4]

$$T(z,\infty) = [T_L - T_R]\, e^{-\beta z} + T_R \tag{3}$$

Thus, it is possible to determine the value of β from a system's steady-state distribution, as shown in Section 4. Diaz-Guilera also notes that in the case of a heated rod, the steady-state temperature distribution can also be described by a sinh function. [4]

2.3 Entropy

In classical thermodynamics, entropy is defined only for states of thermodynamic equilibrium. In order to generalise the concept of entropy to the non-equilibrium state, an assumption of local equilibrium is adopted. According to this assumption, thermodynamic equilibrium dominates locally for a given elemental volume. These elements must be sufficiently large to allow for properties like temperature and pressure to be defined, yet sufficiently small so that such properties do not vary significantly across the element. For this reason, the rod is considered as a series of volume elements. [1] [3]

The entropy balance equation defines the temporal change in entropy as follows: [1]

$$\frac{dS}{dt} = \frac{d_i S}{dt} + \frac{d_e S}{dt} \tag{4}$$

With $d_i S/dt$, the internal entropy production, and $d_e S/dt$, the external entropy transfer.

Using the entropic continuity equation, a Gibbs equation and Fourier's law, Lurié and Wagensberg produce equations for the time evolution of both entropy production and transfer in terms of the temperature of the volume elements. In the analysis, we opt for Rafois and Ortin's version which incorporates a β-dependent components for $d_e S/dt$, and are rescaled to depend only on the length of the volume elements. In terms of the experimental configuration in Fig. 1, these equations are: [3]

$$\frac{d_i S}{dt} \propto -\frac{1}{l}\sum_{i=0}^{16}(T_{i+1} - T_i)\left(\frac{1}{T_{i+1}} - \frac{1}{T_i}\right) \tag{5}$$

$$\begin{aligned}\frac{d_e S}{dt} \propto &-\frac{1}{l}\frac{1}{T_0}(T_1 - T_0) + \frac{1}{l}\frac{1}{T_{17}}(T_{17} - T_{16}) \\ &- \beta^2 l \sum_{i=1}^{16}\left(1 - \frac{T_{17}}{T_i}\right)\end{aligned} \tag{6}$$

where T_n represents the temperature of a given volume element, in which $n = 0$ and $n = 17$ represent the heater and ambiance elements, respectively.

2.4 Minimum Entropy Production

Prigogine's minimum entropy production principle explains the evolution of the entropy production. Formulated in 1947, it states that in the non-equilibrium stationary state, the entropy production assumes a constant minimum value in accordance with the system's boundary conditions. For example, in the case where the system can relax to equilibrium, the entropy production would obtain a minimum value of zero. [1] [5]

The principle states that all thermodynamic quantities are constant in a stationary state. In the case of total entropy this equates to:

$$\frac{dS}{dt} = 0 \tag{7}$$

In combination with Eq. 4, this implies that the internal entropy production equals the external entropy transfer.

3 Experiment

3.1 Configuration

The experiment is undertaken with a heater, a hotplate, a metal rod, sixteen thermocouples, a thermocouple amplifier, a data acquisition device (DAQ), and a desktop equipped with Jupyter Notebook, a Python platform. A schematic of the apparatus is shown in Fig. 1.

There are small slots along the length of the rod, each corresponding to a volume element, separated by a mean distance of 5 *cm*. The exposed ends of the thermocouples are inserted into their respective slots. These are coated in thermal paste to improve the thermal conductivity between the rod and thermocouples.

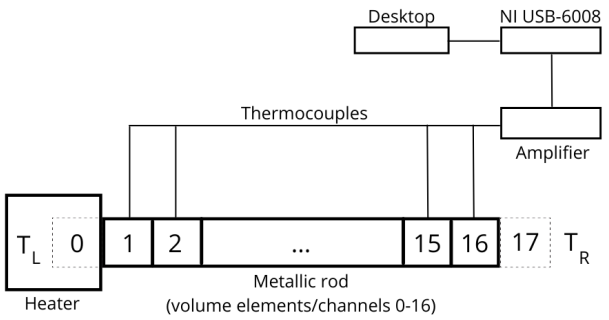


Fig. 1: Schematic of the experimental apparatus. Volume elements zero and seventeen correspond to the heater and ambiance.

The opposite end of each thermocouple is connected to its corresponding channel on the amplifier, which is connected to the DAQ.

Python code is devised in the laboratory to acquire the voltage data obtained by the NI-USB DAQ. The DAQ can only sample eight channels at a given time and so $0\ V$ and $-5\ V$ are applied before sampling the first and second set of eight channels, respectively. There is a negligible time delay between the sampling of the two sets.

The heater features a temperature control which operates in the range $0 - 1000°C$.

3.2 Method

The experiment is undertaken in two stages: calibration and data collection.

First, the thermocouples are calibrated. They are placed in a beaker of ice and water, positioned on hotplate. The Python code is run to sample a single data point, while a digital thermometer is used to record the water temperature. This process is repeated until the water reaches a temperature of $\sim 100°C$. The thermometer is constantly stirred to ensure the uniform circulation of water throughout the beaker. A voltage-temperature calibration curve is constructed using the data collected.

Next, the thermocouples are inserted back into their respective slots along the rod. Voltage data are collected for four instances, each a combination of one of two initial conditions (uniform (U) and non-uniform (N) heating) and heating temperatures (low (S) and high (L)). Non-uniform temperature (N) is achieved by heating a section of the rod using a hairdryer on a retort stand. The four instances are indicated using the abbreviations U.S., N.S., U.L. and N.L..

The Python code is run for each instance, at a sampling frequency of $1/60\ S/s$ for 70 samples, corresponding to a run-time of seventy minutes. It is important to allow the rod to cool to room temperature between the four runs, which may take up to three hours.

The length and radius of the rod, as well as the positions at which the slots occur are measured using a meter stick.

4 Results & Analysis

The data obtained from the calibration and data collection stages are discussed in the following sections.

4.1 Calibration

A voltage-temperature plot is produced in Fig. 2 from the data collected in the calibration stage. The uncertainty on readings from the DAQ and the digital thermometer are taken as $0.05\ V$ and $0.5°C$, respectively.

The is a significant, strong correlation ($r > .997,\ p < .05$) between temperature and voltage across all channels.

Using an ODR model, a linear function is fitted to the data set corresponding to each channel. The sixteen resulting regressions are plotted in Fig. 2. Due to their high degree of similarity, the individual functions are difficult to visually resolve.

There is both a strong, significant correlation and a high coefficient of determination between each set of data and its linear regression ($r > .991,\ p_r < .05,\ r^2 > .983$), implying that the functions fit the data well.

A general calibration function is using the mean of the sixteen fitting parameters, as well as their respective uncertainties. This function is approximately:

$$T = (28.2 \pm 0.1)V + (21.0 \pm 0.1) \tag{8}$$

4.2 Instances 1-4

The Python code is run for the four instances, U.S., N.S., U.L. and N.L.. The voltages obtained from the DAQ

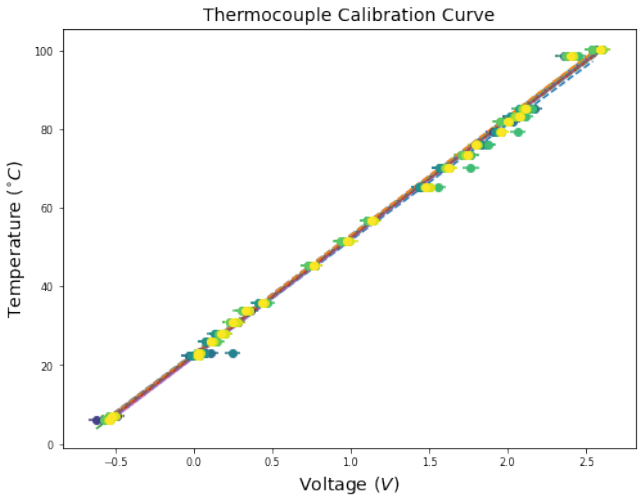


Fig. 2: Calibration data and corresponding linear regressions for channels one-sixteen.

are converted into temperature using Eq. 8. The 0.05 V voltage uncertainty translates to a mean temperature uncertainty of $1.42°C$ for all channels of all instances. The uncertainty on position and time are negligible.

For each instance, a 3D plot is produced of position, temperature and time, as shown in Fig. 3. Each individual channel's data are represented as slice planes, with channels one-sixteen following a light-dark gradient

The U.S. and U.L. plots begin with a mean temperature of $(17.7 \pm 1.4)°C$ and $(19.8 \pm 1.4)°C$. The N.S. and N.L. plots also tend to begin at a higher temperature of at least $23°C$. This can be attributed to the general increase in ambient temperature from the hairdryer, as well as the fact that the rod was not fully cooled to room temperature following the data collection runs preceding N.S. and N.L..

The maximum temperatures of the plots are discussed detail in Section 4.3.

The temperature distributions in Fig. 3 follow an overall sigmoid function, across all channels and instances. However, the distribution can be considered exponential before the temperature plateau. This is in line with Newton's law of cooling in Eq. 1. It is also apparent that the function's height and curvature are inversely proportional to position, and the length of the leading tail is proportional to position.

The N.S. and N.L plots contain a temperature peak between channels zero-eight due to the heating of the hairdryer. This non-uniform distribution is only visible in the range $0 - 20$ mins. There is also a temperature peak in the N.L. plot across channels five-eight at $t \approx 65$ mins due to physical contact when adjusting the thermocouple ends.

The temporal distribution of each channel begins to plateau at approximately $20 - 30°C$ in all four instances. Here, each channel has effectively reached a steady-state and becomes independent of temperature. The rate at which the steady-state is reached. referred to as the steady-state rate, is influenced by both the initial and boundary conditions. This phenomenon is best illustrated by Fig. 4, which features a series of overlaid temperature distributions.

By visual examination of the knee of each curve, it is apparent in Fig. 4b that N.L. reaches its steady-state earlier than U.L., for all channels. Similarly, in Fig. 4a, N.S. also reaches its steady-state earlier than U.S., however, only for channels one-twelve. The difference in steady-state rates is unclear for channels twelve-sixteen.

It should be noted that the non-uniform initial heating in Fig. 4a is to a lower temperature than that of Fig. 4b. This may account for why the difference in steady-state rates is much less pronounced in the former than the latter.

Furthermore, it is apparent in Fig. 4c that U.S. reaches its steady-state earlier than U.L.. This is also the case in Fig. 4d, in which N.S. reaches steady-state earlier than N.L.

Overall, examination of the temperature distributions reveal that the rate at which the steady-state is reached is influenced by both the initial and boundary conditions imposed on the rod. The steady-state is reached earliest for non-uniform heating with a low heater temperature, and latest for uniform heating with a high heater temperature.

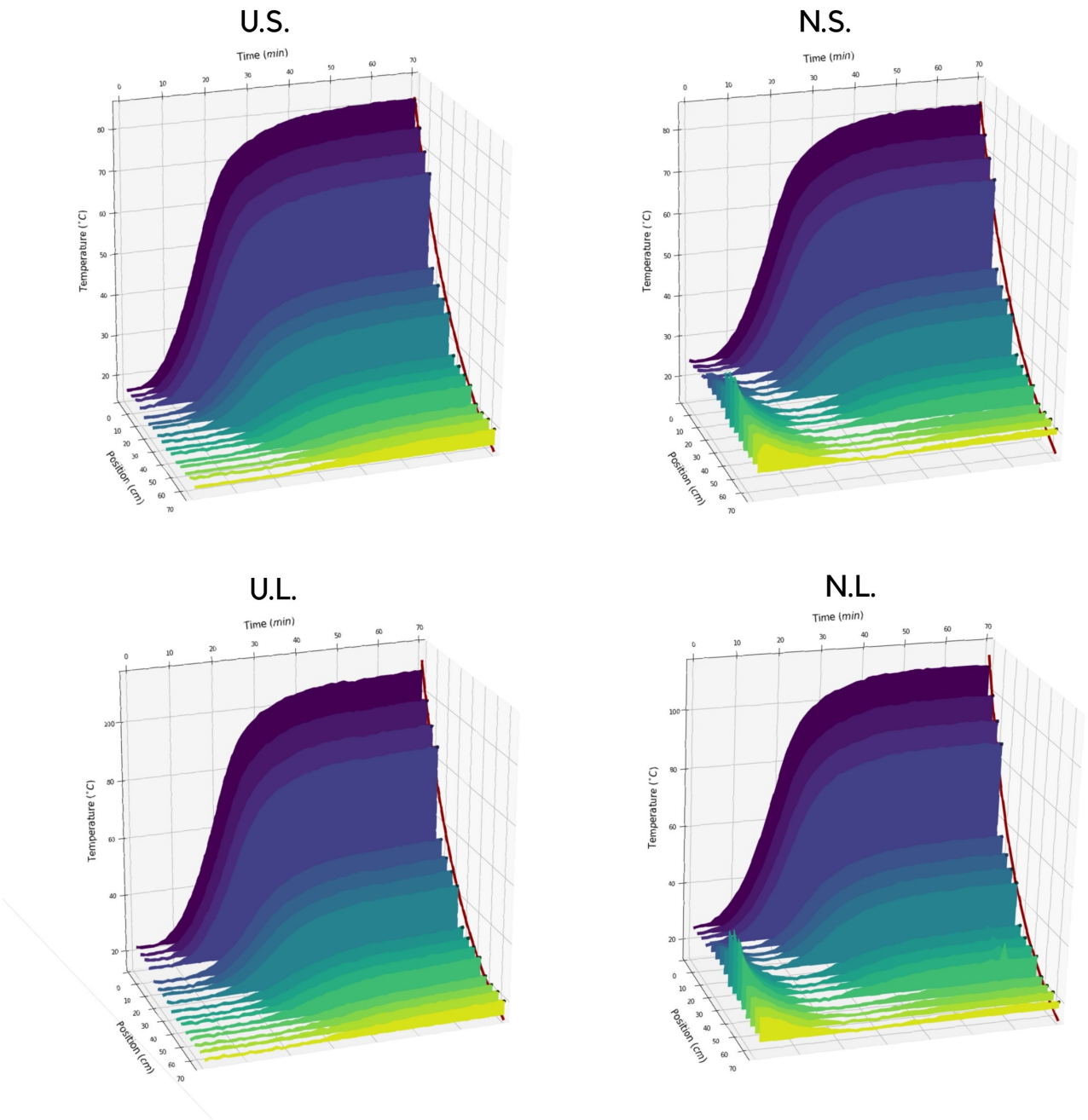
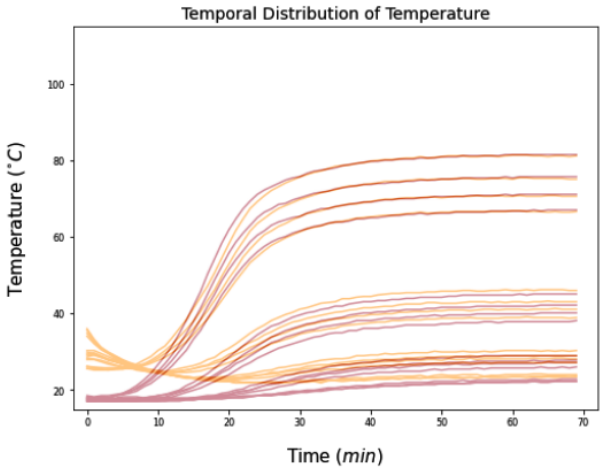
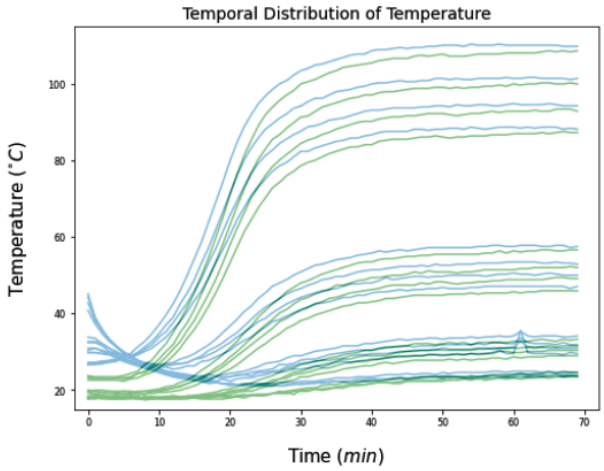


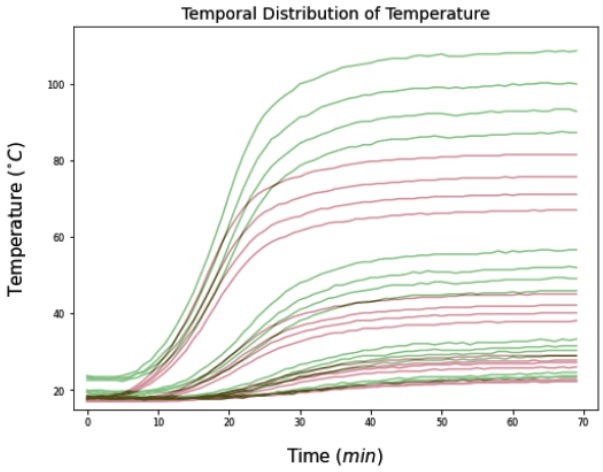
Fig. 3: Temporal evolution of the temperature distribution for the four instances. Note the difference in scale of the y-axis between the top and bottom rows.



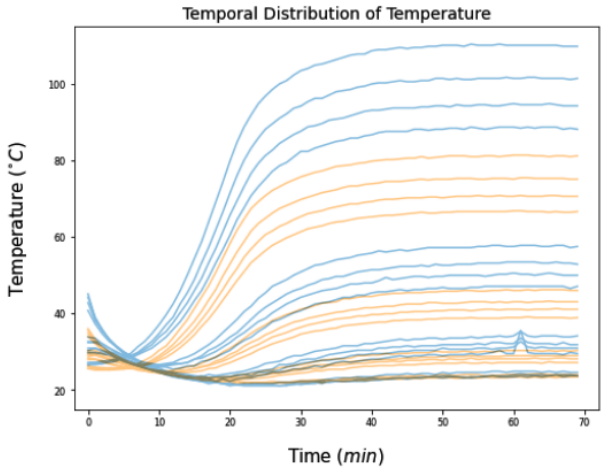
(a) U.S. (red) and N.S. (orange).



(b) U.L. (green) and N.L. (blue).



(c) U.S. (red) and U.L. (green).



(d) N.S. (orange) and N.L. (blue).

Fig. 4: Various temperature distributions overlaid for the purpose of comparison. (a) and (b) each depict two plots in which the heating temperature is the same, and (c) and (d) also each depict two plots in which the initial conditions are the same.

4.3 Steady-State

The final temperature distribution planes i.e. the steady-states, are plotted for each instance in Fig. 5. These plots are fitted to an exponential function of the form:

$$T(t = 70) = A \exp(-Bx) + C \tag{9}$$

where A, B, and C are fitting parameters given in Table 1.

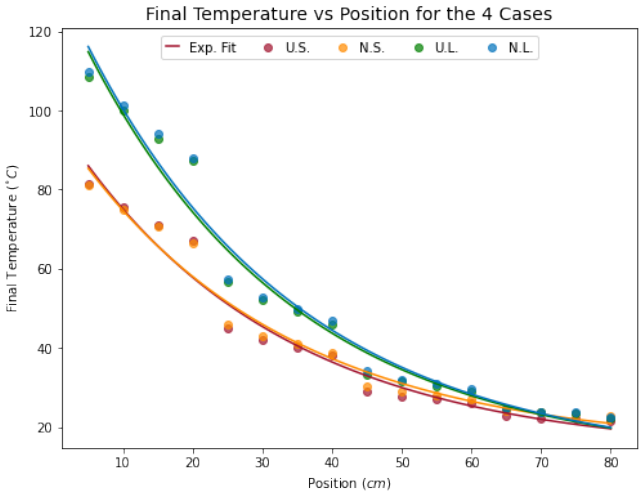


Fig. 5: The final steady-state planes, also indicated in Fig. 3.

	A ($^\circ C$)	B (m^{-1})	C ($^\circ C$)
U.S.	85.6 ± 4.3	3.27 ± 0.57	13.3 ± 4.7
N.S.	83.0 ± 4.0	3.27 ± 0.55	14.3 ± 4.4
U.L.	122.5 ± 6.0	3.30 ± 0.55	11.0 ± 6.4
N.L.	124.1 ± 6.0	3.25 ± 0.55	10.7 ± 6.6

Table 1: Fitting parameters corresponding to Fig. 5.

In all instances, the steady-states appears to be well-described by the decreasing exponential function. In particular, the two high temperature instances have very similar regression fits ($r^2 > .997$), regardless of their differing initial temperature conditions. The same phenomenon is seen for the two low temperature instances ($r^2 > .999$). For reference, ($r^2 < .6$) for any other combination of regressions e.g. U.S. and N.L.. There is also a low percent difference between the fitting parameters of the discussed instances, as shown in Table 1.

This implies that the steady-state distributions depend only on the boundary conditions, and not on the initial temperature conditions. The temperature distribution effectively loses memory of its initial conditions over time.

The parameter B in Table 1 represents the coefficient β in Eq. 2. This parameter tends to be significantly higher for high temperature instances. The parameter A in Table 1 represents the temperature at a position of $0\ m$, the position of the heater.

In accordance with Eq. 3, the three fitting parameters can be used to determine T_R, T_L and β. T_R corresponds to C, β to B, and T_L to the addition of A and C. The T_L values are given in Table 2.

	T_L ($^\circ C$)	H (W/Km^2)
U.S.	99.0 ± 5.3	177.8 ± 56.2
N.S.	99.0 ± 5.0	177.5 ± 55.4
U.L.	133.5 ± 7.0	180.5 ± 56.3
N.L.	134.8 ± 0.7	175.4 ± 54.8

Table 2: Heater temperatures and surface conductances determined from Table. 1.

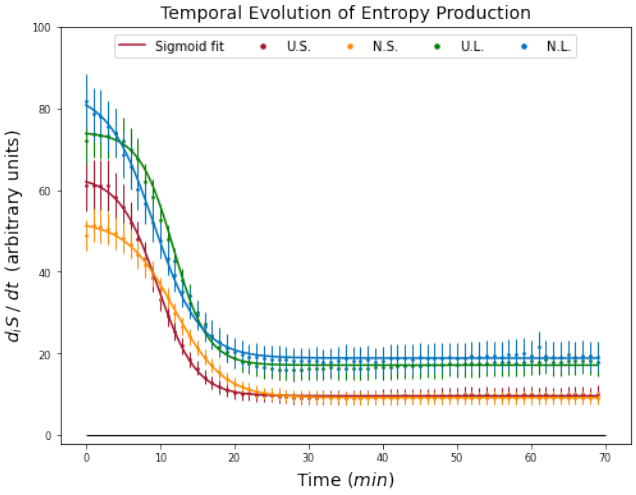
The mean ambiance temperature for U.S. and N.S. is $(13.8 \pm 3.2)^\circ C$, and 10.9 ± 5.6 for U.L. and N.L.. Both these values are highly consistent with one another, and the small temperature disparity can be attributed to the fact that the U.L. and N.L. instances were completed on a separate day to U.S. and N.S.

The mean heater temperature for former and latter are $(84.3 \pm 2.9)^\circ C$ and $(123.3 \pm 4.2)^\circ C$. However, the heater was actually set to a temperature of ~ 100 and $400^\circ C$, respectively. This indicates that there is an inconsistency between the calibration of the heater's temperature and its control dial. Regardless, the true temperature of the heater is determined by extrapolation in Section 4.3.

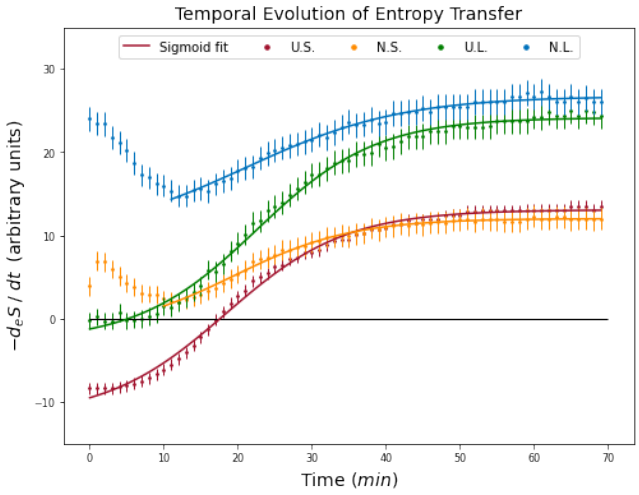
The heat transfer coefficient β is found to lie in the range $(3.25 - 3.30 \pm 0.55)\ m^{-1}$. These values are comparable with those founded by Rafois and Ortin, which lie in the range $(6.5 - 7.0 \pm 0.4)\ m^{-1}$. As discussed in Section 2.1, this discrepancy can be attributed to the different medium and dimensions of the rod used in both experiments.

There appears to be no clear influence of the boundary and initial conditions on the value of β. Rafois and Ortin report higher β values for high heater temperatures, however, this cannot be inferred from the experimental findings.

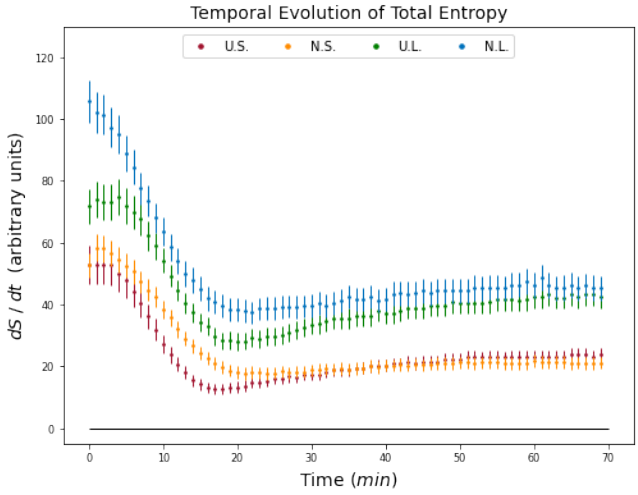
The surface conductance of the rod is calculated for each instance using β and Eq. 2, and the resulting values are given in Table 2. The metal type of the rod is unknown and so λ is presumed to lie in the range of aluminium and graphite at $(237 \pm 60)\ W/mK$.[6] The radius of the rod is measured as $(0.14 \pm 0.01)\ m$.



(a) Entropy production.



(b) Entropy transfer.



(c) Total entropy.

Fig. 6: Temporal evolution of entropic terms from Eq. 4, 5, and 6.

4.4 Entropy

The total entropy, internal entropy production, external entropy transfer are determined for each of the four instances using Eq. 4, 5, and 6, and plotted against time in Fig. 6. The heater and ambiance temperatures from the previous section are substituted into these equations.

Following the first fifteen minutes, all entropic terms evolve monotonously towards a minimum. At the time of steady-state ($\sim 20-30$ mins), they each plateau, effectively becoming independent of time. This is consistent with the Prigogine's minimum entropy production theorem.

The entropy production and transfer plots are fitted to a sigmoid function in the form:

$$\frac{dS}{dt} = A\left[1 + \exp\left(B(x+C)\right]^{-1} + D \qquad (10)$$

where A, B, C and D are fitting parameters given in Table 3 and 4 in the Appendix.

In all instances, the temporal evolution appears to be very well-described by the sigmoid function. This is also supported by the low relative errors associated with each plot's fitting parameters.

The influence of initial and boundary conditions conditions are apparent in the three entropic plots. In Fig. 5, d_iS/dt is highest for N.L. and U.L., the instances of high heater temperatures. The converse is true for the instances of low heater temperatures. The instances of non-uniform initial heating, N.S. and N.L., also begin at higher values of d_iS/dt, than their respective uniform counterparts. At the steady state, the instances of same heater temperatures plateau at approximately the same values. As in the previous section, it is apparent that the change in entropy production loses memory of the initial temperature conditions imposed on the rod.

In a similar manner, $-d_eS/dt$ is highest for N.S. and N.L., the instances of non-uniform heating. Due to their higher channel temperatures, the rate of entropy transfer increases before decreasing and reaching a plateau. Again, the system loses memory of the initial conditions.

The plots in Fig. 5 appears to have begun plateauing. However, the plots still show a slight increase, and thus have a slight dependence on time. This indicates that a true steady-state had not been reached. Therefore, the experiment should have been conducted for a longer run-time of ~ 90 minutes.

It is also notable that the entropy production plot is positive, and in no cases negative, as anticipated from the second law of thermodynamics. The entropy transfer plot contains both positive and negative data points, indicating that it can either increase or decrease locally.

5 Improvements

Numerous changes to the experimental apparatus and procedure could theoretically increase the precision and accuracy of experimental results.

In terms of apparatus, a stirrer hotplate would improve the convection of water in the calibration stage, and thus improving the calibration accuracy. Increasing the number of probes would also increase the number of data points in Fig. 5, allowing for a more accurate determination of the beta coefficient.

For optimum results, these probes should be positioned between channels eight and nine, and twelve and thirteen. However, due to the DAQ's limited channel capacity, this would require either a new DAQ device or an second run of each instance, encompassing the additional thermocouples. The latter this would add over three hours to the run-time of each instance, and is not feasible for a time-sensitive laboratory experiment.

In terms of procedure, there is capacity in the code to record multiple readings per thermocouple, at a given time, with the average being taken as the final reading. Although excessive, it may result in more accurate results.

The hairdryer should also have been left to heat the rod for a longer period of time in both instances of N.S. and N.L.. This would have increased the prominence of the influence of non-uniform initial conditions in Fig. 3, 4, and 6.

The difference between the low and high heater temperatures should have also been increased for the same reason. Due to the DAQ's range of $0-5\ V$, the greatest temperature range that could have been measured is $(\sim 28-162^{\circ}C)$.

6 Conclusion

In conclusion, a simple thermodynamic system consisting of a metal rod, a heating device and a series of thermocouples is used to investigate a wide range of thermodynamic properties and processes, including Newton's law of cooling and non-equilibrium steady states.

Firstly, the DAQ voltages are successfully converted into temperature readings by averaging linear regressions across all of the thermocouple channels.

Using the experimental data, temporal and spatial temperature distributions are formed for four combinations of initial (uniform and non-uniform heating) and boundary (high and low heater temperature) conditions. The temporal temperature distribution follows an exponential function, in accordance with Newton's law of cooling.

It is also shown that these plots plateau, effectively reaching a time-independent non-equilibrium steady state. Through analysis of the knee of each plot, it is found that the rate at which the steady state is reached, is influenced by both the initial and boundary conditions. The steady-state is reached earliest for non-uniform heating with a low heater temperature and latest for the opposite conditions.

The final spatial temperature distribution planes are fitted to an exponential function, in accordance with the steady-state equation derived by Diaz-Guilera. From the fitting parameters, the temperature of the heater and ambiance are found to be approximately 99 and $114°C$. The heat transfer coefficient also lies in the range $(3.25 - 3.30 \pm 0.55)\ m^{-1}$. No relation is found between the value of β and the initial and boundary conditions.

The low r^2 values between the U.S. and N.S., and U.L. and N.L. exponential fits reveal that the steady state is independent of the initial temperature conditions.

Finally, the temporal evolutions of the change in total entropy, entropy transfer and entropy production are plotted. It is shown that these terms evolve monotonously in a sigmoidal fashion, attaining a minimum at the time of steady-state, in accordance with Prigogine's theorem.

The initial heating and boundary conditions are shown to influence the entropy transfer and production plots at early times. Once again it is demonstrated that only the boundary conditions influence the minimum value of the steady state, as it is independent of the initial conditions.

Overall, the findings from the experiment abide with that predicted by various laws of thermodynamics. Despite the high degree of accuracy associated with the findings, the experimental method and apparatus leave scope for improvement.

References

[1] D. Kondepudi and I. Prigogine, *Modern Thermodynamics*, ch. 15-17, pp. 333–400. John Wiley & Sons, 2002.

[2] R. Winterton, "Early study of heat transfer: Newton and fourier," *Heat Transfer Engineering*, vol. 22, pp. 3–7, 2001.

[3] R. Ismael and J. Ortin, "Heat conduction in a metallic rod with newtonian losses," *Am. J. Phys*, vol. 60, pp. 846–851, 1992.

[4] A. Diaz-Guilera, "On heat conduction in one-dimensional solids," *Am. J. Phys*, vol. 58, pp. 779–780, 1990.

[5] L. David and J. Wasenberg, "Concepts of nonequilibrium thermodynamics in discrete model of heat conduction," *Am. J. Phys*, vol. 48, pp. 868–872, 1980.

[6] "Metals, metallic elements and alloys - thermal conductivities."
url: https://www.engineeringtoolbox.com/thermal-conductivity-metals-d_858.html.

7 Appendix

	-A	-B	-C	D
		(10^{-2})		
U.S.	53.6 ± 0.4	38.4 ± 1.1	9.7 ± 0.1	63.3 ± 0.4
N.S.	43.2 ± 0.3	30.3 ± 0.7	11.9 ± 0.1	52.4 ± 0.3
U.L.	56.9 ± 0.5	43.0 ± 2.5	11.8 ± 0.2	74.1 ± 0.5
N.L.	65.1 ± 1.2	31.9 ± 1.4	9.2 ± 0.3	84.1 ± 1.1

Table 3: Fitting parameters corresponding to Fig. 6a.

	A	-B	-C	-D
		(10^{-2})		
U.S.	24.9 ± 0.6	12.3 ± 0.5	18.4 ± 0.5	11.8 ± 0.5
N.S.	13.7 ± 0.8	12.6 ± 0.7	19.6 ± 1.0	1.6 ± 0.8
U.L.	26.9 ± 0.6	12.4 ± 0.5	22.6 ± 0.4	2.7 ± 0.5
N.L.	-17.12 ± 1.6	-9.5 ± 0.8	21.1 ± 1.9	-26.7 ± 0.2

Table 4: Fitting parameters corresponding to Fig. 6b.

Newton's Cooling: Calibration Curve

Import Modules

In [40]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.odr import ODR, Model, Data, RealData
from scipy.stats import pearsonr
from scipy.stats import t as studentt
from sklearn.metrics import r2_score
import matplotlib.ticker as mticker
from matplotlib import cm
import matplotlib.pylab as pl
import csv
```

Import Data

In [41]:
```
# Temperatures recorded in the lab.
T = np.array([65.2,70.1,73.6,76.0,79.4,82.0,83.2,85.2,98.6,100.3,22.4,25.9,28.0,30.9,33.8,35.9,45.3,51.7,56.9,7.0,6.0,23.1])
```

In [42]:
```
# Create dictionary for channel voltages.
Channel_Voltages = {}

# Name dictionary keys according to channel number.
for i in range(1,17):
    key_i = 'Channel_{}'.format(i)
    Channel_Voltages.setdefault(key_i, [])
```

In [43]:
```
# Import data as lists into corresponding dictionary keys.
for i in range(1,23):
    subfilename = "Session_2\\1to16_Calibration_Run" + str(i) + ".txt"
    with open(subfilename, newline='') as subfile:
        n = 1
        for row in csv.reader(subfile):
            Channel_Voltages['Channel_{}'.format(n)].append(row[0])
            n += 1
```

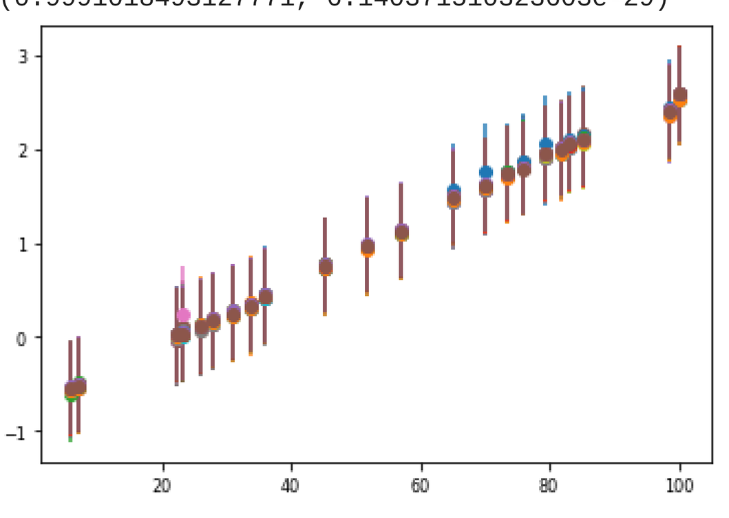
In [44]:
```
# Convert strings to floats.
for i in range(1,17):
    for n in range(0,22):
        Channel_Voltages['Channel_{}'.format(i)][n] = float(Channel_Voltages['Channel_{}'.format(i)][n])
```

Preliminary Plot

In [45]:
```
T_err = 0.5
V_err = 0.05
```

In [46]:
```
# Check voltage data.
for i in range(1,17):
    plt.errorbar(T,Channel_Voltages['Channel_{}'.format(i)],T_err,V_err,marker='o', linestyle='None')
    print(pearsonr(T,Channel_Voltages['Channel_{}'.format(i)]))
```

```
(0.9992943566161934, 5.5067765948652854e-30)
(0.9991958329532691, 2.0339128747250982e-29)
(0.9992085603460226, 1.7340846812330782e-29)
(0.9993138540465811, 4.161447629773989e-30)
(0.9989383088108452, 3.2688428194068157e-28)
(0.9984153325438286, 1.7900436613944494e-26)
(0.9978037352876401, 4.669190565571426e-25)
(0.9991433399173085, 3.827027938871641e-29)
(0.9989294338007402, 3.5524780643068228e-28)
(0.9988552721244611, 6.938840340474355e-28)
(0.9976218540923955, 1.0338470023046990e-24)
(0.9988416535699083, 7.80952729789116e-28)
(0.9991871634923641, 2.264051097235327e-29)
(0.9993269850417614, 3.4304403645927674e-30)
(0.9992216845585651, 1.4671416027683857e-29)
(0.9991018493127771, 6.140371510323603e-29)
```



Note: Would have been easier to fit one function to ALL of the data.

Fitting Functions

In [47]:
```
def linear(beta, x):
    M,C = beta
    return M*x + C
```

In [48]:
```
def fitting_func(xdata, ydata, xerr, yerr, M_guess, C_guess):

    # Create a model.
    DV_model = Model(linear)

    # Create a RealData object.
    data = RealData(xdata, ydata, xerr, yerr)

    # Create ODR with model and data.
    beta0=[M_guess, C_guess] # Guess beta values.
    odr = ODR(data, DV_model, beta0)
    odr.set_job(fit_type=0)

    # Run the regression.
    out = odr.run()

    # Create array of numbers with same interval as x.
    x_fit = np.linspace(np.min(xdata), np.max(xdata), len(xdata))

    # Run the fit function using x_fit and regression values of beta.
    y_fit = linear(out.beta, x_fit)

    return x_fit, y_fit, out.beta, out.sd_beta
```

Apply Fit Function to Each Channel

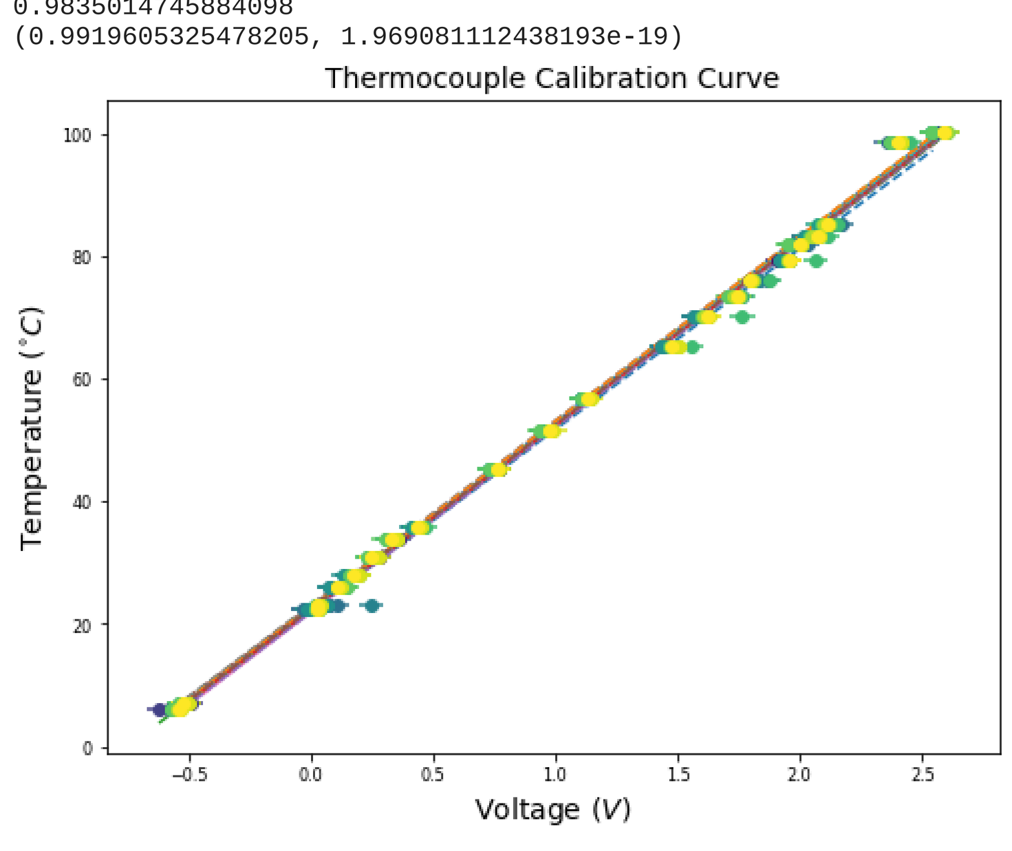
In [49]:
```
X_Fit = []
Y_Fit = []
Param = []
Param_Err = []
```

In [50]:
```
# Plot voltage data
for i in range(1,17):
    x_fit, y_fit, param, param_err = fitting_func(Channel_Voltages['Channel_{}'.format(i)], T, V_err, T_err, 0.03, -0.6)
    X_Fit.append(x_fit)
    Y_Fit.append(y_fit)
    Param.append(param)
    Param_Err.append(param_err)
```

In [51]:
```
T_sorted = sorted(T)
```

In [52]:
```
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111)
colors = pl.cm.viridis(np.linspace(0,1,17))

# Plot each channel data and fit function.
for i in range(1,17):
    plt.errorbar(Channel_Voltages['Channel_{}'.format(i)], T, xerr=V_err, yerr=T_err,
                 linestyle='None', marker='o', color=colors[i])
    plt.plot(X_Fit[i-1], Y_Fit[i-1], linestyle='dashed')

    print(r2_score(Y_Fit[i-1],T_sorted))
    print(pearsonr(T_sorted,Y_Fit[i-1]))


plt.ylabel('Temperature $(^{\circ}C)$')
plt.xlabel('Voltage $(V)$')
plt.title('Thermocouple Calibration Curve', fontsize = 14)
plt.rc('figure', titlesize=14)
plt.rc('axes', labelsize=14)
plt.rc('xtick', labelsize=8)
plt.rc('ytick', labelsize=8)
plt.show()
```

```
0.982913442195686
(0.99196053254782O6, 1.9690811124379223e-19)
0.9828164088578178
(0.9919605325478208, 1.9690811124373804e-19)
0.9804297204209497
(0.9919605325478205, 1.969081112438193e-19)
0.9828012385736059
(0.9919605325478207, 1.9690811124376514e-19)
0.9837101582456282
(0.9919605325478205, 1.969081112438193e-19)
0.9832826167153719
(0.9919605325478205, 1.969081112438193e-19)
0.9832324446704865
(0.9919605325478208, 1.9690811124373804e-19)
0.983784437794529
(0.9919605325478207, 1.9690811124376514e-19)
0.9822438293072531
(0.9919605325478208, 1.9690811124373804e-19)
0.9814403045147115
(0.9919605325478206, 1.9690811124379223e-19)
0.9767023858503663
(0.9919605325478209, 1.9690811124371093e-19)
0.9823783253003997
(0.9919605325478207, 1.9690811124376514e-19)
0.9833843802466801
(0.9919605325478207, 1.9690811124376514e-19)
0.9835344349460009
(0.9919605325478207, 1.9690811124376514e-19)
0.9831310267970349
(0.9919605325478207, 1.9690811124376514e-19)
0.9835014745884098
(0.9919605325478205, 1.969081112438193e-19)
```



Average Across All Fit Parameters

In [53]:
```
# Averaging slopes.
Param_M_Mean = []

Sum = 0
for channel in range(0,15):
    Sum += Param[channel][0]
Param_M_Mean = (1/16)*Sum
```

In [54]:
```
# Averaging intercepts.
Param_C_Mean = []

Sum = 0
for channel in range(0,15):
    Sum += Param[channel][1]
Param_C_Mean = (1/16)*Sum
```

In [55]:
```
# Averaging slope errors.
Param_MErr_Mean = []

Sum = 0
for channel in range(0,15):
    Sum += (Param_Err[channel][0])**2
Param_MErr_Mean = (1/16)*np.sqrt(Sum)
```

In [56]:
```
# Averaging intercept errors.
Param_CErr_Mean = []

Sum = 0
for channel in range(0,15):
    Sum += (Param_Err[channel][1])**2
Param_CErr_Mean = (1/16)*np.sqrt(Sum)
```

In [57]:
```
print(f"M = {Param_M_Mean.round(2)} +/- {Param_MErr_Mean.round(2)}")
print(f"C = {Param_C_Mean.round(2)} +/- {Param_CErr_Mean.round(2)}")
```

```
M = 28.22 +/- 0.08
C = 21.01 +/- 0.11
```

Average Fit Function

In [58]:
```
# Fit function formed using averaged M and C parameters.
# Converts voltage to temperature.
def mean_linear(xdata, xerr):
    YData = []
    Yerr2 = []

    for i in range(0,69):
        ydata = Param_M_Mean*xdata[i] + Param_C_Mean
        YData.append(ydata)

        yerr1 = Param_M_Mean*xdata[i]*(np.sqrt((Param_MErr_Mean/Param_M_Mean)**2
                                    + (xerr/xdata[i])**2))

        yerr2 = np.sqrt((yerr1)**2 + (Param_CErr_Mean)**2)
        Yerr2.append(yerr2)

    return YData, Yerr2
```

Test: Convert U.S. using Average Fit Function

In [59]:
```
# Create dictionary for channel voltages.
Channel_Voltages = {}

# Name dictionary keys according to channel number.
for i in range(1,17):
    key_i = 'Channel_{}'.format(i)
    Channel_Voltages.setdefault(key_i, [])
```

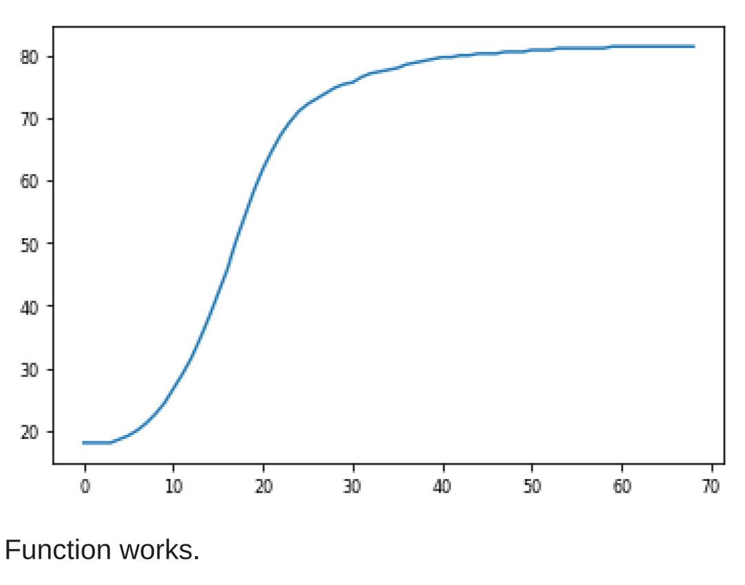
In [60]:
```
# Import data as lists into corresponding dictionary keys.
subfilename = "Session_3\Part2_Run1_Voltage.txt"
with open(subfilename, newline='') as subfile:
    n = 1
    for row in csv.reader(subfile):
        Channel_Voltages['Channel_{}'.format(n)].append(row[0].split(' '))
        n += 1
```

In [61]:
```
# Remove nested list and convert strings to floats.
for i in range(1,17):
    Channel_Voltages['Channel_{}'.format(i)] = Channel_Voltages['Channel_{}'.format(i)][0]
    for n in range(0,70):
        Channel_Voltages['Channel_{}'.format(i)][n] = float(Channel_Voltages['Channel_{}'.format(i)][n])
```

In [62]:
```
Temp, Temp_err = mean_linear(Channel_Voltages['Channel_1'], V_err)
```

In [63]:
```
plt.plot(Temp)
```

Out[63]:
```
[<matplotlib.lines.Line2D at 0x2501c7e7a90>]
```



Function works.

Newton's Cooling: U.S.

Import Modules

In [143…
```python
import numpy as np
import matplotlib.pyplot as plt
import csv
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pylab as pl
from scipy.optimize import curve_fit
from scipy.odr import ODR, Model, Data, RealData
from matplotlib import cm
import math
```

In [144…
```python
#%matplotlib qt
```

Import Voltages

In [145…
```python
# Create dictionary for channel voltages.
Channel_Voltages = {}

# Name dictionary keys according to channel number.
for i in range(1,17):
    key_i = 'Channel_{}'.format(i)
    Channel_Voltages.setdefault(key_i, [])
```

In [146…
```python
# Import data as lists into corresponding dictionary keys.
subfilename = "Session_3\Part2_Run1_Voltage.txt"
with open(subfilename, newline='') as subfile:
    n = 1
    for row in csv.reader(subfile):
        Channel_Voltages['Channel_{}'.format(n)].append(row[0].split(' '))
        n += 1
```

In [147…
```python
# Remove nested list and convert strings to floats.
for i in range(1,17):
    Channel_Voltages['Channel_{}'.format(i)] = Channel_Voltages['Channel_{}'.format(i)][0]
    for n in range(0,70):
        Channel_Voltages['Channel_{}'.format(i)][n] = float(Channel_Voltages['Channel_{}'.format(i)][n])
```

Convert Voltage to Temperature

In [148…
```python
# Create dictionary for channel temperatures.
Channel_Temps = {}

# Name dictionary keys according to channel number.
for i in range(1,17):
    key_i = 'Channel_{}'.format(i)
    Channel_Temps.setdefault(key_i, [])
```

Note: Ideally, the mean_linear function below would be imported. The function and its associated parameters have been inserted manually here due to import issues.

In [149…
```python
Param_M_Mean = 28.2207
Param_MErr_Mean = 0.0768
Param_C_Mean = 21.0103
Param_CErr_Mean = 0.1083
```

In [150…
```python
def mean_linear(xdata, xerr):
    YData = []
    Yerr2 = []

    for i in range(0,70):
        ydata = Param_M_Mean*xdata[i] + Param_C_Mean
        YData.append(ydata)

        yerr1 = Param_M_Mean*xdata[i]*(np.sqrt((Param_MErr_Mean/Param_M_Mean)**2
                                + (xerr/xdata[i])**2))

        yerr2 = np.sqrt((yerr1)**2 + (Param_CErr_Mean)**2)
        Yerr2.append(yerr2)

    return YData, Yerr2
```

In [151…
```python
V_err = 0.05
```

In [152…
```python
# Add converted temperature values to their corresponding dictionary keys.
for i in range(1,17):
    Temp, Temp_err = mean_linear(Channel_Voltages['Channel_{}'.format(i)], V_err)
    Channel_Temps['Channel_{}'.format(i)].append(Temp)
    Channel_Temps['Channel_{}'.format(i)] = Channel_Temps['Channel_{}'.format(i)][0]
    print(np.mean(Temp_err))
```

```
1.4215083030231301
1.4202726825079797
1.4194075835806346
1.4186824975973262
1.4160405304906991
1.4158283952149167
1.4157061048950048
1.415585296157048
1.415268539148341
1.415244368011355
1.415233645518048
1.415220282168793
1.4151974968681775
1.4151971919165975
1.4151945223332536
1.4151975502100003
```

In [153…
```python
Temp_err= 1.415 # Take the mean temperature error for convenience.
```

Position

Note: Would be much more convenient to create an array for positions, as in the Time section.

In [154…
```python
# Create dictionary for channel positions.
Channel_Positions = {}

# Name dictionary keys according to channel number.
for i in range(1,17):
    key_i = 'Channel_{}'.format(i)
    Channel_Positions.setdefault(key_i, [])
    Channel_Positions['Channel_{}'.format(i)] = 5*(i)
```

Time

In [155…
```python
Time = np.zeros(70)
for i in range(0,70):
    Time[i] = i
```

Data Analysis

In [156…
```python
# The mean first recorded temperature across all channels.
meanfirst = []
for i in range(1,16):
    meanfirst.append(Channel_Temps['Channel_{}'.format(i)][0])

# Determine the error on the mean first recorded temperature across all channels.
x = 0
for i in range(1,17):
    x += 1.41**2

print(f"The mean first temperature across all channels is: {np.mean(meanfirst).round(1)} +/- {(1/16)*np.sqrt(x).round(1)}")
```

```
The mean first temperature across all channels is: 17.7 +/- 0.35
```

In [157…
```python
# The maximum recorded temperature.
print(f"The maximum recorded temp is: {np.max(Channel_Temps['Channel_1']).round(1)} +/- {Temp_err}")
```

```
The maximum recorded temp is: 81.4 +/- 1.415
```

Determination of Beta

In [158…
```python
# The final recorded temperature for each channel.
finaltemp = []
for i in range(1,17):
    finaltemp.append(Channel_Temps['Channel_{}'.format(i)][-1])
```

In [159…
```python
# The final recorded position for each channel.
finalposition = []
for i in range(1,17):
    finalposition.append(Channel_Positions['Channel_{}'.format(i)])
```

In [160…
```python
def exp(beta, x):
    A,B,C = beta
    return A*np.exp(-B*x) + C
```

In [161…
```python
def fitting_func(xdata, ydata, xerr, yerr, A_guess, B_guess, C_guess):

    # Create a model.
    DV_model = Model(exp)

    # Create a RealData object.
    data = RealData(xdata, ydata, xerr, yerr)

    # Create ODR with model and data.
    beta0=[A_guess, B_guess, C_guess] # Guess beta values.
    odr = ODR(data, DV_model, beta0)
    odr.set_job(fit_type=0)

    # Run the regression.
    out = odr.run()

    # Create array of numbers with same interval as x.
    x_fit = np.linspace(np.min(xdata), np.max(xdata), 100)

    # Run the fit function using x_fit and regression values of beta.
    y_fit = exp(out.beta, x_fit)

    return x_fit, y_fit, out.beta, out.sd_beta
```

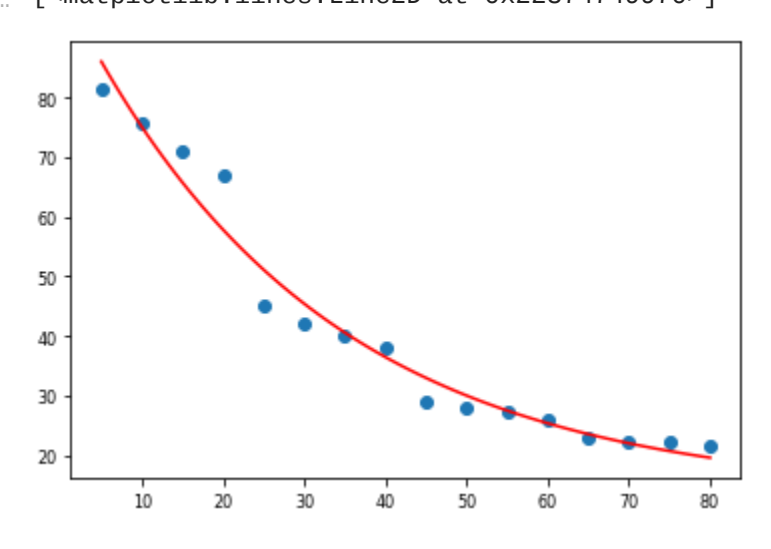
In [162…
```python
XFit, YFit, params, params_errs = fitting_func(finalposition,finaltemp,V_err,Temp_err,8.56358571e+01, 3.27337849e-02, 1.33319654e+01)
```

In [163…
```python
A = params[0]
A_err = params_errs[0]
B = params[1]*100
B_err = params_errs[1]*100
C = params[2]
C_err = params_errs[2]
T0 = A + C #Also nown as T_l, the heater temp.
T0_err = np.sqrt(A_err**2 + C_err*2)

print(f"Param A = {A.round(1)} +/- {A_err.round(1)}")
print(f"Param B = {(B).round(2)} +/- {(B_err).round(2)} (1/m)")
print(f"Param C or T_R = {C.round(1)} +/- {C_err.round(1)}")
print(f"T0 or T_L = {T0.round(1)} +/- {T0_err.round(1)}")
```

```
Param A = 85.6 +/- 4.3
Param B = 3.27 +/- 0.57 (1/m)
Param C or T_R = 13.3 +/- 4.7
T0 or T_L = 99.0 +/- 5.3
```

In [164…
```python
# Check visual fit to plot of final position against final temperature plane.
plt.scatter(finalposition, finaltemp)
plt.plot(XFit, YFit, color='red')
```

Out[164…
```
[<matplotlib.lines.Line2D at 0x22374749970>]
```



In [165…
```python
np.savetxt('Beta1_Run1.txt', np.vstack((XFit,YFit)).T, delimiter=' ')
np.savetxt('Beta2_Run1.txt', np.vstack((finaltemp,finalposition)).T, delimiter=' ')
```

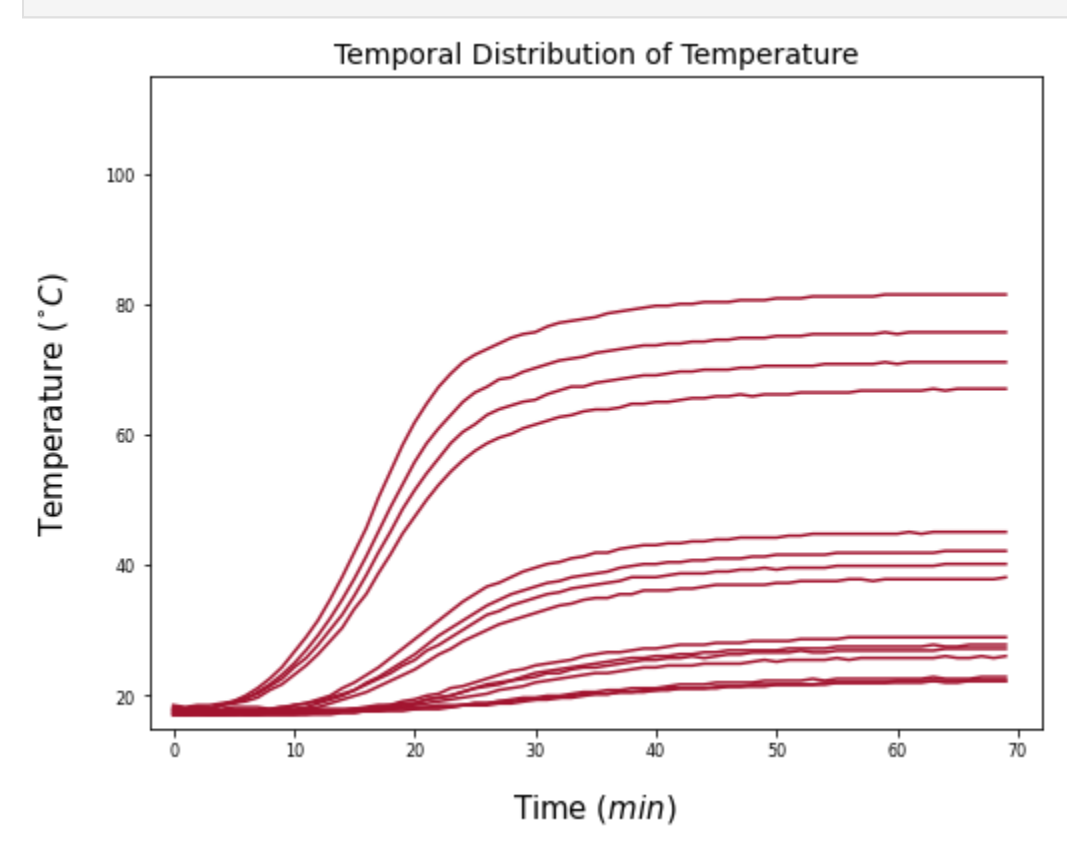
In [166…
```python
lam = 237 # W/mK
lam_err = 60
r = 0.14 # m
r_err = 0.01
H = (lam*r*B**2)/2
H_err = H * np.sqrt((B_err/B)**2 + (r_err/r)**2 + (lam_err/lam)**2)
print(f"H = {H.round(1)} +/- {H_err.round(2)} W/Km^2")
```

```
H = 177.8 +/- 56.2 W/Km^2
```

2D Plot

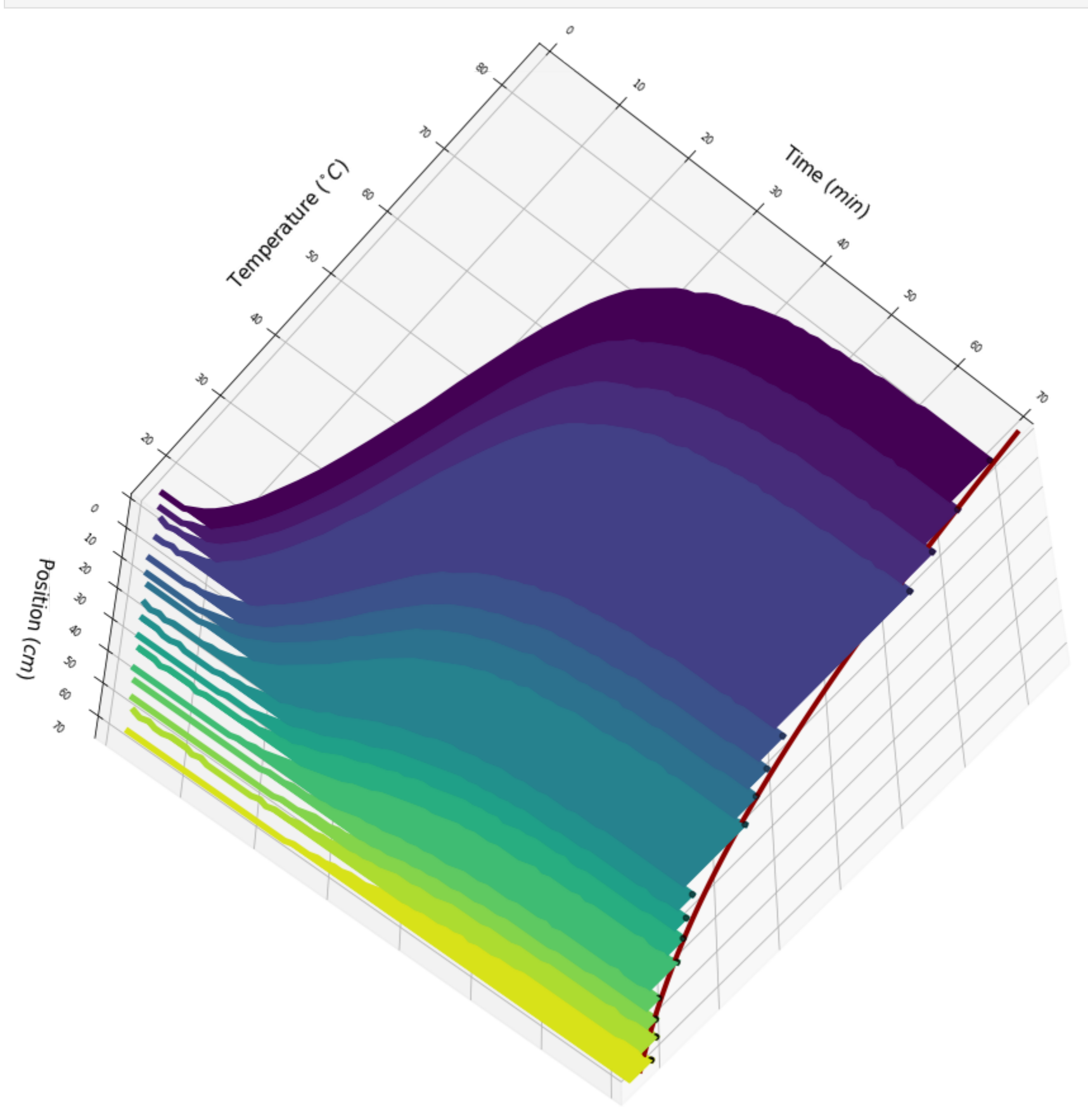
In [167…
```python
red = '#A2142F'
blue = '#0072BD'
green = 'green'
orange = 'darkorange'
```

In [168…
```python
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111)
colors = pl.cm.spring(np.linspace(0,1,16))

for i in range(1,16):
    X = Time
    Y = Channel_Temps['Channel_{}'.format(i)]

    Z   = Channel_Positions

    #ax.fill_between(X, np.min(Y), Y, interpolate=False, color=colors[i])

    ax.plot(X, Y, color=red)

# Labels
ax.set_xlabel('Time $(min)$', labelpad=15, fontsize=15)
ax.set_ylabel('Temperature $(^{\circ}C)$',labelpad=15, fontsize=15)
ax.set_title('Temporal Distribution of Temperature', fontsize = 14)
#plt.legend(loc='lower center', bbox_to_anchor=(0.5,0.91),shadow=False,ncol=5,markerscale=1.0,handlelength=1.0)

# Limits
ax.set_xlim([-2, 72])
ax.set_ylim([15, 115])

plt.rc('figure', titlesize=14)
plt.rc('axes', labelsize=14)
plt.rc('xtick', labelsize=8)
plt.rc('ytick', labelsize=8)

plt.show()
```



3D Plot

In [169…
```python
fig = plt.figure(figsize=(16,15))
ax = fig.add_subplot(111, projection='3d')
colors = pl.cm.viridis(np.linspace(0,1,17))
ax.view_init(119,-50)

# Plot each channel.
for i in range(1,17):
    X = Time
    Y = Channel_Temps['Channel_{}'.format(i)]
    Z = Channel_Positions['Channel_{}'.format(i)]

    ax.add_collection3d(plt.fill_between(X,np.min(Y),Y,color=colors[i-1]), zs=Z, zdir='Z') # lower boundary z=-30
    ax.plot(X, Y, Z, color=colors[i-1], linewidth=5)

# Plot final position and temperature plane.
ZFit = 100*[69]
ZFit2 = 16*[69]
ax.plot(ZFit, YFit, XFit, color='darkred', linewidth=4)
ax.scatter(ZFit2, finaltemp, finalposition, marker='o',color='black')

# Labels
ax.set_xlabel('Time $(min)$', labelpad=15, fontsize=15)
ax.set_ylabel('Temperature $(^{\circ}C)$',labelpad=15, fontsize=15)
ax.zaxis.set_rotate_label(False)  # disable automatic rotation
ax.set_zlabel('Position $(cm)$',labelpad=15, fontsize=15, rotation=-100)

# Limits
ax.set_xlim([0, 70])
ax.set_ylim([15, 85])
ax.set_zlim([0, 75])

# Ticks
plt.xticks(rotation=-40)
plt.yticks(rotation=-40)
ax.zaxis.set_tick_params(rotation=-40)
ax.tick_params(axis='both', which='major', pad=10)
plt.show()
```



Entropy Transfer

$$\frac{d_eS}{dt} \propto -\frac{1}{l}\frac{1}{T_0}(T_1-T_0)+\frac{1}{l}\frac{1}{T_n}(T_n-T_{n-1})-\beta^2 l\sum_{i=1}^{n-1}\left(1-\frac{T_n}{T_i}\right) \qquad (1)$$

```
\begin{equation}
\mbox{LHS} = - \frac{1}{l} \frac{1}{T_0} (T_1-T_0) \newline
\mbox{MID} = \frac{1}{l} \frac{1}{T_n}(T_n-T_{n-1}) \newline
\mbox{RHS} = - \beta^2 l \sum_{i=1}^{n-1} \left(1 - \frac{T_n}{T_i} \right)
\end{equation}
```

Notes: The maximum n is 17. The summation section will be determined first. T_n is the ambiance temp, known as T_{17}. For the purpose of simplicity, the errors on T_0 and T_{17} are taken to be the same as all other temperatures.

In [170…
```python
T0 = A # Furnace temperature.
T1 = Channel_Temps['Channel_1'] # Channel 1 temperatures.
T16 = Channel_Temps['Channel_16'] # Channel 16 temperatures.
T17 = C # Ambiance temperature.
l = 0.05 # Length between volume elements. [m].
```

In [171…
```python
EntropyTrans = []
EntropyTrans_err = []

for i in range(0,70):
    summation = 0
    summation_err = 0

    for channel in range(1,17):
        summation += 1 - T17/Channel_Temps['Channel_{}'.format(channel)][i]

        total_err = (T17/Channel_Temps['Channel_{}'.format(channel)][i])*np.sqrt((Temp_err/T17)**2 + (Temp_err/Channel_Temps['Channel_{}'.format(channel)][i])**2)
        summation_err += total_err**2 # Square root is taken later, for convenience.

    entire_equation = (-1/(l*T0))*(T1[i]-T0) + (1/(l*T17))*(T17-T16[i])-(B**2)*l*summation
    EntropyTrans.append(entire_equation)

    # Determination of all errors. (LHS, MID, RHS)
    LHS_1_err = Temp_err/T0**2
    LHS_2_err = np.sqrt(2*(Temp_err)**2)
    LHS_err = np.abs((-1/(l*T0))*(T1[i]-T0) * np.sqrt((LHS_1_err/T0)**2 + (LHS_2_err/(T1[i]-T0))**2))

    MID_err = LHS_err

    summation_err = np.sqrt(summation_err) # Completing the additon of uncertainties.
    RHS_err = np.abs((B**2)*l*summation * np.sqrt(2*(B_err/B)**2 + (summation_err/summation)**2))

    entire_err = np.sqrt(LHS_err**2 + MID_err**2 + RHS_err**2)

    if np.isnan(entire_err) == True:
            entire_err = 0

    EntropyTrans_err.append(entire_err)
```

Entropy Production

$$\frac{d_iS}{dt} \propto -\frac{1}{l}\sum_{i=0}^{n-1}(T_{i+1}-T_i)\left(\frac{1}{T_{i+1}}-\frac{1}{T_i}\right) \qquad (2)$$

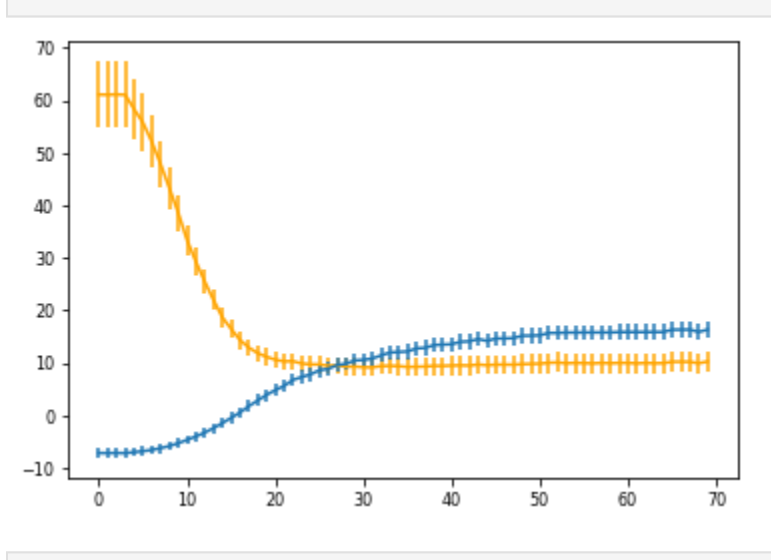
```
\begin{equation}
\mbox{LHS} = (T_{i+1}-T_i) \newline
\mbox{RHS} = \left( \frac{1}{T_{i+1}} - \frac{1}{T_i} \right)
\end{equation}
```

Note: The maximum n is 17.

In [172…
```python
EntropyProd = []
EntropyProd_err = []

for i in range(0,70):
    summation = 0
    summation_err = 0

    for channel in range(0,17):
        if channel == 0: # If i == 0:
            LHS = T1[i] - T0
            RHS = 1/T1[i] - 1/T0

            LHS_err = np.sqrt(2*(Temp_err)**2)
            RHS_1_err = Temp_err/(T1[i]**2)
            RHS_2_err = Temp_err/(T0**2)
            RHS_err = np.sqrt(RHS_1_err**2 + RHS_2_err**2)

        elif channel > 0 and channel < 16: # If i is equal to/between 1 and 15:
            LHS = Channel_Temps['Channel_{}'.format(channel+1)][i] - Channel_Temps['Channel_{}'.format(channel)][i]
            RHS = 1/Channel_Temps['Channel_{}'.format(channel+1)][i] - 1/Channel_Temps['Channel_{}'.format(channel)][i]

            LHS_err = np.sqrt(2*(Temp_err)**2)
            RHS_1_err = Temp_err/(Channel_Temps['Channel_{}'.format(channel+1)][i]**2)
            RHS_2_err = Temp_err/(Channel_Temps['Channel_{}'.format(channel)][i]**2)
            RHS_err = np.sqrt(RHS_1_err**2 + RHS_2_err**2)

        elif channel == 16: # If i == 16:
            LHS = T17 - T16[i]
            RHS = 1/T17 - 1/T16[i]

            LHS_err = np.sqrt(2*(Temp_err)**2)
            RHS_1_err = Temp_err/(T17**2)
            RHS_2_err = Temp_err/(T16[i]**2)
            RHS_err = np.sqrt(RHS_1_err**2 + RHS_2_err**2)

        summation += LHS * RHS

        total_err = (LHS*RHS)*(np.sqrt((LHS_err/LHS)**2 + (RHS_err/RHS)**2))
        if np.isnan(total_err) == True:
            total_err = 0
        summation_err += total_err**2

    EntropyProd.append(summation * -1/l)
    EntropyProd_err.append(np.sqrt(summation_err) * 1/l)
```

Entropy Transfer & Production

In [173…
```python
# Check the scaling of both plots together.
plt.errorbar(Time, EntropyProd, xerr=0, yerr=EntropyProd_err, color='orange')
plt.errorbar(Time, [-1*elem for elem in EntropyTrans], EntropyTrans_err);
```



In [174…
```python
np.savetxt('EntropyProd1.txt', np.vstack((Time,EntropyProd,EntropyProd_err)).T, delimiter=' ')
np.savetxt('EntropyTrans1.txt', np.vstack((Time,[-1*elem for elem in EntropyTrans],EntropyTrans_err)).T, delimiter=' ')
```

Newton's Cooling: Combined Entropy Plots

Imports

In [244…

```
import numpy as np
import matplotlib.pyplot as plt
import csv
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pylab as pl
from scipy.optimize import curve_fit
from scipy.odr import ODR, Model, Data, RealData
from matplotlib import cm
```

Import Data

In [245…

```
lst = []
file = open('EntropyProd1.txt')
for line in file:
    lst.append([ float(x) for x in line.split()])
    Time = [x[0] for x in lst]
    EntropyProd1 = [x[1] for x in lst]
    EntropyProd1_err = [x[2] for x in lst]

lst = []
file = open('EntropyTrans1.txt')
for line in file:
    lst.append([ float(x) for x in line.split()])
    EntropyTrans1 = [x[1] for x in lst]
    EntropyTrans1_err = [x[2] for x in lst]
```

In [246…

```
lst = []
file = open('EntropyProd2.txt')
for line in file:
    lst.append([ float(x) for x in line.split()])
    EntropyProd2 = [x[1] for x in lst]
    EntropyProd2_err = [x[2] for x in lst]

lst = []
file = open('EntropyTrans2.txt')
for line in file:
    lst.append([ float(x) for x in line.split()])
    EntropyTrans2 = [x[1] for x in lst]
    EntropyTrans2_err = [x[2] for x in lst]
```

In [247…

```
lst = []
file = open('EntropyProd3.txt')
for line in file:
    lst.append([ float(x) for x in line.split()])
    Time = [x[0] for x in lst]
    EntropyProd3 = [x[1] for x in lst]
    EntropyProd3_err = [x[2] for x in lst]

lst = []
file = open('EntropyTrans3.txt')
for line in file:
    lst.append([ float(x) for x in line.split()])
    EntropyTrans3 = [x[1] for x in lst]
    EntropyTrans3_err = [x[2] for x in lst]
```

In [248…

```
lst = []
file = open('EntropyProd4.txt')
for line in file:
    lst.append([ float(x) for x in line.split()])
    Time = [x[0] for x in lst]
    EntropyProd4 = [x[1] for x in lst]
    EntropyProd4_err = [x[2] for x in lst]

lst = []
file = open('EntropyTrans4.txt')
for line in file:
    lst.append([ float(x) for x in line.split()])
    EntropyTrans4 = [x[1] for x in lst]
    EntropyTrans4_err = [x[2] for x in lst]
```

Combined Plots

In [249…

```
red = '#A2142F'
blue = '#0072BD'
green = 'green'
orange = 'darkorange'
```

Fitting Functions

In [250…

```
def exp(beta, x):
    A,B,C,D = beta
    return A/(1+np.exp(B*(x+C))) + D
```

In [251…

```
def fitting_func(xdata, ydata, xerr, yerr, A_guess, B_guess, C_guess, D_guess):

    # Create a model.
    DV_model = Model(exp)

    # Create a RealData object.
    data = RealData(xdata, ydata)

    # Create ODR with model and data.
    beta0=[A_guess, B_guess, C_guess, D_guess] # Guess beta values.
    odr = ODR(data, DV_model, beta0)
    odr.set_job(fit_type=0)

    # Run the regression.
    out = odr.run()

    # Create array of numbers with same interval as x.
    x_fit = np.linspace(np.min(xdata), np.max(xdata), 100)

    # Run the fit function using x_fit and regression values of beta.
    y_fit = exp(out.beta, x_fit)

    return x_fit, y_fit, out.beta, out.sd_beta
```

Entropy Production

In [252…

```
XFit1, YFit1, params1, params1_errs = fitting_func(Time[:], EntropyProd1[:], 1e-16, EntropyProd1_err[:],
                                                   -53.59803448,  -0.38369072,  -9.67501095,  63.28904186)
```

In [253…

```
XFit2, YFit2, params2, params2_errs = fitting_func(Time[:], EntropyProd2[:], 1e-16, EntropyProd2_err[:],
                                                   -43.15724149,  -0.30286055, -11.94789542,  52.41076877)
```

In [254…

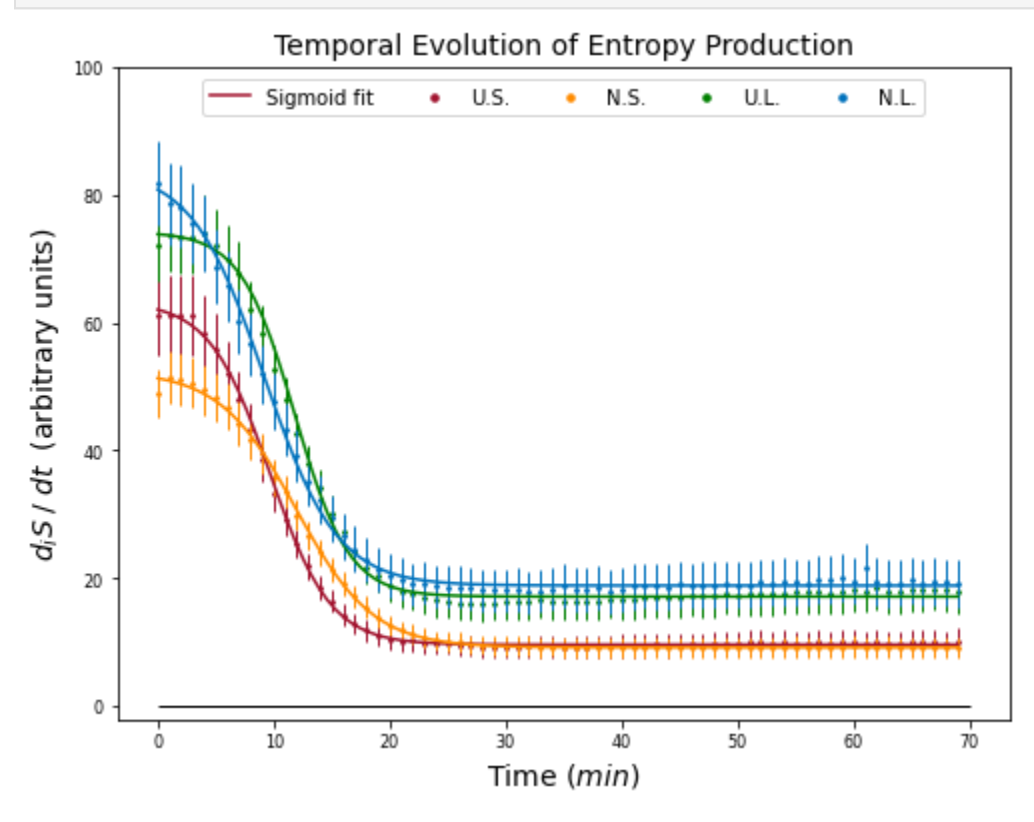
```
XFit3, YFit3, params3, params3_errs = fitting_func(Time[:], EntropyProd3[:], 1e-16, EntropyProd3_err[:],
                                                   -56.94838847,  -0.42979395, -11.82006234,  74.13910736)
```

In [255…

```
XFit4, YFit4, params4, params4_errs = fitting_func(Time[:], EntropyProd4[:], 1e-16, EntropyProd4_err[:],
                                                   -65.13150137,  -0.31878942,  -9.19736225,  84.04994283)
```

In [256…

```
fig = plt.figure(figsize=(8,6))

plt.errorbar(Time, EntropyProd1, yerr=EntropyProd1_err, color=red, ls='none', elinewidth=1)
plt.errorbar(Time, EntropyProd2, yerr=EntropyProd2_err, color=orange, ls='none', elinewidth=1)
plt.errorbar(Time, EntropyProd3, yerr=EntropyProd3_err, color=green, ls='none', elinewidth=1)
plt.errorbar(Time, EntropyProd4, yerr=EntropyProd4_err, color=blue, ls='none', elinewidth=1)

plt.scatter(Time, EntropyProd1, color=red, marker='o', s=3, label='U.S.')
plt.scatter(Time, EntropyProd2, color=orange, marker='o', s=3, label='N.S.')
plt.scatter(Time, EntropyProd3, color=green, marker='o', s=3, label='U.L.')
plt.scatter(Time, EntropyProd4, color=blue, marker='o', s=3, label='N.L.')

plt.plot(XFit1, YFit1, color=red, label='Sigmoid fit')
plt.plot(XFit2, YFit2, color=orange)
plt.plot(XFit3, YFit3, color=green)
plt.plot(XFit4, YFit4, color=blue)

plt.hlines(y=0, xmin=0, xmax=70, linewidth=1, color='black')

plt.ylabel('$d_iS\;/\;dt\;$ (arbitrary units)')
plt.xlabel(r'Time $(min)$')
plt.title('Temporal Evolution of Entropy Production', fontsize = 14)
plt.legend(loc='lower center', bbox_to_anchor=(0.5,0.91), ncol=5, markerscale=2.0)

plt.rc('figure', titlesize=14)
plt.rc('axes', labelsize=14)
plt.rc('xtick', labelsize=8)
plt.rc('ytick', labelsize=8)
plt.ylim([-2, 100])
plt.show()
```



In [257…

```
print(f"{np.round(params1[0],1)} +/- {np.round(params1_errs[0],1)}")
print(f"{np.round(params1[1]*1e2,1)} +/- {np.round(params1_errs[1]*1e2,1)}")
print(f"{np.round(params1[2],1)} +/- {np.round(params1_errs[2],1)}")
print(f"{np.round(params1[3],1)} +/- {np.round(params1_errs[3],1)}")
```

```
-53.6 +/- 0.4
-38.4 +/- 1.1
-9.7 +/- 0.1
63.3 +/- 0.4
```

In [258…

```
print(f"{np.round(params2[0],1)} +/- {np.round(params2_errs[0],1)}")
print(f"{np.round(params2[1]*1e2,1)} +/- {np.round(params2_errs[1]*1e2,1)}")
print(f"{np.round(params2[2],1)} +/- {np.round(params2_errs[2],1)}")
print(f"{np.round(params2[3],1)} +/- {np.round(params2_errs[3],1)}")
```

```
-43.2 +/- 0.3
-30.3 +/- 0.7
-11.9 +/- 0.1
52.4 +/- 0.3
```

In [259…

```
print(f"{np.round(params3[0],1)} +/- {np.round(params3_errs[0],1)}")
print(f"{np.round(params3[1]*1e2,1)} +/- {np.round(params3_errs[1]*1e2,1)}")
print(f"{np.round(params3[2],1)} +/- {np.round(params3_errs[2],1)}")
print(f"{np.round(params3[3],1)} +/- {np.round(params3_errs[3],1)}")
```

```
-56.9 +/- 0.5
-43.0 +/- 2.5
-11.8 +/- 0.2
74.1 +/- 0.5
```

In [260…

```
print(f"{np.round(params4[0],1)} +/- {np.round(params4_errs[0],1)}")
print(f"{np.round(params4[1]*1e2,1)} +/- {np.round(params4_errs[1]*1e2,1)}")
print(f"{np.round(params4[2],1)} +/- {np.round(params4_errs[2],1)}")
print(f"{np.round(params4[3],1)} +/- {np.round(params4_errs[3],1)}")
```

```
-65.1 +/- 1.2
-31.9 +/- 1.4
-9.2 +/- 0.3
84.1 +/- 1.1
```

Entropy Transfer

In [261…

```
XFit1, YFit1, params1, params1_errs = fitting_func(Time[:], EntropyTrans1[:], 1e-16, EntropyTrans1_err[:],
                                                   24.89272486,  -0.12307022, -18.38980973, -11.81250984)
```

In [262…

```
params4
```

Out[262…

```
array([-65.13175502,  -0.31878573,  -9.19731877,  84.05018822])
```

In [263…

```
XFit2, YFit2, params2, params2_errs = fitting_func(Time[10:], EntropyTrans2[10:], 1e-16, EntropyTrans2_err[10:],
                                                    8.1286214 ,  -0.22362235, -26.53169049,   3.62552028)
```

In [264…

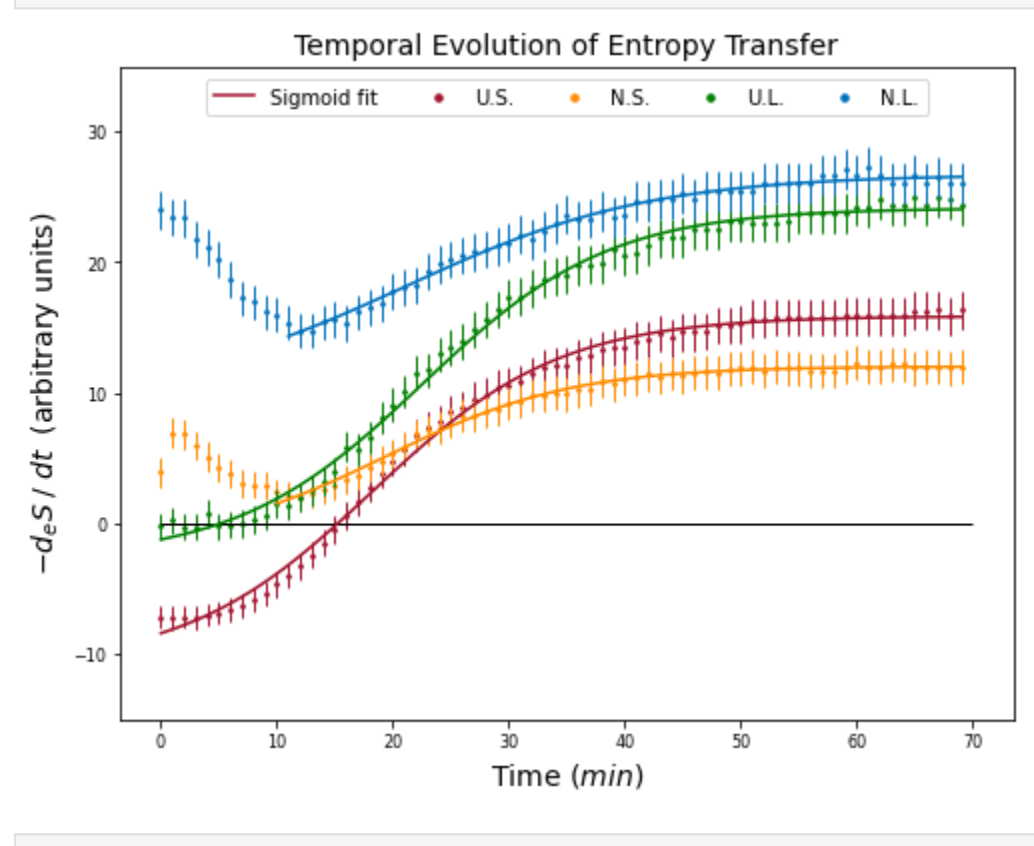
```
XFit3, YFit3, params3, params3_errs = fitting_func(Time[:], EntropyTrans3[:], 1e-16, EntropyTrans3_err[:],
                                                   26.88721579,  -0.1244019 , -22.56517259,  -2.74642184)
```

In [265…

```
XFit4, YFit4, params4, params4_errs = fitting_func(Time[11:], EntropyTrans4[11:], 1e-16, EntropyTrans4_err[11:],
                                                   -8.09983621,   0.2085425 , -32.82386127,  26.13316131)
```

In [266…

```
fig = plt.figure(figsize=(8,6))

plt.errorbar(Time, EntropyTrans1, yerr=EntropyTrans1_err, color=red, ls='none', elinewidth=1)
plt.errorbar(Time, EntropyTrans2, yerr=EntropyTrans2_err, color=orange, ls='none', elinewidth=1)
plt.errorbar(Time, EntropyTrans3, yerr=EntropyTrans3_err, color=green, ls='none', elinewidth=1)
plt.errorbar(Time, EntropyTrans4, yerr=EntropyTrans4_err, color=blue, ls='none', elinewidth=1)

plt.scatter(Time, EntropyTrans1, color=red, marker='o', s=3, label='U.S.')
plt.scatter(Time, EntropyTrans2, color=orange, marker='o', s=3, label='N.S.')
plt.scatter(Time, EntropyTrans3, color=green, marker='o', s=3, label='U.L.')
plt.scatter(Time, EntropyTrans4, color=blue, marker='o', s=3, label='N.L.')

plt.plot(XFit1, YFit1, color=red, label='Sigmoid fit')
plt.plot(XFit2, YFit2, color=orange)
plt.plot(XFit3, YFit3, color=green)
plt.plot(XFit4, YFit4, color=blue)

plt.hlines(y=0, xmin=0, xmax=70, linewidth=1, color='black')

plt.ylabel('$-d_eS\;/\;dt\;$ (arbitrary units)')
plt.xlabel(r'Time $(min)$')
plt.title('Temporal Evolution of Entropy Transfer', fontsize = 14)
plt.legend(loc='lower center', bbox_to_anchor=(0.5,0.91), ncol=5, markerscale=2.0)

plt.rc('figure', titlesize=14)
plt.rc('axes', labelsize=14)
plt.rc('xtick', labelsize=8)
plt.rc('ytick', labelsize=8)
plt.ylim([-15, 35])
plt.show()
```



In [267…

```
print(f"{np.round(params1[0],1)} +/- {np.round(params1_errs[0],1)}")
print(f"{np.round(params1[1]*1e2,1)} +/- {np.round(params1_errs[1]*1e2,1)}")
print(f"{np.round(params1[2],1)} +/- {np.round(params1_errs[2],1)}")
print(f"{np.round(params1[3],1)} +/- {np.round(params1_errs[3],1)}")
```

```
26.8 +/- 0.6
-12.4 +/- 0.5
-18.3 +/- 0.5
-10.9 +/- 0.6
```

In [268…

```
print(f"{np.round(params2[0],1)} +/- {np.round(params2_errs[0],1)}")
print(f"{np.round(params2[1]*1e2,1)} +/- {np.round(params2_errs[1]*1e2,1)}")
print(f"{np.round(params2[2],1)} +/- {np.round(params2_errs[2],1)}")
print(f"{np.round(params2[3],1)} +/- {np.round(params2_errs[3],1)}")
```

```
13.7 +/- 0.8
-12.6 +/- 0.7
-19.6 +/- 1.0
-1.6 +/- 0.8
```

In [269…

```
print(f"{np.round(params3[0],1)} +/- {np.round(params3_errs[0],1)}")
print(f"{np.round(params3[1]*1e2,1)} +/- {np.round(params3_errs[1]*1e2,1)}")
print(f"{np.round(params3[2],1)} +/- {np.round(params3_errs[2],1)}")
print(f"{np.round(params3[3],1)} +/- {np.round(params3_errs[3],1)}")
```

```
26.9 +/- 0.6
-12.4 +/- 0.5
-22.6 +/- 0.4
-2.7 +/- 0.5
```

In [270…

```
print(f"{np.round(params4[0],1)} +/- {np.round(params4_errs[0],1)}")
print(f"{np.round(params4[1]*1e2,1)} +/- {np.round(params4_errs[1]*1e2,1)}")
print(f"{np.round(params4[2],1)} +/- {np.round(params4_errs[2],1)}")
print(f"{np.round(params4[3],1)} +/- {np.round(params4_errs[3],1)}")
```

```
-17.1 +/- 1.6
9.5 +/- 0.8
-21.2 +/- 1.9
26.7 +/- 0.2
```

Total Entropy

In [271…

```
Entropy1 = []
Entropy2 = []
Entropy3 = []
Entropy4 = []

for i in range(0,70):
    Entropy1.append(EntropyTrans1[i]+EntropyProd1[i])
    Entropy2.append(EntropyTrans2[i]+EntropyProd2[i])
    Entropy3.append(EntropyTrans3[i]+EntropyProd3[i])
    Entropy4.append(EntropyTrans4[i]+EntropyProd4[i])
```

In [272…

```
Entropy1_err = []
Entropy2_err = []
Entropy3_err = []
Entropy4_err = []

for i in range(0,70):
    Entropy1_err.append(np.sqrt(EntropyTrans1_err[i]**2+EntropyProd1_err[i]**2))
    Entropy2_err.append(np.sqrt(EntropyTrans2_err[i]**2+EntropyProd2_err[i]**2))
    Entropy3_err.append(np.sqrt(EntropyTrans3_err[i]**2+EntropyProd3_err[i]**2))
    Entropy4_err.append(np.sqrt(EntropyTrans4_err[i]**2+EntropyProd4_err[i]**2))
```

In [273…

```
fig = plt.figure(figsize=(8,6))

plt.errorbar(Time, Entropy1, yerr=Entropy1_err, color=red, ls='none', elinewidth=1)
plt.errorbar(Time, Entropy2, yerr=Entropy2_err, color=orange, ls='none', elinewidth=1)
plt.errorbar(Time, Entropy3, yerr=Entropy3_err, color=green, ls='none', elinewidth=1)
plt.errorbar(Time, Entropy4, yerr=Entropy4_err, color=blue, ls='none', elinewidth=1)

plt.scatter(Time, Entropy1, color=red, marker='o', s=3, label='U.S.')
plt.scatter(Time, Entropy2, color=orange, marker='o', s=3, label='N.S.')
plt.scatter(Time, Entropy3, color=green, marker='o', s=3, label='U.L.')
plt.scatter(Time, Entropy4, color=blue, marker='o', s=3, label='N.L.')

plt.hlines(y=0, xmin=0, xmax=70, linewidth=1, color='black')

plt.ylabel('$dS\;/\;dt\;$ (arbitrary units)')
plt.xlabel(r'Time $(min)$')
plt.title('Temporal Evolution of Total Entropy', fontsize = 14)
plt.legend(loc='lower center', bbox_to_anchor=(0.5,0.91), ncol=5, markerscale=2.0)

plt.rc('figure', titlesize=14)
plt.rc('axes', labelsize=14)
plt.rc('xtick', labelsize=8)
plt.rc('ytick', labelsize=8)
plt.ylim([-5, 130])
plt.show()
```