PHYC40600 - Physics with Astronomy and Space Science Lab 2;

Demonstrating non-equilibrium steady state conduction along a metallic rod with Newtonian losses

Daragh Hollman
daragh.hollman@ucdconnect.ie
(Dated: February 15, 2024)

The aims of this report were to demonstrate and investigate the steady-state solution of a metallic rod subject to a fixed temperature difference, a common example of a non-equilibrium steady-state system. It was demonstrated that the system evolved to the steady-state for uniform and non-uniform initial temperature distributions, showing that the steady-state depends solely on the boundary conditions. However, we do see that initial conditions do have the effect of changing the speed at which the system reaches the steady-state. Steady-states were demonstrated with furnace temperatures $T_H = (158 \pm 1.5)\,^\circ\mathrm{C}$, and $(147 \pm 3)\,^\circ\mathrm{C}$, resulting in exponential decay constants $\beta = (5.17 \pm 0.06)\,\mathrm{m}^{-1}$, and $\beta = (5.8 \pm 0.1)\,\mathrm{m}^{-1}$, corresponding well to literature results. We also undergo an investigation of the entropy production and flow in the system as it evolves and observe that the entropy production and flow decrease to a minimum as the system evolves to steady-state.

I. INTRODUCTION AND THEORY

In this report, heat conduction along a cylindrical rod will be investigated while subject to a constant temperature difference on each end of the rod, and with the assumption that the rod is not insulated and is exchanging heat over throughout with a so called "thermal bath". The hot side of the rod is kept at a constant temperature T_H, while the colder side of the rod and the thermal bath are open to the air for the ambient temperature T_C (under the assumption that this is relatively constant).

The thermal bath in this instance is simple just the air around the rod. This work follows closely procedure laid out by Ràfols and Ortín (1992) [1]. It is important to note that this example of a one-dimensional rod is a common example for the demonstration of the evolution of a thermodynamic system towards a non-equilibrium steady state, however unlike this demonstration, this rod is generally assumed to be perfectly insulated - something not possible to achieve in reality [1].

A. Local Energy Balance

The conservation of internal energy u in a solid is defined as:

$$\rho \frac{du}{dt} = -\nabla \cdot \vec{J}_q \tag{1}$$

where ρ is the density of the rod, and and $\vec{J}_q$ the heat flux [1, 2]. In solids, the internal energy is solely dependent on temperature [2], giving us a relation for the evolution of the temperature of the rod:

$$\rho c \frac{\partial T}{\partial t} = -\nabla \cdot \vec{J}_q \tag{2}$$

where c is the specific heat capacity, [1, 2]. The heat in the rod will flow according to Fourier's law [1, 2]:

$$\vec{J}_q = -\lambda \nabla T \tag{3}$$

where λ is the thermal conductivity of the rod, finally yielding:

$$\rho c \frac{\partial T}{\partial t} = \nabla \cdot (\lambda \nabla T) \tag{4}$$

This is a differential equation, which governs how the temperature distribution in the rod evolves. This can be solved with some initial and boundary conditions [1]. Newton's Law of cooling describes the boundary conditions at the surface of the rod [2]:

$$\lambda \hat{n} \cdot (\nabla T) + H(T - T_C) \tag{5}$$

where $\hat{n}$ is the unit normal to the surface (i.e. in the radial direction) and H is the conductance of the surface (which depends both on the thermal and geometric properties of the two media). Newton's law of cooling describes how the normal heat flow through the surface is proportional to the difference in temperature between the surface and the external thermal bath [1].

Two assumptions are made when we invoke Newton's law of cooling. The first of these is that the rod is assumed to have uniform temperature throughout a given cross section, ignoring any radial temperature gradient [1]. The second assumption, as introduced previously, is that the air surrounding the surface of the rod is assumed to be a thermal bath at constant temperature T_C. This second assumption is reasonable for two reasons: the heat capacity of the air is very large, and convection around the rod will circulate new air to replace the heated air [1].

With these assumptions equations 4 and 5, the boundary condition, into one-dimensional form:

$$\rho c \frac{\partial T(z,t)}{\partial t} = \lambda \frac{\partial^2 T(z,t)}{\partial z^2} - \frac{2H}{r}\left[T(z,t) - T_C\right] \tag{6}$$

where r is the radius of the rod [1, 2].

B. Steady-state Solution

As each side of the rod are kept a constant and different temperatures, the system cannot reach thermodynamic equilibrium. However, it can relax to a final steady-state which no longer evolves further. This is characterised by a nonzero transport of energy through the rod, whilst having a non-uniform temperature distribution [1]. The steady-state solution is found by requiring that the above equation doesn't change in time [1]. Hence:

$$\frac{\partial T(z,t)}{\partial t} = 0$$

therefore,

$$0 = \lambda \frac{\partial^2 T(z,t)}{\partial z^2} - \frac{2H}{r}\left[T(z,t) - T_C\right] \tag{7}$$

which is a linear second order differential equation with general solution:

$$\theta(z,t) = T(z,t) - T_C \tag{8}$$

$$\theta = Ae^{-\beta z} \tag{9}$$

Then substituting this solution into the equation and through the identification of terms, we find the steady-state temperature distribution, an exponential decay [1].

$$\left[T(z,\infty) - T_C\right] = \left[T_H - T_C\right] e^{-\beta z} \tag{10}$$

with,

$$\beta = \left(\frac{2H}{\lambda r}\right)^{\frac{1}{2}}$$

C. Entropy Production and Flow

The entropy of the system must come into question due to the lack of thermal equilibrium. In classical thermodynamics, the definition of entropy is only defined for states in thermodynamic equilibrium [1]. To get around this issue, the assumed concept of local equilibrium must be introduced [1]. If the system is divided into small volume elements such that within each volume element, thermodynamic equilibrium applies, it is valid to consider that volume as being in thermal equilibrium. Hence for each volume element of the system, the Gibbs' fundamental relation is applicable [1, 3]. Using this relation, Ràfols and Ortín (1992) [1] derive an equation for entropy balance in the local volume elements, and from there, derive in full functions for the evolution of the entropy flow and entropy production in time [1]. Their derivation is excluded from this report but they result in the following relations:

$$\frac{d_e S}{dt} \propto \int_0^L \frac{1}{L}\left(\frac{\partial^2 T}{\partial z^2}\right) dz - \int_0^L \frac{1}{T^2}\left(\frac{\partial T}{\partial z}\right) dz - \frac{2H}{\lambda r}\int_0^L (T - T_R) dz \tag{11}$$

for the evolution of the entropy production (derived from the transfer of heat across the boundary of the system), and:

$$\frac{d_i S}{dt} \propto \int_0^L \frac{1}{T^2}\left(\frac{\partial T}{\partial z}\right)^2 dz \tag{12}$$

for the evolution of the entropy flow (due to changes inside the system), where L is the length of the rod along the z axis, T is the temperature of the rod at point z, and H and λ are the surface and linear conduction as previously defined.

Therefore in this experiment, to demonstrate and investigate the steady-state solution of a non-equilibrium thermodynamic system, a rod will be heated to constant temperature at one end, while the other is kept at a constant temperature. The temperature of the rod will be measured along discrete positions as the system evolves, until it eventually reaches the steady state. These temperatures will be used to investigate the properties of the steady-state along with the entropy of the system.

II. METHODOLOGY

A. Apparatus

The experiment was set up as shown in figure 1. The metallic rod was left open to the air on one side, while heated with a furnace on the other. Sixteen type K thermocouples were used to measure the temperature along the rod. These were affixed to the rod using thermal paste to ensure a good thermal contact, and converted the temperature along the rod into a corresponding voltage. As such, these thermocouples needed to be calibrated as elaborated on in the next section. The analogue-to-digital converter (ADC) - used to convert the thermocouple signal to a computer-readable voltage - only had inputs for eight thermocouples, while sixteen were present. A digital-to-analogue converter (DAC) input was used in combination with a 16 to 8 channel switch to switch between two sets of eight inputs which removed this issue.

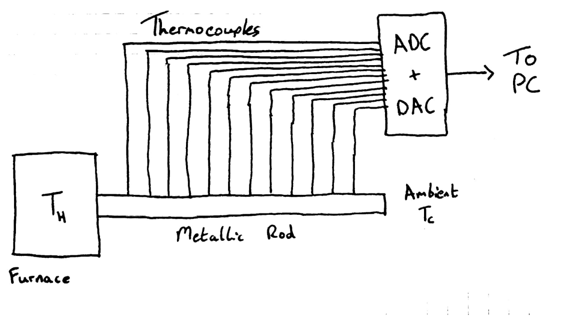


FIG. 1: A sketch of the experimental setup and data acquisition devices [1].

B. Calibration of the Thermocouple

The thermocouples were all calibrated simultaneously through the heating of water. The thermocouples were bundled together and placed into a beaker filled with water, ensuring that all were fully submerged and not touching the edges of the beaker. Starting from ice water, a hot plate was used to slowly heat the water as calibration voltages from the thermocouples were recorded. A simple handheld thermometer was used to measure the temperature for each calibration measurement. These calibration data were plotted, and a quadratic least squares fit was applied to create a function to convert from voltage to temperature.

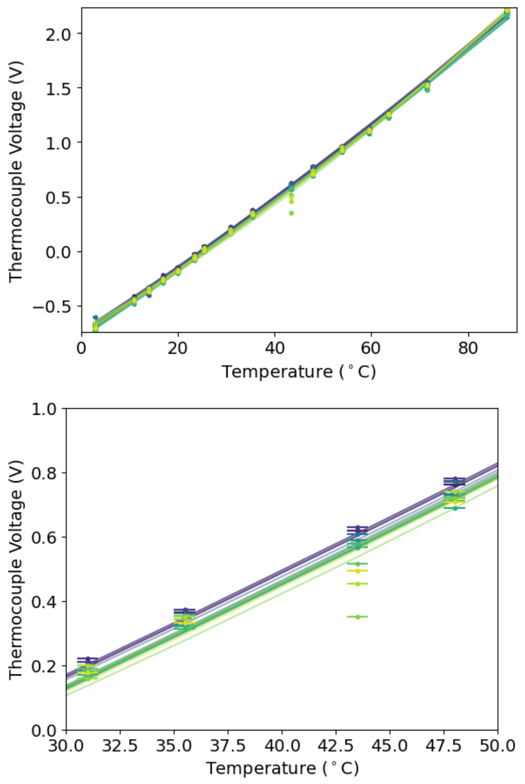


FIG. 2: The full calibration data ranging from 4 °C to 95 °C, a quadratic least squares fit is applied to yield the calibration function. The second panel shows a zoomed region of this plot to show the slight differences between each thermocouple. Each colour shown corresponds to one of the thermocouple pins.

C. Experimental Procedure

1. Data Acquisition

A Python script was written to automate the data collection process. For one hour (or any other specified length of time), the script records the temperatures each minute from each thermocouple along the bar. Starting from uniform equilibrium conditions - i.e. the entire bar was at the ambient temperature of the room - the furnace was turned on and the data collection process started. This was also repeated for non-uniform initial conditions. A hair-drier was used to heat up one section of the bar above the ambient temperature of the room, and again the furnace and data collection process were started. All data was recorded and can be accessed via Github (`https://github.com/daraghhollman/UCD_4thYear_Labs/tree/main/NewtonsCooling`).

2. Entropy Investigation

To investigate the entropy evolution of the system equations 11 and 12 need to be discretised to correspond to each thermocouple. This has been derived by Lurié and Wagensberg [4] in full, resulting in the following:

$$\frac{d_iS}{dt} \propto -\frac{1}{l}\sum_{i=0}^{n-1}(T_{i+1}-T_i)\left(\frac{1}{T_{i+1}}-\frac{1}{T_i}\right) \tag{13}$$

$$\begin{aligned}\frac{d_eS}{dt} \propto & -\frac{1}{l}\frac{1}{T_0}(T_1T_0)+\frac{1}{l}\frac{1}{T_n}(T_n-T_{n-1}) \\ & -\beta^2 l\sum_{i=1}^{n-1}\left(1-\frac{T_n}{T_i}\right)\end{aligned} \tag{14}$$

where T_0 represents the furnace temperature T_H, and T_n represents the ambient temperature T_C. l is the length of each volume element in the discrete model of the rod - which in our case, is simply the distance between each thermocouple pin. These equations can simply be applied itteratively to each pin temperature for each time step to investigate how the entropy production and flow evolve in time.

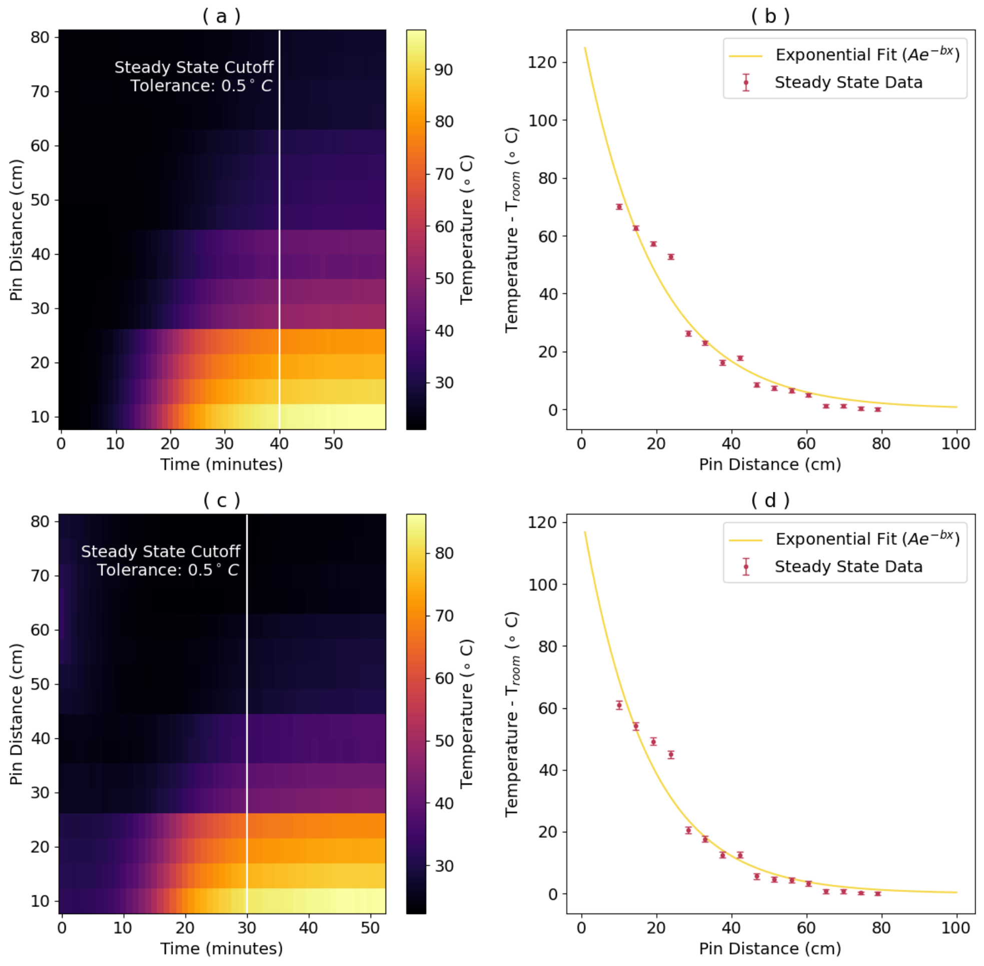


FIG. 3: (a, c) A heat-map plot for the uniform (top panel) and non-uniform (bottom panel) initial conditions data. Heat of each pin is plotted as a function of time. The steady-state solution is reached at the white line. (b, d) The average temperature of each pin after the steady-state condition was reached is plotted. A decaying exponential fit $f(x) = Ae^{-\beta x}$ was plotted and fits the data well.

III. RESULTS AND ANALYSIS

A. Calibration

The thermocouple calibration data was plotted in figure 2. A quadratic least squares fit was applied with the form $Ax^2 + Bx + C$, yielding parameters for each thermocouple. A quadratic fit was chosen as we expect the response from the thermocouple to be non-linear [5].

B. Uniform Initial Conditions

The uniform initial conditions data were plotted as a heat-map (see figure 3a) to show the evolution over time towards the steady-state. We determined that the system had reached the steady state when each point along the rod had a change in temperature from the previous time step of less than 0.5 °C. A white line was drawn atop the heat-map to denote this position. For each time step after this point, the ambient subtracted room temperature was averaged and plotted as points against the distance of the corresponding pin (see figure 3b). The error on each point is the standard deviation of the averaged data for that point. The steady-state is easily observed to be an exponential, consistent with equation 10, and hence a least squares fit of the form $f(x) = Ae^{-\beta x}$ was used to find the furnace temperature ($A+T_C$, where T_C is assumed to be the same temperature as the final pin temperature) $T_H = (158 \pm 1.5)\,°\mathrm{C}$, and the parameter $\beta = (5.17 \pm 0.06)\,\mathrm{m}^{-1}$.

C. Non-Uniform Initial Conditions

This same process was repeated for the non-uniform initial conditions, with the respective heat-map and steady-state plotted in figure 3 (c, d). The non-uniform conditions can be seen in the slightly purple hue centred around a pin distance of 65 cm. Note, while the same steady-state condition of $\Delta T = 0.5\,°\mathrm{C}$ was applied, the steady-state solution is reached 10 minutes earlier. This is likely due to the colder side of the rod already being somewhat hotter and needing a smaller change in temperature to reach the steady state. The furnace temperature was determined to be $(147 \pm 3)\,^{\circ}\mathrm{C}$, and the parameter $\beta = (5.8 \pm 0.1)\,\mathrm{m}^{-1}$.

D. Entropy Investigation

The discretised equations for entropy evolution were calculated for each time-step of both the uniform and non-uniform data sets. These values were recorded and plotted in figure 4. We observe few differences between the uniform data (top panel), and the non-uniform data (bottom panel). The uniform data converge to values of $\frac{dS_e}{dt} = 1.77 \pm 0.02$ and $\frac{dS_i}{dt} = 0.69 \pm 0.004$ while the non-uniform data converges to values of $\frac{dS_e}{dt} = 1.8 \pm 0.03$ and $\frac{dS_i}{dt} = 0.7 \pm 0.002$, both calculated based on the mean and standard deviation of the final five data points. We see that these values are the same between the uniform and non-uniform data, however, from previous results, we do expect the entropy production and flow to converge on the same value [1], which is not observed. This slight imbalance in the entropy production would potentially suggest that the system is not in local thermal equilibrium, however, we clearly see this is the case from the heat-maps.

The figures determined for the entropy flow $\frac{dS_i}{dt} = 0.69 \pm 0.004$ (uniform) and $\frac{dS_i}{dt} = 0.7 \pm 0.002$ (non-uniform) compare well with analytic measurements for similar experimental setup. Ràfols and Ortín (1992) find $\frac{dS_i}{dt} = 0.742$ (uniform) and $\frac{dS_i}{dt} = 0.765$ (non-uniform) with similar temperature and β conditions [1].

IV. CONCLUSION

The steady-state solution of a metallic rod subject to a fixed temperature difference was investigated analytically and experimentally. It was demonstrated that the system evolved to the steady-state for uniform and non-uniform initial temperature distributions, showing that the steady-state depends solely on the boundary conditions. However, we do see that initial conditions do have the effect of changing the speed at which the system reaches the steady-state. Steady-states were demonstrated with furnace temperatures $T_H = (158 \pm 1.5)\,^{\circ}\mathrm{C}$, and $(147 \pm 3)\,^{\circ}\mathrm{C}$, resulting in exponential decay constants $\beta = (5.17 \pm 0.06)\,\mathrm{m}^{-1}$, and $\beta = (5.8 \pm 0.1)\,\mathrm{m}^{-1}$, corresponding well to previously produced results [1]. We also see - with a brief investigation of entropy in the system - that the entropy production and flow decrease to a minimum as the system evolves to steady-state.

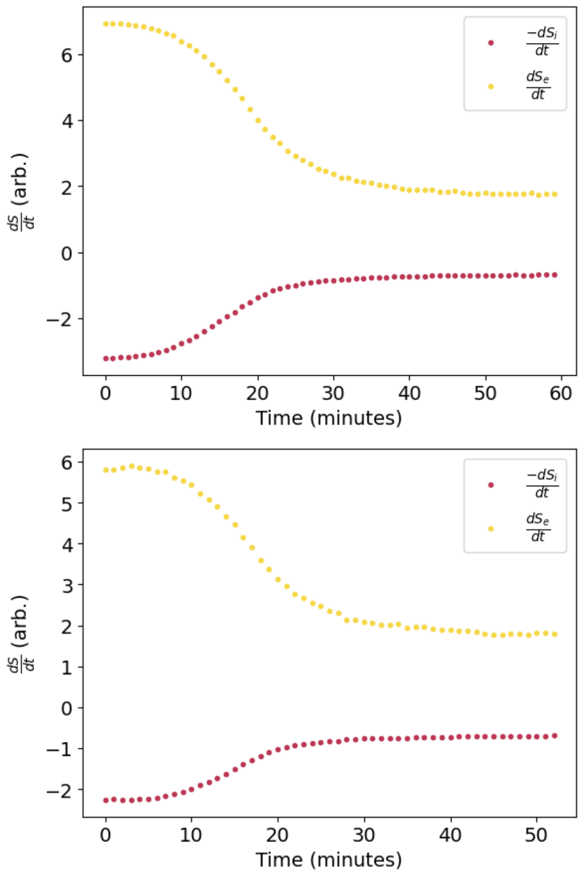


FIG. 4: Evolution of entropy in the system as it approaches steady state. In the top panel is the entropy production and flow for the uniform initial conditions, and in the bottom panel is the production and flow for the non-uniform initial conditions. We see the steady-state characterised by the data reaching a minimum and stable value.

[1] Ismael Rafols and Jordi Ortin. Heat conduction in a metallic rod with newtonian losses. *American journal of physics*, 60(9):846–852, 1992.

[2] Albert Díaz-Guilera. On heat conduction in one-dimensional solids. *American journal of physics*, 58(8):779–780, 1990.

[3] A.M. Steane and Oxford University Press. *Thermodynamics: A Complete Undergraduate Course*. Oxford University Press, 2016.

[4] David Lurié and Jorge Wagensberg. Concepts of nonequilibrium thermodynamics in discrete model of heat conduction. *American journal of physics*, 48(10):868–872, 1980.

[5] Calibrating thermocouples. `https://www.thomasnet.com/articles/instruments-controls/calibrating-thermocouples/`, 2020. Accessed on 2024-02-14.

Appendix A: Python Notebook

In [ ]:
```
from pydaqmx_helper.adc import ADC
from pydaqmx_helper.dac import DAC
```

In [ ]:
```
from tqdm import tqdm
from glob import glob
import time
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from scipy.optimize import curve_fit
from matplotlib.colors import LogNorm

plt.rcParams.update({'font.size': 14})
```

In [ ]:
```
thermocoupleADC = ADC()
thermocoupleDAC = DAC(0)
```

In [ ]:
```
thermocoupleADC.addChannels([0, 1, 2, 3, 4, 5, 6, 7], ADC_mode="DAQmx_Val_RSE")
thermocoupleADC.getActiveChannels()
```

In [ ]:
```
thermocoupleADC.sampleVoltages()
```

Calibration

In [ ]:
```
# Need to convert from adc pin voltage to temperature
```

In [ ]:
```
def SwitchPinLayer(dac, layer):

    # We have 16 thermocouples but only 8 ADC ouputs, a 16 to 8 channel switch is used
    # We describe the two sets of thermocouples as layers, and switch between them using this function
    # This function sets voltage switch to the value corresponding to the 'layer' parameter which is 0 or 1

    if layer == 0:
        dac.writeVoltage(0)

    elif layer == 1:
        dac.writeVoltage(5)

    else:
        raise ValueError("Parameter: 'Layer' must be of type int and either 0 or 1")

    return
```

In [ ]:
```python
def MeasureTemperatures(adc, dac, calibrationFunction=None):
    # Function to record the temperatures of all the pins
    # Returns a list with the length of the number of pins, with each value being the voltage of pin 'i'.

    # Define a new list containing thermocouple outputs

    temperatures = list()

    # First ensure pin switch is on layer 0
    SwitchPinLayer(dac, 0)

    # Record voltages for each pin
    firstPins = adc.sampleVoltages()

    for el in firstPins.values():
        temperatures.append(el[0]) # first element of el is needed as the values of the returned dictionary are tuples

    # Switch pins to second layer
    SwitchPinLayer(dac, 1)

    # Record voltages for each pin
    secondPins = adc.sampleVoltages()

    for el in secondPins.values():

        temperatures.append(el[0])

    return temperatures
```

In [ ]:
```python
MeasureTemperatures(thermocoupleADC, thermocoupleDAC)
```

In [ ]:
```python
def SaveCalibrationData(path, temperature):
    data = MeasureTemperatures(thermocoupleADC, thermocoupleDAC)

    fullPath = path + "/nc_calibration_" + str(temperature)

    np.savetxt(fullPath, data)
```

In [ ]:
```python
SaveCalibrationData(r"C:\Users\Student\Desktop\20302561\Calibration", "test")
```

In [ ]:
```python
SaveCalibrationData(r"C:\Users\Student\Desktop\20302561\Calibration", 71.5)
```

Loading Calibration Data

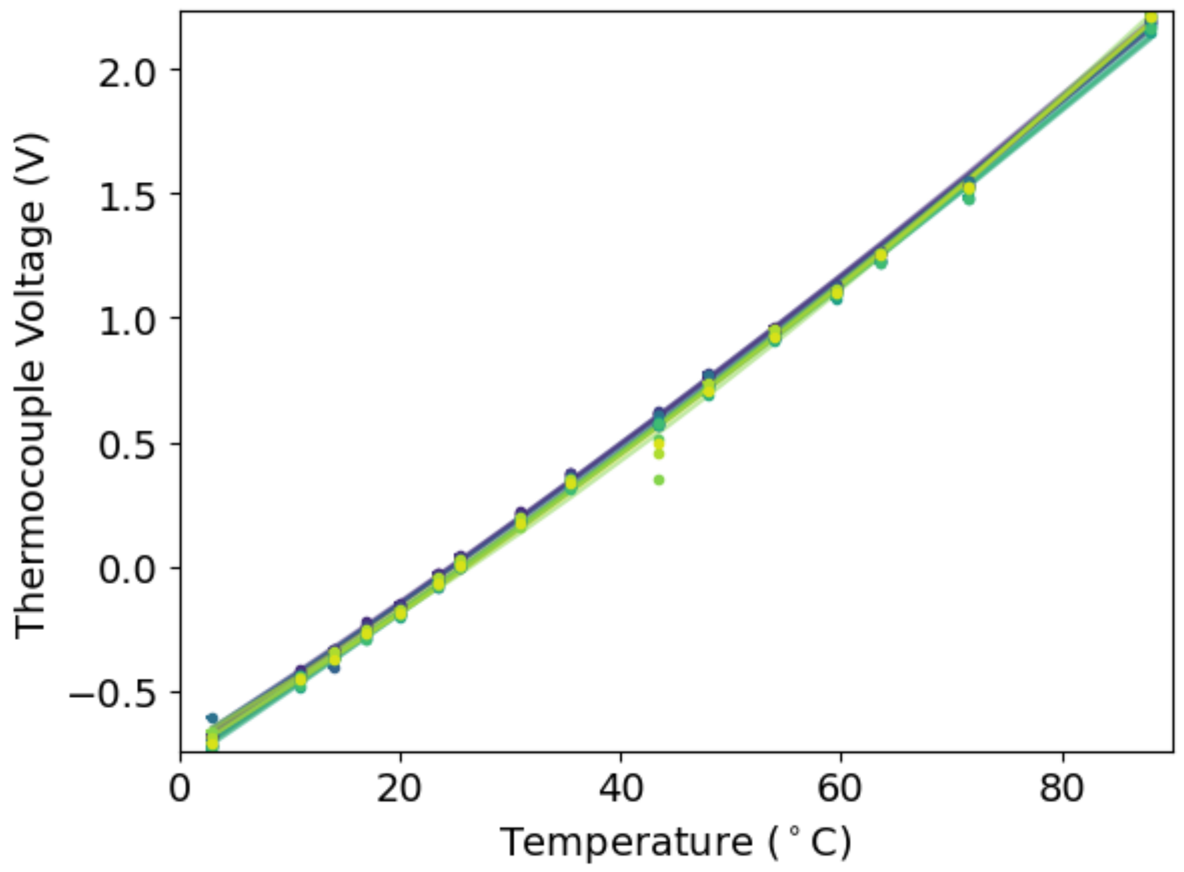
In [ ]:
```
def GetCalibrationData(path):
    paths = glob(path)
    paths.sort(key = lambda x: float(x.split('_')[-1]))

    data = [0] * len(paths)
    temperature = [float(el.split('_')[-1]) for el in paths]

    for i, path in enumerate(paths):
        data[i] = np.loadtxt(path)

    return [temperature, np.array(data)]
```

In [ ]:
```
def QuadraticFit(x, a, b, c):

    if type(x) == np.dtype('float64'):
        x = [x]

    output = []

    for el in x:
        output.append(a*el**2 + b*el + c)

    return output
```

In [ ]:
```python
def PlotCalibration(calibrationData):

    fig, ax = plt.subplots()

    pins = [ calibrationData[1][:,i] for i in range(len(calibrationData[1]))]

    fitPars = []
    fitCov = []

    for i, pinValues in enumerate(pins):

        ax.errorbar(calibrationData[0], pinValues, xerr=0.5, fmt=".", color=plt.get_cmap("viridis").__call__(i / 16), label=f"Thermocouple Index {i}")

        #print(calibrationData[0])
        #print(pinValues)

        pars, cov = curve_fit(QuadraticFit, calibrationData[0], pinValues)

        ax.plot(calibrationData[0], QuadraticFit(calibrationData[0], pars[0], pars[1], pars[2]), alpha=0.5,
                                                  color=plt.get_cmap("viridis").__call__(i / 16))

        fitPars.append(pars)
        fitCov.append(cov)


    ax.set_xlabel(r"Temperature ($^\circ$C)")
    ax.set_ylabel("Thermocouple Voltage (V)")
    ax.margins(0)
    ax.set_xlim(0, 90)
    #plt.legend()

    return (fitPars, fitCov)
```

```
In [ ]: def CreateCalibrationFunction(fitPars):

            def CalibrationFunction(voltages):

                temperatures = []

                for i, pinVoltage in enumerate(voltages):

                    quadratic = np.polynomial.Polynomial(np.flip(fitPars[i]))

                    roots = (quadratic - pinVoltage).roots()

                    for r in roots:
                        if r > 0:
                            root = r

                    temperatures.append(root)

                return temperatures

            return CalibrationFunction
```

```
In [ ]: calibrationData = GetCalibrationData(r"../data/calibration/*")
        fitPars, fitCov = PlotCalibration(calibrationData) # Thermocouple error is neglegible
        CalibrationFunction = CreateCalibrationFunction(fitPars)
        #print(fitPars)
        #print(np.sqrt(np.diag(fitCov)))
        #plt.ylim(0, 1)
        #plt.xlim(30, 50)
```


Thermocouple Voltage (V)
Temperature (°C)
2.0
1.5
1.0
0.5
0.0
−0.5
0
20
40
60
80

Experiment

In [ ]:
```
def ExperimentRun(measurements, timeBetween):

    # measurements: Number of measurements to be made
    # timeBetween: Time to wait between each measurment (seconds)

    # files containing the data for each measurment will be saved before the next one begins

    startTime = time.time()

    for i in tqdm(range(measurements)):

        print(f"Taking measurment {i} of {measurements}")

        ## DO MEASUREMENT

        print("Measurment complete, writing to file")

        currentTime = time.time() - startTime
        currentTimeRounded = int(currentTime)

        ## Write to file. Note that filename should contain the time at which the measurment was taken

        print(f"Waiting {timeBetween} seconds")

        time.sleep(timeBetween)

    print("All measurments complete, exiting function")
    totalTime = time.time() - startTime
    print(f"Total time ellapsed: {totalTime}")

    return
```

Plotting

In [ ]:
```
def GetData(path):
    paths = glob(path)
    paths.sort(key = lambda x: float(x.split('_')[-1]))

    data = [0] * len(paths)
    time = [float(el.split('_')[-1]) for el in paths]

    for i, path in enumerate(paths):
        data[i] = np.loadtxt(path)

    return [time, np.array(data)]
```

In [ ]:
```python
def PinNumberToDistance(pinNumbers):
    # Takes pin index and return the distance from the resevoir in cm

    firstDistance = 10 # 10 cm
    interDistance = 4.6 # 46 mm

    pinDistances = []

    for pinNumber in pinNumbers:
        pinDistances.append(firstDistance + interDistance * pinNumber)

    return pinDistances
```

```
In [ ]: def dS_internal(pinTemperatures, length, furnaceTemperature):

            total = 0
            temperatures = [furnaceTemperature, *pinTemperatures]
            temperatures = [t + 273.15 for t in temperatures]

            for i in range(0, len(temperatures) - 2):
                currentT = temperatures[i]
                nextT = temperatures[i + 1]

                total += (nextT - currentT) * ((1 / nextT) - (1 / currentT))

            return ( - 1 / length) * total

        def dS_external(pinTemperatures, length, beta, furnaceTemperature, t):

            temperatures = [furnaceTemperature, *pinTemperatures]
            temperatures = [t + 273.15 for t in temperatures]

            a = (- 1 / length) * (1 / temperatures[0]) * (temperatures[1] - temperatures[0])
            b = (1 / length) * (1 / temperatures[-1]) * (temperatures[-1] - temperatures[-2])

            c = 0

            for i in range(1, len(temperatures) - 2):

                c += -1 * (beta*100)**2 * length * (1 - (temperatures[-1] / temperatures[i]))

            if t == 5:
                print(temperatures)
                print(f"A: {a}")
                print(f"B: {b}")
                print(f"C: {c}")

            return a + b + c
```

In [ ]:

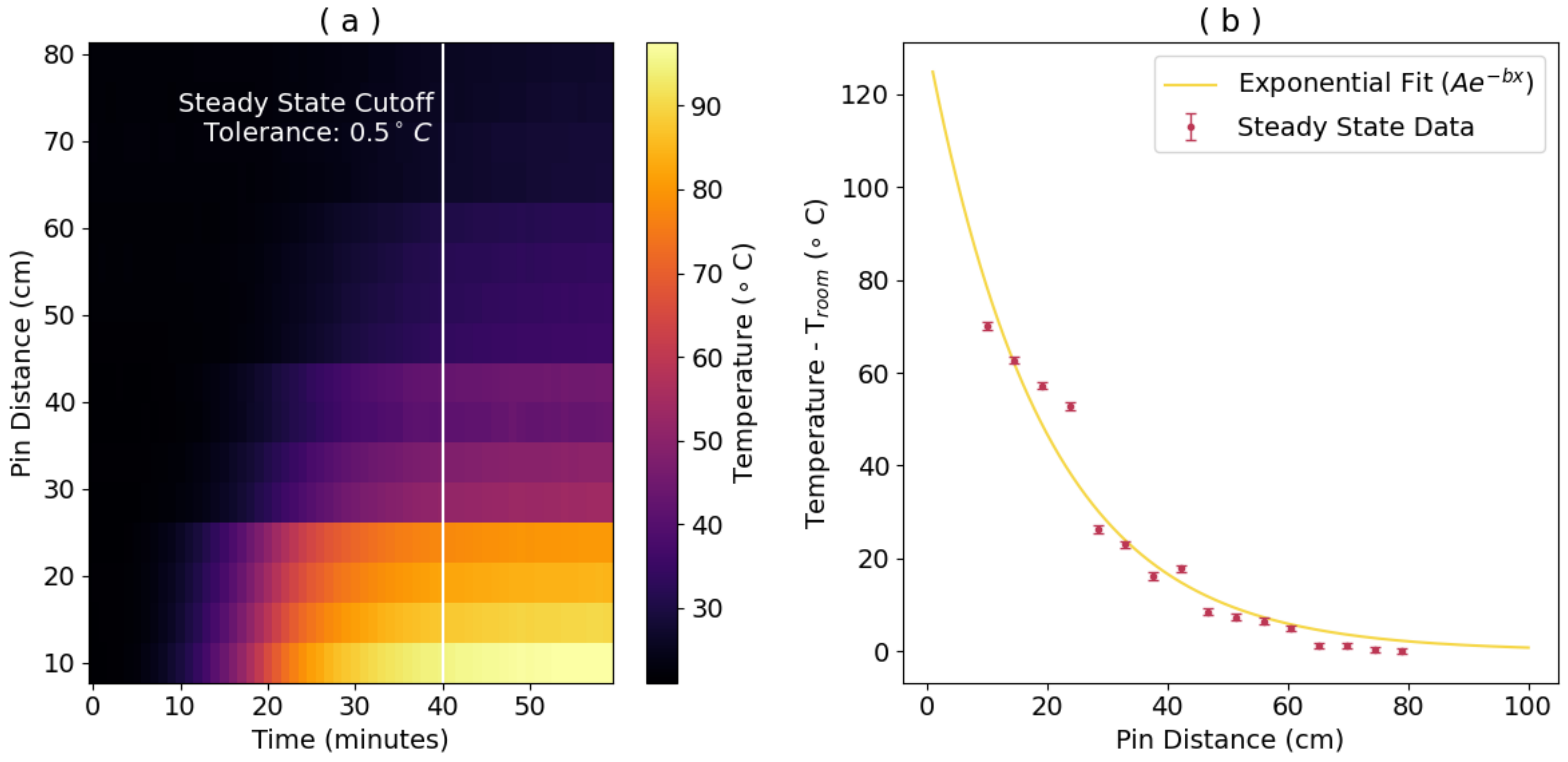
```python
def DetermineSteadyStateCutoff(temperatureArray, tolerance):
    # Finds the time along the array where the temperatures change by no greater than the tolerance for each pin
    timeIndex = 20

    while timeIndex < len(temperatureArray):

        if timeIndex == len(temperatureArray) -1:
            raise Exception("Steady state not found, tolerance too high!")

        elif np.any( (temperatureArray[timeIndex + 1 ] - temperatureArray[timeIndex]) > tolerance ):
            timeIndex += 1
            continue

        else:
            print("Complete at t=" + str(timeIndex))
            break

    return timeIndex

def PlotSteadyState(ax, temperatureArray, plot=True):

    startIndex = DetermineSteadyStateCutoff(temperatureArray, 0.5)

    steadyStates = temperatureArray[startIndex:-1]

    meanTemperatures = []
    temperatureStds = []

    i = 0
    while i < len(steadyStates[0]):
        meanTemperatures.append(np.mean(steadyStates[:,i]))
        temperatureStds.append(np.std(steadyStates[:,i]))
        i += 1

    pinPositions = PinNumberToDistance(np.arange(len(temperatureArray.T)))

    # FITTING
    def ExpFunc(x, a, b):
        if type(x) == np.dtype('float64'):
            x = [x]

        output = []

        for el in x:
            output.append(a * np.exp(- b * el))

        return output
    # Subtract room temperature from mean temperatures
    roomTemperature = meanTemperatures[-1]
    meanTemperatures = meanTemperatures - roomTemperature

    pars, cov = curve_fit(ExpFunc, pinPositions, meanTemperatures, [80, 0.5], sigma=temperatureStds, absolute_sigma=True)


    # PLOTTING
    if plot:
        cmap = cm.get_cmap('inferno')

        a, b= pars
        print(pars)
        xRange = np.linspace(1, 100, 1000)
        ax.plot(xRange, ExpFunc(xRange, a, b), color=cmap(0.9), label="Exponential Fit ($Ae^{-bx}$)")

        ax.errorbar(pinPositions, meanTemperatures, yerr=temperatureStds, fmt=".", color=cmap(0.5), linewidth=1, capsize=3, label="Steady State Data")
        ax.set_ylabel(r"Temperature - T$_{room}$ ($\circ$ C)") # No need for this one with the colorbar in the middle
        ax.set_xlabel("Pin Distance (cm)")
        ax.legend()

    return (startIndex, pars, cov, roomTemperature)
```

In [ ]:

```
def MakePlots(data, entropy=False, heatMap=False):

    # REFORMATING DATA
    temperatureArray = np.zeros((len(data[0]), len(data[1][0])))

    for i, timeRow in enumerate(temperatureArray):
        temperatures = CalibrationFunction(data[1][i]) # convert from volatages to Celcius
        temperatureArray[i] = temperatures

    # HEATMAP PLOTTING
    if heatMap:
        fig, axes = plt.subplots(1, 2, figsize=(12,6))

        x = np.arange(len(temperatureArray))
        y = PinNumberToDistance(np.arange(len(temperatureArray.T)))

        cbar = axes[0].pcolormesh(x, y, temperatureArray.T, cmap="inferno")
        fig.colorbar(cbar, label=r"Temperature ($\circ$ C)")

        axes[0].set_xlabel("Time (minutes)")
        axes[0].set_ylabel("Pin Distance (cm)") # Note we need to convert from pin number to distance from hot resevoir
        # First pin is 10 cm away from hot resevoir
        # Each pin after is 46 mm away from the previous

        startIndex, _, _, _ = PlotSteadyState(axes[1], temperatureArray)
        cutOffColour = "white"
        axes[0].axvline(startIndex, color=cutOffColour)
        axes[0].text(startIndex - 1, 70, "Steady State Cutoff\nTolerance: 0.5$^\circ \,C$", color=cutOffColour, horizontalalignment="right")

        axes[0].set_title("( c )")
        axes[1].set_title("( d )")

        plt.tight_layout()

    # ENTROPY PLOTS
    if entropy:
        fig = plt.figure()
        ax = fig.add_subplot(111)

        _, pars, cov, roomTemperature = PlotSteadyState(None, temperatureArray, plot=False)
        beta = pars[1]
        print(f"Beta: ({beta*100} +/- {np.sqrt(cov[1][1]) * 100}) 1/m")

        internalEntropy = []
        externalEntropy = []

        for i, timeRow in enumerate(temperatureArray):
            temperatures = CalibrationFunction(data[1][i]) # convert from volatages to Celcius
            temperatureArray[i] = temperatures

            length = 0.046 # 46 mm
```

```
In [ ]: MakePlots(GetData("/home/daraghhollman/Main/ucd_4thYearLabs/NewtonsCooling/data/Uniform/*"), heatMap=True, entropy=False)
```

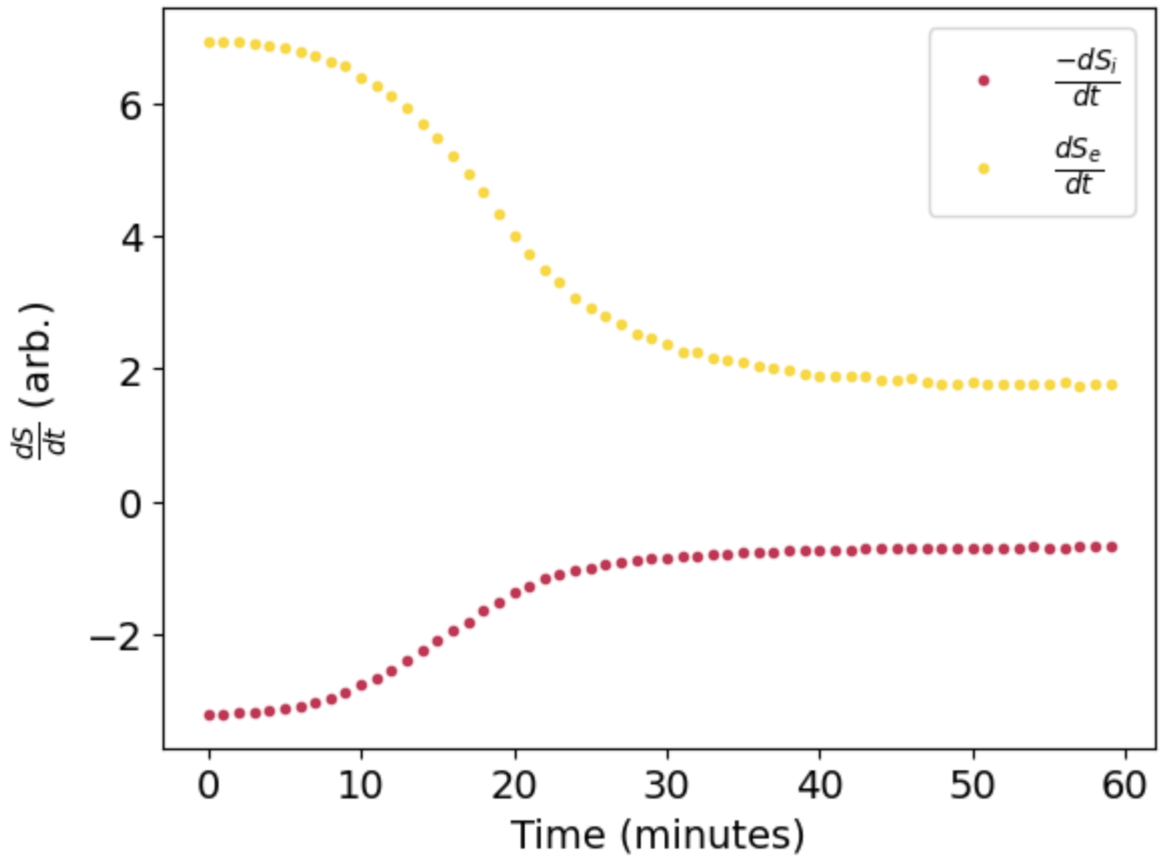
```
                print(f"Furnace Temperature: ({pars[0] + roomTemperature} +/- {np.sqrt(cov[0][0])}) C")
                print(temperatures)

            internalEntropy.append(-1 *dS_internal(temperatures, length=length, furnaceTemperature=pars[0] + roomTemperature))
            externalEntropy.append(dS_external(temperatures, length, beta, pars[0] + roomTemperature, i))

        cmap = cm.get_cmap('inferno')
        print(f"EXT: {np.mean(externalEntropy[-5:-1])} +/- {np.std(externalEntropy[-5:-1])}")
        print(f"INT: {np.mean(internalEntropy[-5:-1])} +/- {np.std(internalEntropy[-5:-1])}")
        ax.scatter(np.arange(len(temperatureArray)), internalEntropy, marker=".", label=r"$\frac{-dS_i}{dt}$", color=cmap(0.5))
        ax.scatter(np.arange(len(temperatureArray)), externalEntropy, marker=".", label=r"$\frac{dS_e}{dt}$", color=cmap(0.9))

        ax.set_xlabel("Time (minutes)")
        ax.set_ylabel(r"$\frac{dS}{dt}$ (arb.)")

        ax.legend()
```



```
In [ ]: MakePlots(GetData("/home/daraghhollman/Main/ucd_4thYearLabs/NewtonsCooling/data/Uniform/*"), heatMap=False, entropy=True)
```

```
Complete at t=40
Beta: (5.165907822168314 +/- 0.060112855360566875) 1/m
Furnace Temperature: (157.89919748866188 +/- 1.648150788615526) C
[22.52098009479071, 22.06285973366716, 22.11755452796978, 21.99213999574022, 21.354390818915398, 21.608617697100062, 21.174761499889712, 21.3075959553171
47, 21.39712656572045, 21.804156061492282, 21.82499795191574, 21.687824962187555, 22.061473653663892, 22.49886003875767, 22.06476766989573, 22.0020723612
7147]
[431.0491974886619, 295.6709800947907, 295.21285973366713, 295.26755452796976, 295.1421399957402, 294.5043908189154, 294.7586176971, 294.3247614998897, 2
94.4575959553171, 294.5471265657204, 294.95415606149226, 294.9749979519157, 294.83782496218754, 295.21147365366386, 295.6488600387577, 295.2147676698957,
295.15207236127145]
A: 6.827537884538418
B: -0.004617760197066815
C: 0.010918695108608083
EXT: 1.7656794510388836 +/- 0.016215066639972787
INT: -0.6865977784818382 +/- 0.0035992490745622545
```

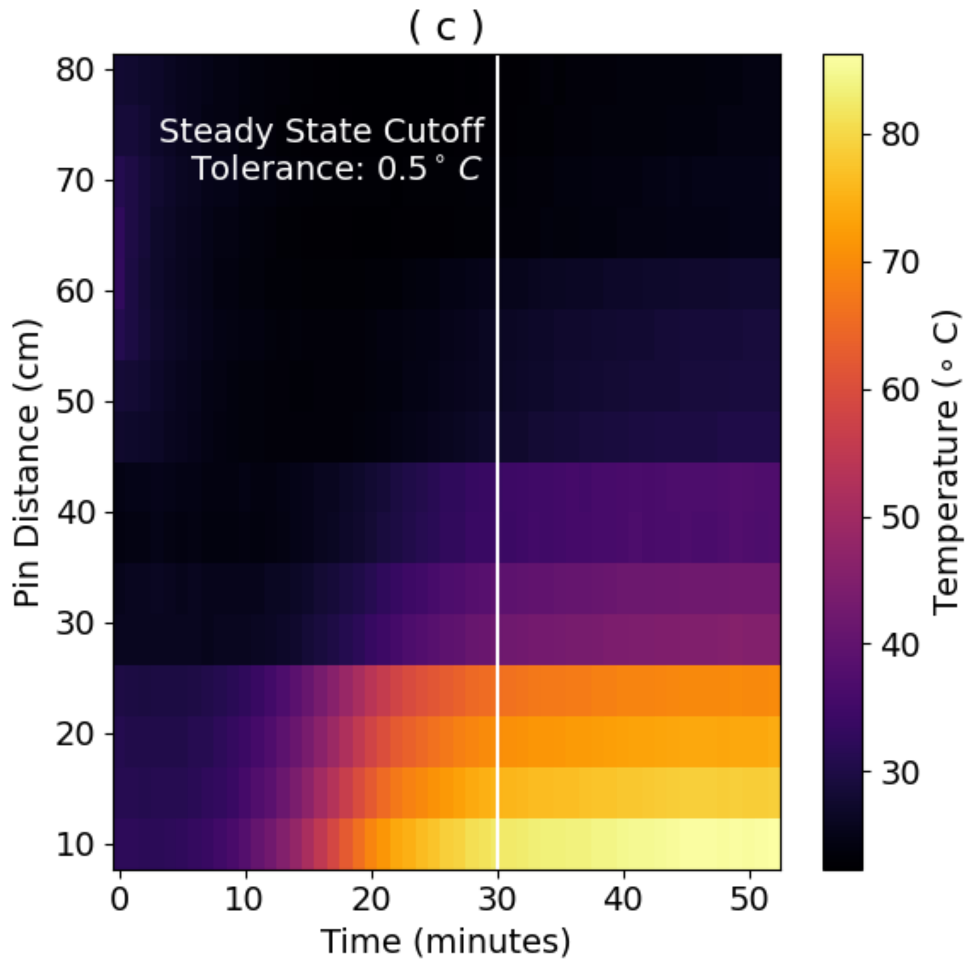
```
/tmp/ipykernel_1180800/1556911531.py:45: RuntimeWarning: overflow encountered in exp
  output.append(a * np.exp(- b * el))
/tmp/ipykernel_1180800/2982705687.py:61: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be removed two min
or releases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap(obj)`` instead.
  cmap = cm.get_cmap('inferno')
```

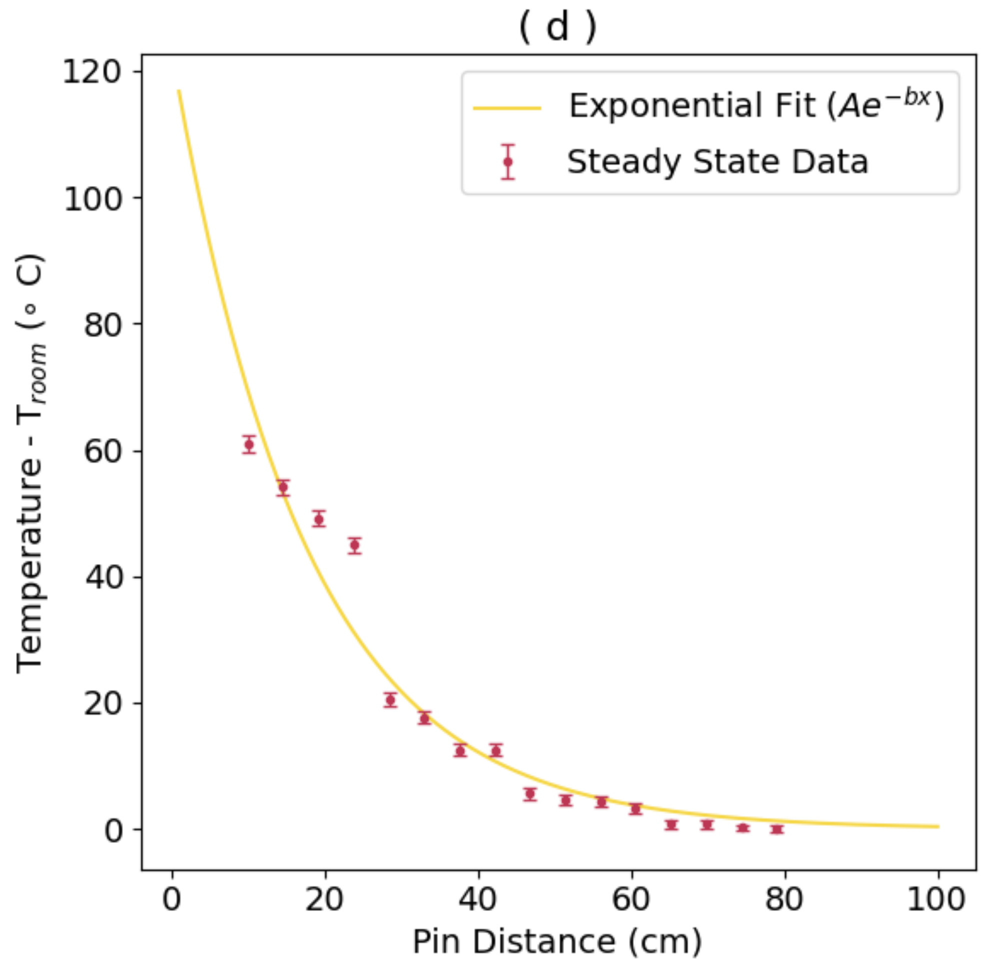


In [ ]:
```
MakePlots(GetData("/home/daraghhollman/Main/ucd_4thYearLabs/NewtonsCooling/data/NonUniform/*"), heatMap=True, entropy=False)
```

```
Complete at t=30
[1.23659838e+02 5.78949973e-02]
```

```
/tmp/ipykernel_1180800/1556911531.py:45: RuntimeWarning: overflow encountered in exp
  output.append(a * np.exp(- b * el))
/tmp/ipykernel_1180800/1556911531.py:58: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be removed two min
or releases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap(obj)`` instead.
  cmap = cm.get_cmap('inferno')
```

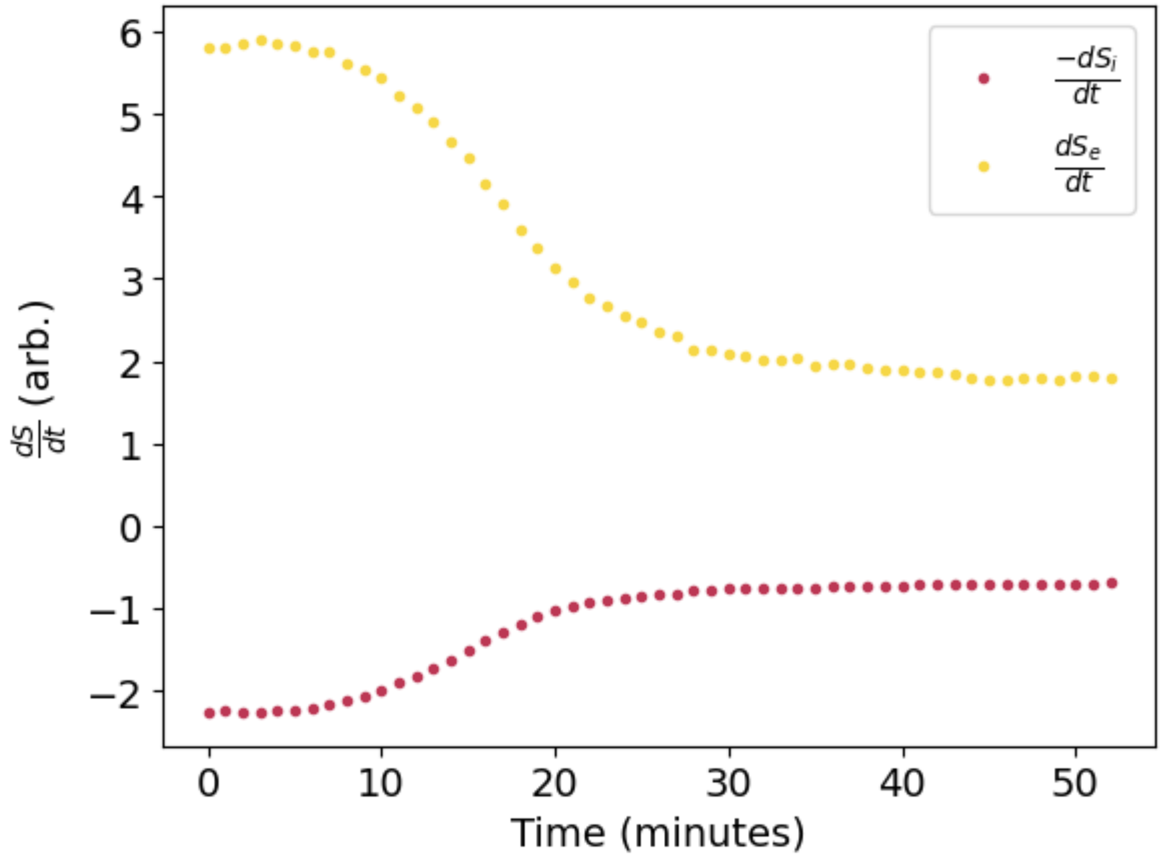




```
In [ ]: MakePlots(GetData("/home/daraghhollman/Main/ucd_4thYearLabs/NewtonsCooling/data/NonUniform/*"), heatMap=False, entropy=True)
```

```
Complete at t=30
Beta: (5.789499725997166 +/- 0.10534723968606945) 1/m
Furnace Temperature: (147.16596707831744 +/- 2.6984035778922615) C
[32.45982740811246, 31.14523893335278, 30.569625167500273, 29.69599880323655, 25.887608513986137, 25.508448039951798, 24.187015205545933, 24.621437841687
484, 25.63037788657392, 25.678425742494113, 26.098344372393594, 25.892920362215772, 26.007213005694705, 26.31266507003041, 25.74613428699488, 25.66189303
6318727]
[420.3159670783174, 305.60982740811244, 304.2952389333528, 303.7196251675002, 302.8459988032365, 299.03760851398613, 298.6584480399518, 297.3370152055459
, 297.7714378416875, 298.7803778865739, 298.8284257424941, 299.24834437239355, 299.04292036221574, 299.1572130056947, 299.46266507003037, 298.89613428699
49, 298.8118930363187]
A: 5.932707599224616
B: -0.006128710332880056
C: -0.10336200579604853
EXT: 1.8000898840512896 +/- 0.026124723842881455
INT: -0.7002391646568079 +/- 0.002617424191552274
```

```
/tmp/ipykernel_1180800/1556911531.py:45: RuntimeWarning: overflow encountered in exp
  output.append(a * np.exp(- b * el))
/tmp/ipykernel_1180800/2341118038.py:61: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be removed two min
or releases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap(obj)`` instead.
  cmap = cm.get_cmap('inferno')
```


$\frac{dS}{dt}$ (arb.)
Time (minutes)
$\frac{-dS_i}{dt}$
$\frac{dS_e}{dt}$