ENGR 133 Individual Project Report

LC4 - 2

Ethan Kessel

Group Members:
Jasper Jones, Nathan Yakupkovic, David Kopp

Purdue University

December 7, 2020

Contents

List of Figures

List of Tables

1 Introduction

This project will be exploring the electrical behavior of an astable multivibrator oscillator circuit making use of an operational amplifier using Python and the integrate module of scipy. In a world where microcontrollers allow for easy control of electrical outputs and small devices, more elegant electromechanical solutions have fallen to the wayside of the entry-level hobbyist or engineer. Although my first choice for my major is Aeronautical and Astronautical Engineering, I still have a passion for the electronics that power modern avionics and our daily lives, and want to gain a better understanding of the operation of circuits. This semester, I took an electronics workshop through the Office of Professional Practice, during which we constructed a circuit to blink an LED using nothing but passive components and an op-amp IC. Such a circuit is exceedingly easy to implement using an Arduino microcontroller, but the task can be achieved just as effectively with a much more elegant solution that is cheap and does not require a computer unnecessarily. A reconstruction of the circuit I constructed is shown in Figure 1.

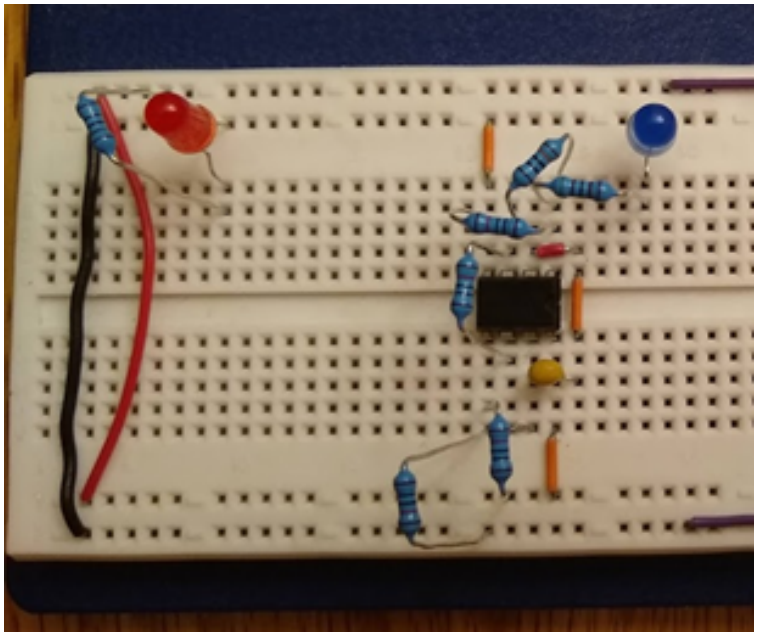


Figure 1: Op-Amp Circuit Assembled on a Solderless Breadboard

During the assignment where I was building this circuit, I did not have a good intuition as to how it functioned. I decided that I would attempt to model the circuit in Python using the integrate module of scipy to solve a differential system numerically. I had to perform some research into the behavior of operational amplifiers as well as apply knowledge gained from my participation in PHYS 272 this semester to come to a solution. I then also compared my theoretical results to reality using an oscilloscope I have on-hand at home, which showed some interesting results.

2 Problem Analysis

2.1 Circuit Diagram

The first step to analyzing the problem was to draw a circuit diagram. One had been provided as part of the workshop however it did not fully match the circuit that was actually built and furthermore did not let me intuitively understand what was occurring in the circuit. I utilized Autodesk EAGLE to redraw the circuit diagram of my circuit, which is shown in Figure 2.

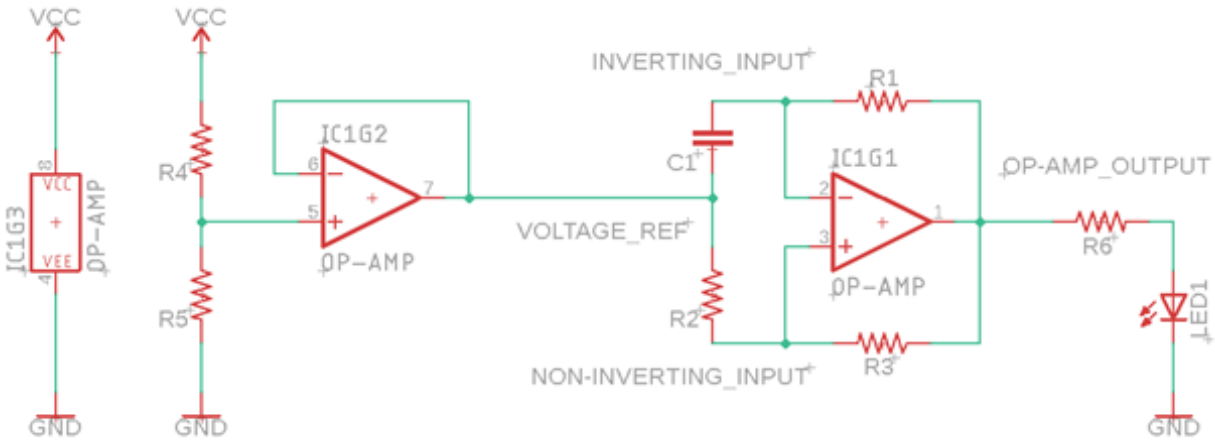


Figure 2: Autodesk EAGLE Circuit Diagram

The circuit has two main functional parts, each using a singular operational amplifier. On the left is what is known as a voltage follower; it is an amplifier set up with negative feedback such that it produces unity gain. What this is useful for is maintaining a specific voltage at a node regardless of the current drawn. The input of this voltage follower is connected to a voltage divider (consisting of resistors `R4` and `R5`), and copies the voltage produced at that node to the output. On the right is the astable multivibrator oscillator. It consists of a voltage divider and an RC circuit, connected between a constant reference voltage produced by the voltage follower and the op-amp's output. Further explanation of how this produces an oscillating signal will be discussed once the behavior of an op-amp and the circuit components are discussed.

2.2 Operational Amplifier Behavior

Early in the research for this project I needed to find a way to model the behavior of an operational amplifier. Fortunately, this ended up being very simple. I was able to find a textbook chapter online [1] that provided an excellent explanation of the material, however the chapter has since become pay-for and I can no longer access the material on the page, so I will continue using material from my notes. The model of an operational amplifier is quite simple and shown in Equation 1.

$$v_{\text{out}} = \begin{cases} V_{CC} & \text{if } v_{\text{out}} > V_{CC} \\ A\left(v^+ - v^-\right) + \dfrac{V_{CC} + V_{EE}}{2} & \text{if } V_{EE} \leq v_{\text{out}} \leq V_{CC} \\ V_{EE} & \text{if } v_{\text{out}} < V_{EE} \end{cases} \tag{1}$$

These equations show that the output of the operational amplifier is linearly proportional to the differential input voltage $(v^+ - v^-)$ but is capped between the voltages supplied to the op-amp. Visually, this is shown in Figure 3.

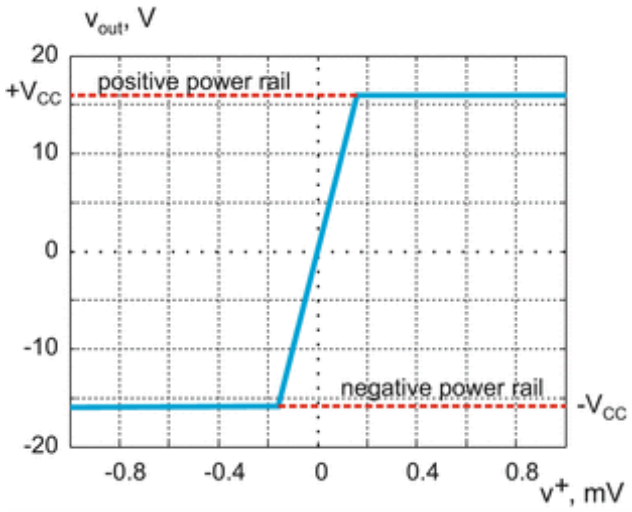


Figure 3: Output of an Operational Amplifier vs. Differential Input Voltage

Because the amplification value A is often very high (on the order of 10^6 or more), an operational amplifier in open-loop operation is almost always saturated at the voltages supplied, producing a digital signal. In this sense, the op-amp is a voltage comparator, producing a positive output when v^+ is greater than v^-, and a negative output when the opposite is true. However, it is important to note that this is ideal behavior. In reality, the chapter noted that the output is often a bit short of completely reaching supplied voltages. For my purposes I ignored this and approximated the op-amp as ideal. Furthermore, a real op-amp has several other non-ideal characteristics, such as a non-infinite impedance between the two inputs, and a nonzero impedance of the output. The effects of these are negligible and vastly complicate the modeling, so I will approximate them away as well.

In the specifications for operational amplifiers, the unitless amplification value A is not given and instead a gain in decibels is specified. The formula to convert between decibels and amplification is rather simple and is shown in Equation 2.

$$\text{Gain} = 20\text{dB} \cdot \log_{10} A \quad \leftrightarrow \quad A = 10^{\left(\frac{\text{Gain}}{20\text{dB}}\right)} \tag{2}$$

2.3 Mathematical Analysis of Circuit Dynamics

The oscillator shown in Figure 2 consists of two common electrical devices, a voltage divider and an RC circuit. The op-amp measures the voltage of the voltage divider with its non-inverting input, and the voltages of the RC circuit with its inverting input. Both the voltage

divider and RC circuit are connected across the regulated voltage reference and the operational amplifier output. The voltage at the non-inverting input, as created by the voltage divider, is given by Equation 3.

$$v^+ = \frac{(v_{\text{out}} - v_{\text{ref}})\, R_2}{R_2 + R_3} + v_{\text{ref}} \tag{3}$$

The equations for the RC circuit section are a bit more complicated and involve differential equations. Using Kirchhoff's Voltage Law, Ohm's Law, and the equation for the voltage drop of a capacitor, Equation 4 is obtained.

$$\begin{aligned} \Delta V_{\text{capacitor}} + \Delta V_{\text{resistor}} &= v_{\text{out}} - v_{\text{ref}} \\ \frac{Q}{C} + IR_1 &= v_{\text{out}} - v_{\text{ref}} \\ \frac{Q}{C} + \dot{Q}R_1 &= v_{\text{out}} - v_{\text{ref}} \\ \dot{Q} &= \frac{v_{\text{out}} - v_{\text{ref}}}{R_1} - \frac{Q}{R_1 C} \end{aligned} \tag{4}$$

Using the charge on the capacitor Q the voltage at the inverting input can be found as in Equation 5.

$$\begin{aligned} v^- &= v_{\text{ref}} + \Delta V_{\text{capacitor}} \\ &= v_{\text{ref}} + \frac{Q}{C} \end{aligned} \tag{5}$$

All of these may be combined into a large system of equations (Equation 6).

$$\begin{cases} \dot{Q} &= \dfrac{v_{\text{out}} - v_{\text{ref}}}{R_1} - \dfrac{Q}{R_1 C} \\ v^- &= v_{\text{ref}} + \dfrac{Q}{C} \\ v^+ &= \dfrac{(v_{\text{out}} - v_{\text{ref}})\, R_2}{R_2 + R_3} + v_{\text{ref}} \\ v_{\text{out}} &= \begin{cases} V_{CC} & \text{if } v_{\text{out}} > V_{CC} \\ A\left(v^+ - v^-\right) + \dfrac{V_{CC} + V_{EE}}{2} & \text{if } V_{EE} \le v_{\text{out}} \le V_{CC} \\ V_{EE} & \text{if } v_{\text{out}} < V_{EE} \end{cases} \end{cases} \tag{6}$$

Although a number of approximations and simplifications were made, the resulting system is still quite complicated and ugly, with piecewise components and circular references. Fortunately, solving such a system numerically alleviates the majority of these issues.

2.4 Numerical Solution

Since v_{out} depends on v^+ and v^-, and similarly v^+ and v^- depend on v_{out}, an analytical solution is difficult. Using a numerical approach allows for a rather natural solution. The

voltage at each of op-amp inputs (v^+ and v^-) may be calculated using the values from the previous time step and then fed into the op-amp function for the current time step. This resolves the issue with the circular references complicating the equation, and should be accurate assuming a small enough time step is used.

Such a solution was implemented in Python using the `scipy.integrate.ode` module. When run, the following results (Figure 4) were produced with the constants shown in Table 1.

Table 1: Constant Parameters for Simulation

Constant	Value	
C	$1 \cdot 10^{-6}$	F
R_1	$100 \cdot 10^{3}$	Ω
R_2	$10 \cdot 10^{3}$	Ω
R_3	$10 \cdot 10^{3}$	Ω
Gain	100	dB
V_{CC}	5.0	V
V_{EE}	0.0	V
v_{ref}	2.5	V
t_0	0.0	sec
t_f	1.0	sec
Δt	0.001	sec

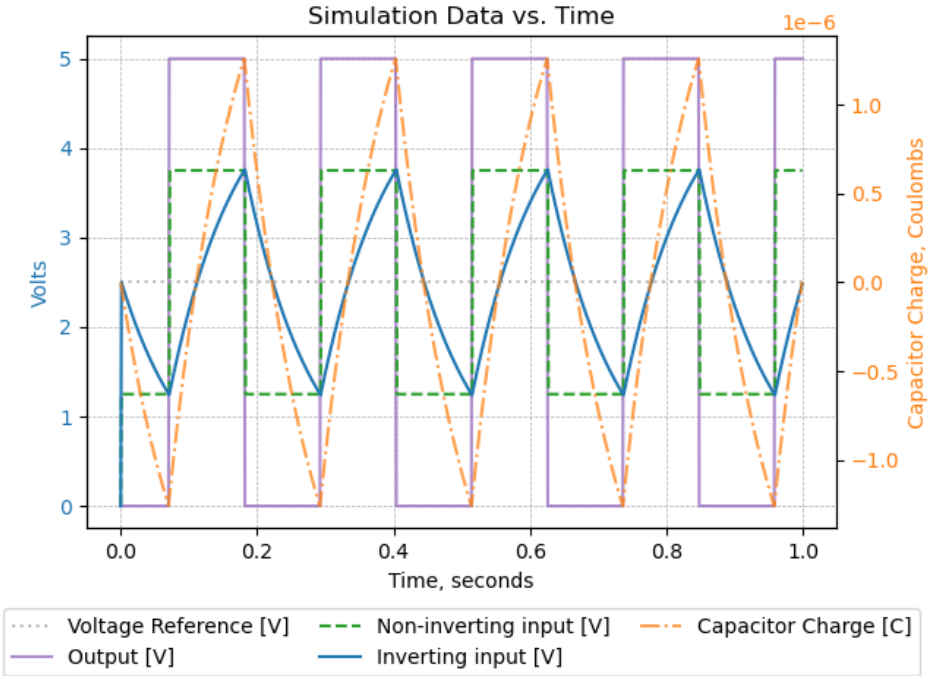


Figure 4: Simulation Results using Constants in Table 1

2.5 Analysis of Oscillation Parameters

The method of oscillation becomes apparent looking at Figure 4. The op-amp output is constantly saturated at either V_{CC} or V_{EE}, and the voltage divider creates a voltage in between v_{out} and v_{ref} at the non-inverting input. The voltage at the inverting input changes continually as the capacitor charges and discharges; the voltage it produces at the inverting input constantly chases the voltage at the non-inverting input due to the polarization of $v_{\text{out}} - v_{\text{ref}}$, and upon reaching the non-inverting input the output swaps and the process repeats in the opposite direction. Using this observation, it is possible to derive analytical solutions for the time behavior of the oscillator.

The initial step is to identify a solution to the differential equation in Equation 4. There is a well-known solution for an RC circuit and it is shown in Equation 7.

$$\frac{Q}{C} = v_{\text{capacitor}} = v_{\text{applied}} \left(1 - e^{-\frac{t}{R_1 C}}\right) \tag{7}$$
$$\text{where } v_{\text{applied}} = v_{\text{out}} - v_{\text{ref}}$$

To find the time elapsed for when the voltage output is high, consider a segment of time from when the voltage output just switched from low (V_{EE}) to high (V_{CC}). What needs to be found is the time taken for the inverting input voltage to charge from the non-inverting input voltage when the output is low (Equation 8) to the non-inverting input voltage when the voltage is high (Equation 9).

$$\begin{aligned}
v^-\big|_{t=t_s} &= v^+\big|_{t=t_s} \\
v_{\text{ref}} + \frac{Q}{C}\bigg|_{t=t_s} &= \frac{(v_{\text{out}} - v_{\text{ref}})\, R_2}{R_2 + R_3} + v_{\text{ref}}\bigg|_{t=t_s} \\
(v_{\text{out}} - v_{\text{ref}}) \left(1 - e^{-\frac{t}{R_1 C}}\right)\bigg|_{t=t_s} &= \frac{(v_{\text{out}} - v_{\text{ref}})\, R_2}{R_2 + R_3}\bigg|_{t=t_s} \\
(V_{CC} - v_{\text{ref}}) \left(1 - e^{-\frac{t_s}{R_1 C}}\right) &= \frac{(V_{EE} - v_{\text{ref}})\, R_2}{R_2 + R_3}
\end{aligned} \tag{8}$$

$$\begin{aligned}
v^-\big|_{t=t_f} &= v^+\big|_{t=t_f} \\
(V_{CC} - v_{\text{ref}}) \left(1 - e^{-\frac{t_f}{R_1 C}}\right) &= \frac{(V_{CC} - v_{\text{ref}})\, R_2}{R_2 + R_3}
\end{aligned} \tag{9}$$

The high output time period Δt_{high} is found from the difference $t_f - t_s$ (Equation 10).

$$
\begin{aligned}
(V_{CC} - v_{\text{ref}})\left(1 - e^{-\frac{t_s}{R_1 C}}\right) &= \frac{(V_{EE} - v_{\text{ref}})\,R_2}{R_2 + R_3} \\
1 - e^{-\frac{t_s}{R_1 C}} &= \frac{(V_{EE} - v_{\text{ref}})\,R_2}{(R_2 + R_3)\,(V_{CC} - v_{\text{ref}})} \\
e^{-\frac{t_s}{R_1 C}} &= 1 - \frac{(V_{EE} - v_{\text{ref}})\,R_2}{(R_2 + R_3)\,(V_{CC} - v_{\text{ref}})} \\
t_s &= -R_1 C \ln\left[1 - \frac{(V_{EE} - v_{\text{ref}})\,R_2}{(R_2 + R_3)\,(V_{CC} - v_{\text{ref}})}\right]
\end{aligned}
$$

$$
\begin{aligned}
t_f &= -R_1 C \ln\left[1 - \frac{(V_{CC} - v_{\text{ref}})\,R_2}{(R_2 + R_3)\,(V_{CC} - v_{\text{ref}})}\right] \\
&= -R_1 C \ln\left[1 - \frac{R_2}{R_2 + R_3}\right]
\end{aligned}
$$

$$
\Delta t_{\text{high}} = t_f - t_s = R_1 C \left(\ln\left[1 - \frac{(V_{EE} - v_{\text{ref}})\,R_2}{(R_2 + R_3)\,(V_{CC} - v_{\text{ref}})}\right] - \ln\left[1 - \frac{R_2}{R_2 + R_3}\right]\right) \tag{10}
$$

By a similar computation, the low output time period Δt_{low} may be found (Equation 11).

$$
\Delta t_{\text{low}} = R_1 C \left(\ln\left[1 - \frac{(V_{CC} - v_{\text{ref}})\,R_2}{(R_2 + R_3)\,(V_{EE} - v_{\text{ref}})}\right] - \ln\left[1 - \frac{R_2}{R_2 + R_3}\right]\right) \tag{11}
$$

If I place some constraints on the constant parameters of these equations, it is possible to solve for a desired $R_1 C$ time constant given a desired period or frequency. If I constrain $R_2 = R_3$ and that v_{ref} is between V_{CC} and V_{EE} such that $(V_{EE} - v_{\text{ref}}) = -(V_{CC} - v_{\text{ref}})$, the equation greatly simplifies yielding Equations 12 and 13.

$$
\Delta t_{\text{high}} = \Delta t_{\text{low}} = R_1 C \left(\ln\left[\frac{3}{2}\right] - \ln\left[\frac{1}{2}\right]\right) = R_1 C \ln 3
$$

$$
T = \Delta t_{\text{high}} + \Delta t_{\text{low}} = 2 R_1 C \ln 3 \quad \rightarrow \quad R_1 C = \frac{T}{2 \ln 3} \tag{12}
$$

$$
f = \frac{1}{T} = \frac{1}{2 R_1 C \ln 3} \quad \rightarrow \quad R_1 C = \frac{1}{2 f \ln 3} \tag{13}
$$

3 Program Details

The Python script I developed for this project comes with three main execution sections:

(i) Load in the simulation constant values.

(ii) Simulate the system using `scipy.integrate.ode`.

(iii) Output the simulation results and other calculated statistics.

3.1 Section (i) – Load Constants

All user input occurs in Section (i) with loading the simulation constants. There are three primary options for user input in this section:

1. Load default constant values.
2. Load constants from a `.json` file.
3. Load constants via manual user input.
   (a) Input all constants manually.
   (b) Calculate RC value given period or frequency.

Options 1 and 2 are similar as they both load the constants from a `.json` file. Option 2 allows the user to specify a path to a `.json` constants file (or just the name of the file if in the same directory as the scripts). The input is validated to make sure it is a `.json` file and has all of the necessary constant parameters (in a user-defined function). Option 1 simply bypasses the user input for the file path used in Option 2 and uses the `.json` defaults file that is specified in the code. Option 3 also uses the same defaults file as Option 1, but will allow the user to override the constant parameters with their own value. Any input for an entry that is left blank or invalid is left as the default in this case. Suboption 3a allows the user to input parameters for all of the constant values (see Table 1 for the constants used). Suboption 3b allows the user to specify a desired period or frequency and calculate the needed RC time constant as in Section 2.5 using Equation 12. The program then calculates the correct resistor and capacitor pairing given either a fixed resistance or capacitance. As noted in Section 2.5, certain assumptions must be made for this calculation to be performed, and as a result the program limits what other constant values may be defined in this case, with the rest being kept as the defaults. The permitted values are shown in Table 2.

Table 2: Allowed User-Changeable Constant Parameters for RC Time Constant Calculation

Constant	Units
Gain	dB
t_0	sec
t_f	sec
Δt	sec

At the end of this section the constant parameters are redisplayed for the user's reference.

3.2 Section (ii) – Run Simulation

In Section (ii), the simulation is prepared and run. A user-defined function is used to produce a function object that behaves as specified in Equation 1, and several `numpy` arrays are prepared to store the simulation data over time. The time steps array is made first using the `arange` function, and all other data series are made using `zeros_like` to match. The differential equation (Equation 4) is defined to take in the system state and output $\frac{dQ}{dt}$ in

accordance with what is required by `scipy.integrate.ode`, and an ODE solver object is created and initialized with the initial values to solve the IVP. The system is then run while successful until the simulation end time is reached using the equations in Equation System 6.

3.3 Section (iii) – Output Results

Finally in Section (iii) output is produced for the user. The oscillation parameters (frequency, period, duty cycle, high period, and low period) are calculated using Equations 10 and 11 in a user-defined function, then displayed to the console. A graph is then created using `matplotlib` to show the evolution of the system value over time. Lastly, the simulation values are exported to an `output.csv` file for future reference.

3.4 User-Defined Functions

The Python program makes use of five user-defined functions for various purposes. Their uses are briefly summarized in Table 3.

Table 3: User-Defined Functions Used in the Simulation Code

UDF Name (Arguments)	Returns	Summary
`validateJSONconstants(path)`	none	Checks a `.json` file specified to make sure it has the necessary parameters. Raises an exception to be handled if the file is missing parameters, passes otherwise.
`dBtoAmp(decibels)`	unitless amplitude	Converts decibels to amplitude as shown in Equation 2.
`makeOpAmpFunc(gain, vcc, vee)`	function object	Generates a function object that behaves like Equation 1.
`userInputCalculate(defaults_fpath)`	constants namespace `c`	Gets user input for simulation constant parameters or calculates parameters based on desired frequency/period.
`computeFrequencyParams(c)`	`tuple` of values	Determines the frequency, period, duty cycle, high period, and low period using Equations 10 and 11 based on the constant parameters in the namespace `c`.

In the main script a `dQ_dt(t, Q, consts, Vout)` function is defined. This is required for the ODE solver to function and is only used for that purpose.

3.5 Use of `lambda` Declaration

In Python the keyword `lambda` is used to declare anonymous function objects. It gets its name from *Lambda Calculus*, which is a mathematical system for expressing computation based on binding abstracted functions to variables. Unlike regular functions in Python which are created with the `def` keyword followed by a function name, `lambda` functions are never defined with a name, hence the term "anonymous". `lambda` functions are incredibly useful as they allow for the definition of simple functions on the fly that can be assigned to variables or passed into functions expecting a function input. The following listing demonstrates how a `lambda` was used to pass the constructor for `simpleNamespace` into the `.json` loader function. It specifies how the produced object (a `dict` type) should be fed into the constructor.

Example of a `lambda` Used to Pass a Function Into Another Function

```
        c = json.load(json_file_obj, object_hook = lambda d : SimpleNamespace(**d))
```

A `lambda` was also used in `makeOpAmpFunc` to create a function object.

Example of a `lambda` Used to Create an Anonymous Function

```
# Use a lambda to create a function that behaves like an op amp
def makeOpAmpFunc(gain, vcc, vee = 0):
    if vcc <= vee:
        raise ValueError("Vcc must have a higher potential than Vee")
    return lambda ninv, inv : (np.clip((gain * (ninv   inv) + (vcc + vee) / 2), vee, vcc))
```

They also saw use in the `userInputCalculate` function to help with unit conversion. By assigning the resultant function object to a variable I was able to make the code accept different input units depending on what the user selected but still produce the correct output without having to add extra logic to process the user's numerical input.

A `lambda` Used to Create Different Functional Behaviors Depending on User Input

```
        while True: # Determine whether or not user wants to calculate using frequency or period
            instr = input("Please select [f]requency or [p]eriod  > ").lower()
            if instr == 'f':
                prompt = "frequency [Hz]"
                convert = lambda inp : 1 / inp # Converter function from Hz to sec
                break
            elif instr == 'p':
                prompt = "period [s]"
                convert = lambda inp: inp      # Pass through function (sec  > sec)
                break
            print("Please enter 'f' or 'p' only.")
```

```
        while True: # Determine whether or not user wants R or C to be a known value
            instr = input("Please select whether to fix [r]esistance or [c]apacitance  > ").lower
                ()
```

```
if instr == 'r':
    prompt = "resistance [Ohms]"
    calculate = lambda R : (R, RC / R) # Takes R and makes a tuple (R, C) using RC
    break
elif instr == 'c':
    prompt = "capacitance [Farads]"
    calculate = lambda C : (RC / C, C) # Takes C and makes a tuple (R, C) using RC
    break
print("Please enter 'r' or 'c' only.")
```

4 Comparison to Actual Circuit

With access to an oscilloscope at home, I was able to compare my simulation results to empirical data. With the circuit constructed as in Figure 1, I connected it to an oscilloscope and digital power supply as shown in Figure 5.

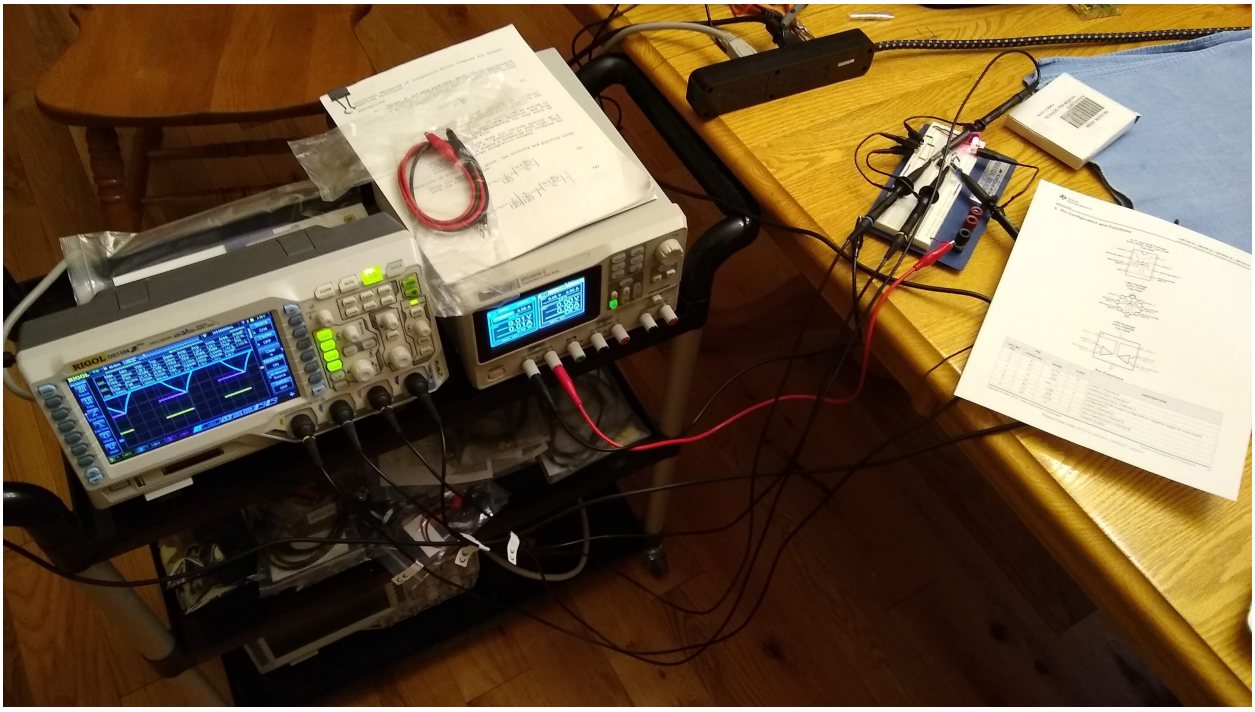
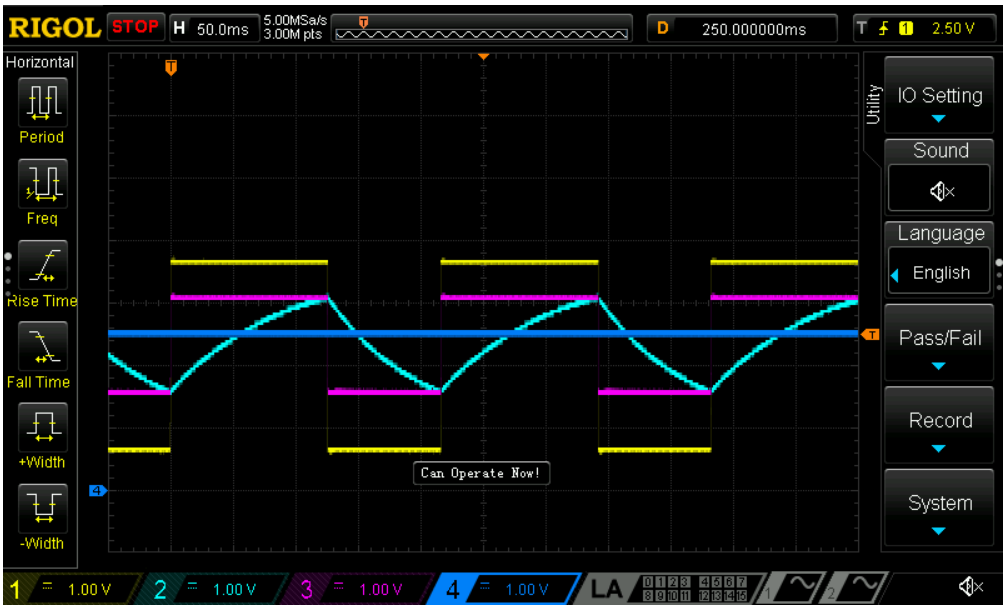
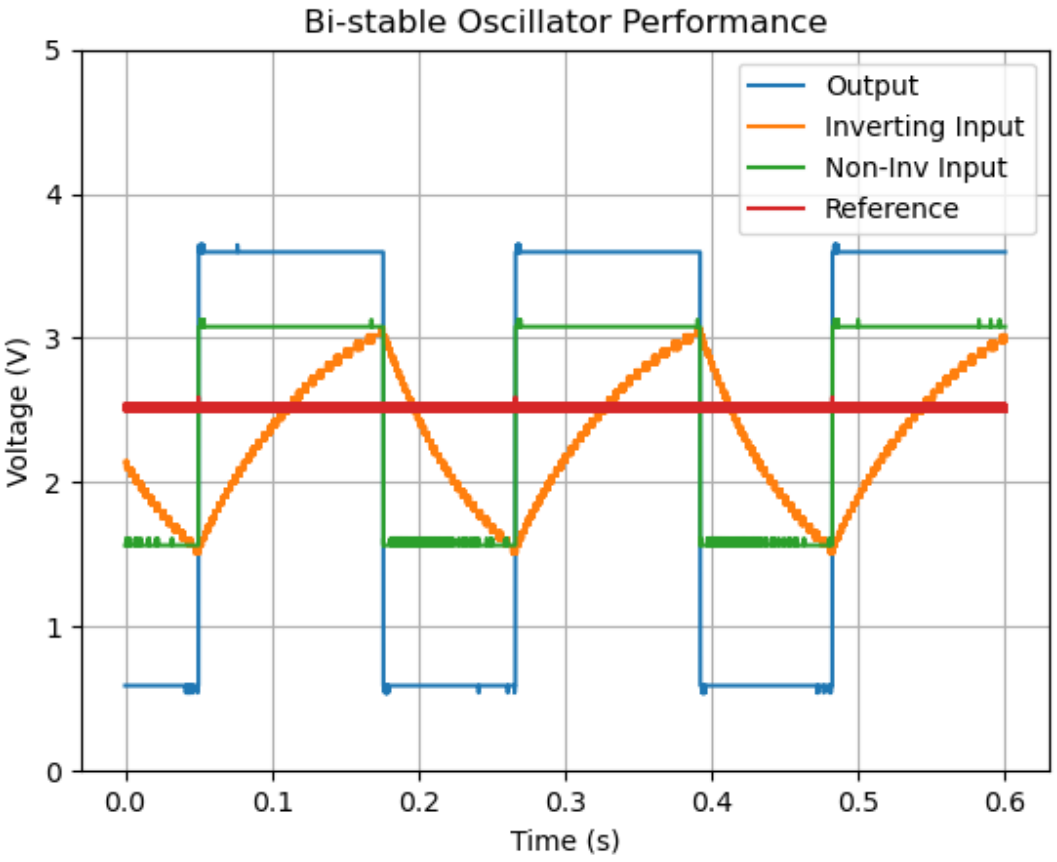


Figure 5: Oscillator Circuit as Connected to Oscilloscope

I performed data collection of the operation of the actual circuit, which revealed interesting results which deviated from my predictions. Graphs produced from the oscilloscope are shown in Figure 6. Components with values matching those specified in Table 1 were used; resistors were accurate to within 1% and an LM358-N operational amplifier was used.



(a) Screenshot Produced by Oscilloscope



(b) Graph Produced from Exported Data

Figure 6: Data Collected from the Oscilloscope

The resulting graphs look remarkably similar to what my simulation showed, however what was immediately obvious was that the output produced was not a square wave with a 50% duty cycle, and the frequency was off from what was expected. Table 4 shows a comparison between the expected and measured values.

Table 4: Expected and Measured Values of Circuit Using Constants in Table 1

	Expected	Measured
Frequency (Hz)	4.551	4.619
Period (sec)	0.220	0.217
Duty Cycle	50.0%	58.0%
High Period (sec)	0.110	0.126
Low Period (sec)	0.110	0.091
Output Max. (V)	5.0	3.621
Output Min. (V)	0.0	0.560
Oscillator Ref. (V)	2.5	2.541

After analyzing these data, I noticed that the outputs of the operational amplifier were not reaching $V_{CC} = 5.0\text{V}$ and $V_{EE} = 0.0\text{V}$ as expected, while v_{ref} was reasonably close to the expected 2.5V. Notably, the operational amplifier was reaching the saturation voltages different distances from the expected output on either end. The low output was only 0.56V too high, but the high output was nearly 1.38V too low. As a result, the differential voltage $v_{\text{out}} - v_{\text{ref}}$ applied to the RC circuit varied in magnitude between the high and low periods which threw off the periods of each oscillation cycle. Since the magnitude of the differential voltage applied is greater in the low state than the high state, the capacitor charges faster in this state and the period is shortened versus the high state.

This deviation from the expected results somewhat invalidates the simple model assumption (Equation 1) that I had used to make my simulation and predictions. The uneven output voltage offset from V_{CC} and V_{EE} are not something that my simulator addresses so the results produced are not completely accurate to reality. Furthermore, my attempts to find this phenomenon documented in the LM358-N's specification sheet did not turn up anything, so attempting to correct my model is not something I shall be able to do in the immediate future. However, recalling the assumptions made for calculations in Section 2.5, placing v_{ref} equidistant from the output voltages of the op-amp causes all terms relating to voltages in the circuit to disappear. Therefore, if v_{ref} could be adjusted to be in between the two output voltages, the results would return to the predicted values. This could be achieved by replacing the voltage divider connected to the left op-amp in Figure 2 with an adjustable resistive device such as a potentiometer.

5 Conclusion

I was able to produce a model of an operational amplifier-based oscillator circuit using Python and `scipy` that enabled me to understand its basis of operation by charging and discharging a capacitor. I then used that new understanding of the circuit's operation to

make predictions of the frequency and period of the signal the circuit would produce just by using the constant values of various circuit components, which I integrated into my program. These equations could also be used to back into the values needed for the circuit components to achieve a desired frequency or period after placing several key restraints on some of the constant values. Finally, I compared my predictions to measurements performed on an actual circuit, which revealed a major discrepancy between reality and the assumptions I had made about the behavior of an operational amplifier in regards to the saturation points of the output voltage, which influenced the oscillation frequency and duty cycle. After performing an analysis of the issue and comparing it to my equations, I came up with a potential solution to resolve the timing issues by using a potentiometer.

References

[1] Bitar S.J. N. Makarov S., Ludwig R. *Practical Electrical Engineering*. Springer, Cham, 2016.

Appendix

A User's Guide

The program is run by executing `ENGR133_Project_circuitSim_ekessel.py`. All three files (`ENGR133_Project_circuitSim_ekessel.py`, `ENGR133_Project_circuitSim_funcs_ekessel.py`, and `ENGR133_Project_circuitSim_defaults_ekessel.json`) must be in the same directory for the code to work. Upon running the program, you will be greeted with this prompt:

```
Please select a mode or 0 to exit:
 1) Load default values.
 2) Load from file.
 3) Enter manually or calculate.

Enter a mode ->
```

The options here are the same as described in Section 3.1. Invalid inputs will prompt the user to re-enter their input. Entering `0` causes the program to exit. Entering `1` has the program load the default values, while entering `2` has the program load the values from an external file as shown below.

```
Enter a mode -> 2

Enter the name/path to a .json constants file (or 'exit' to exit)
    ->
```

A path to a valid `.json` file must be entered or the program will re-prompt the user. Entering `exit` makes the program exit. A valid `.json` file will have the `.json` extension and contain all of the constants listed in Table 1. See the final listing in Appendix B for an example of how how a constants file should be configured.

Entering `3` brings the user to another set of prompts, shown below.

```
Enter a mode -> 3

Please select a mode or 0 to exit:
 1) Input circuit values.
 2) Calculate RC constant for frequency/period.

Enter a mode ->
```

Like previously, entering `0` exits. Entering `1` brings the user to the interface in the following listing where the user can enter any value for those listed in Table 1. Blank or invalid values are treated as implying the default for that constant, and no retries are given for bad entries, so be careful with entering your values here. It will also accept values given in scientific notation like `1.0e-6`.

```
Enter a mode -> 1
For each of the values below, enter a positive value or leave
   blank for the default:

 C    (Default =   1e-06 [Farads]) -->
 R1   (Default =  100000 [Ohms]) ----> 1000000
 R2   (Default =   10000 [Ohms]) ----> asdf
 R3   (Default =   10000 [Ohms]) ----> invalid inputs just
    default to the default
 GAIN (Default =     100 [dB]) ------>
 VCC  (Default =     5.0 [Volts]) ---> 9.0
 VEE  (Default =     0.0 [Volts]) --->
 VREF (Default =     2.5 [Volts]) ---> 4.5
 T_0  (Default =     0.0 [Seconds]) ->
 T_F  (Default =     1.0 [Seconds]) -> 2.5
 DT   (Default =   0.001 [Seconds]) -> 1e-4
```

Alternatively, if `2` is selected, the user is brought to an interface where the RC values are calculated based on a desired frequency or period. The user may specify a few constants as above, but will then be prompted to enter `f` for frequency or `p` for period, the value for their selection, then to enter `r` to fix resistance or `c` to fix capacitance, followed by the value for that selection. Invalid selections for the calculation modes will cause the prompt to be repeated.

```
Enter a mode -> 2
In order to calcuate for a specific frequency or period, the
   default values will be used for most constants.
For each of the values below, enter a positive value or leave
   blank for the default:

 GAIN (Default =     100 [dB]) ------>
 T_0  (Default =     0.0 [Seconds]) ->
 T_F  (Default =     1.0 [Seconds]) -> 10
 DT   (Default =   0.001 [Seconds]) ->

Please select [f]requency or [p]eriod -> p
Enter the desired period [s] -> 2.5
The needed RC time constant is 1.138 [Ohm-Farads].

Please select whether to fix [r]esistance or [c]apacitance -> c
Enter the desired capacitance [Farads] -> 1e-6
```

After successfully providing the constant values via any of the methods above, they are relayed back to the user and the simulation is run. The following listing shows the example output for the mode 3.1 inputs above.

```
The simulation constants are:
 C    =   1e-06 [Farads]
 R1   = 1000000.0 [Ohms]
 R2   =   10000 [Ohms]
 R3   =   10000 [Ohms]
 GAIN =     100 [dB]
 VCC  =     9.0 [Volts]
 VEE  =     0.0 [Volts]
 VREF =     4.5 [Volts]
 T_0  =     0.0 [Seconds]
 T_F  =     2.5 [Seconds]
 DT   =  0.0001 [Seconds]

Running Simulation: 100%|##########| 25000/25000 [00:01<00:00,
   17358.24it/s]
```

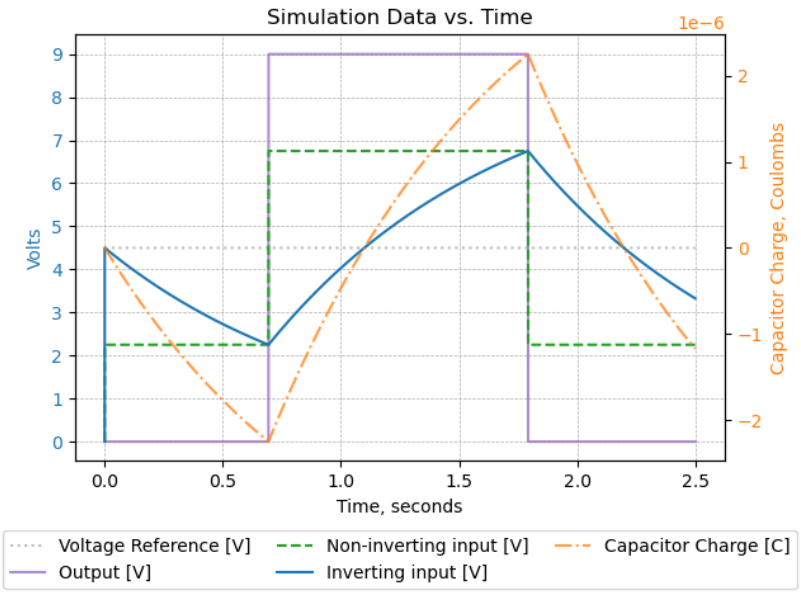
Similarly, for the mode 3.2 inputs above.

```
The simulation constants are:
 C    =   1e-06 [Farads]
 R1   = 1137799.0332835466 [Ohms]
 R2   =   10000 [Ohms]
 R3   =   10000 [Ohms]
 GAIN =     100 [dB]
 VCC  =     5.0 [Volts]
 VEE  =     0.0 [Volts]
 VREF =     2.5 [Volts]
 T_0  =     0.0 [Seconds]
 T_F  =    10.0 [Seconds]
 DT   =   0.001 [Seconds]

Running Simulation: 100%|##########| 10000/10000 [00:00<00:00,
   17788.22it/s]
```

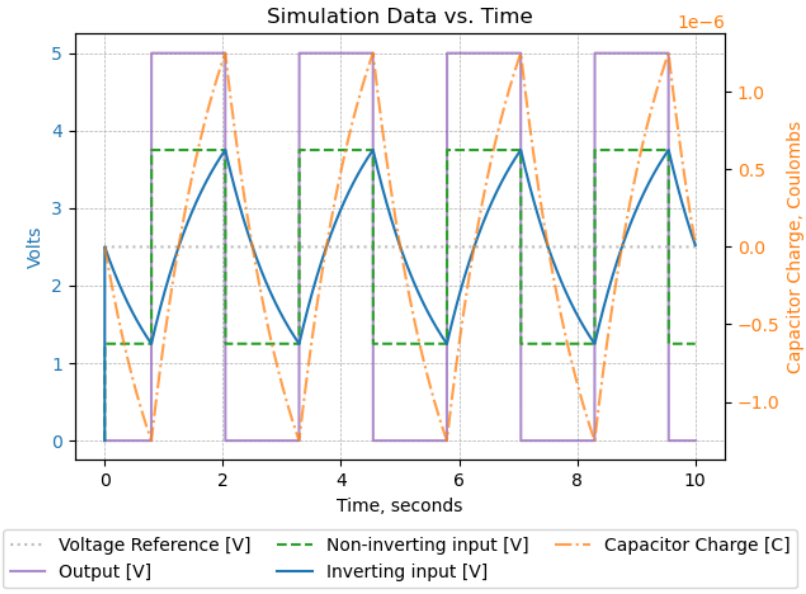
After the simulation is run, graphs are produced and an analysis of the oscillation characteristics is displayed to the console (Figure 7). The outputs for the example inputs shown for mode 3.1 are in Figure 7a, and the outputs for the mode 3.2 example are in Figure 7b.

(a) Simulation Results for Entry Mode 3.1 Example




```
The circuit oscillates with a frequency of 0.455 Hz (Period =
   2.197 s).
The duty cycle is 50.0% (1.099 s high / 1.099 s low).
```

(b) Simulation Results for Entry Mode 3.2 Example




```
The circuit oscillates with a frequency of 0.400 Hz (Period =
   2.500 s).
The duty cycle is 50.0% (1.250 s high / 1.250 s low).
```

Figure 7: Example Graph and Console Outputs

B Code

ENGR133_Project_circuitSim_ekessel.py

```
import numpy as np
from scipy.integrate import ode
import matplotlib.pyplot as plt

import os
import sys
import json
from types import SimpleNamespace # Container where values can be accessed like members
from tqdm import tqdm # Progress bar

from ENGR133_Project_circuitSim_funcs_ekessel import * # Import everything


''' ===== Data/File Loading Section ===== '''

filedir = os.path.dirname(os.path.abspath(__file__)) # Locate this file and save directory
DEFAULTS_FNAME = "ENGR133_Project_circuitSim_defaults_ekessel.json"

# Ask the use whether they want to load default data, load a different file, or input data manually
options = ["Load default values.", "Load from file.", "Enter manually or calculate."]
print("\nPlease select a mode or 0 to exit:")
for idx, option in enumerate(options):
    print(f" {idx + 1}) {option}")

while True: # Loop until valid input is given
    instr = input("Enter a mode  > ")
    try:
        inval = int(instr)
        if not (0 <= inval <= len(options)):
            raise ValueError(f"Invalid entry: Please enter a number 0 {len(options)}")
        break
    except ValueError as e:
        print(f"Error: {e}")

if inval == 0: # Exit
    print("Exiting...")
    sys.exit()
elif inval == 3: # Manual input, load using UDF
    c = userInputCalculate(os.path.join(filedir, DEFAULTS_FNAME))
else:
```

```
    if inval == 1: # Default file
        json_path = os.path.join(filedir, DEFAULTS_FNAME)

    else: # User input for filename
        while True: # Loop until valid input is given
            instr = input("Enter the name/path to a .json constants file (or 'exit' to exit)  > "
                )
            try:
                if instr == 'exit':
                    print("Exiting...")
                    sys.exit()

                path = os.path.abspath(instr) # Locate absolute location of specified path
                if not os.path.isfile(path):
                    raise ValueError("The path you specified does not lead to a file")
                if not path.lower().endswith(('.json')):
                    raise ValueError("The file you specified is not a .json file")

                validateJSONconstants(path)   # UDF to check file has everything needed

                json_path = path
                break

            except ValueError as e:
                print(f"Error: {e}")

    # We have a valid path now, load the constants
    with open(json_path) as json_file_obj:
        # Load json but wrapping it into a SimpleNamespace
        c = json.load(json_file_obj, object_hook = lambda d : SimpleNamespace(**d))

c.A = dBtoAmp(c.GAIN)   # Compute the amplification constant from the gain in dB

# Display the constants for the simulation that were just loaded
print("\nThe simulation constants are:")
for val_name in CONST_NAMES:
    print(f" {val_name:4} = {c.__getattribute__(val_name):7} [{CONST_NAMES[val_name]}]")


''' ===== Simulation Section ===== '''

# Differential equation function. Recieves t and a state vector Q and returns dQ/dt
def dQ_dt(t, Q, consts, Vout):
    return [((Vout   consts.VREF) /  consts.R1)   (Q[0] / (consts.R1 * consts.C))]

```

```
85  op_amp = makeOpAmpFunc(c.A, c.VCC, c.VEE)  # Make the op amp function
86
87  # Arrays to hold simulation data
88  time_steps_s = np.arange(c.T_0, c.T_F, c.DT)    # Time steps for the simulation
89  cap_charge_C = np.zeros_like(time_steps_s)  # Charge on the capacitor (Coulombs)
90  output_volts = np.zeros_like(time_steps_s)  # Output voltage of the op amp
91  invert_inp_V = np.zeros_like(time_steps_s)  # Inverting input voltage on the op amp
92  ninvrt_inp_V = np.zeros_like(time_steps_s)  # Non inverting input voltage on the op amp
93
94  reference_V = np.full_like(time_steps_s, c.VREF) # Used in plotting later
95
96  # Initialize the numerical integrator
97  rk4 = ode(dQ_dt).set_integrator('dopri5')   # Runge Kutta method of order (4)5
98  rk4.set_initial_value([cap_charge_C[0]], c.T_0) # Specify the IVP conditions
99
100 # Run the simulation
101 print(flush=True) # Flush the buffer before the progress bar starts
102 with tqdm(total=len(time_steps_s), desc="Running Simulation") as pbar: # Create a progress bar
        manually
103     ts_index = 1 # Current time step index
104     pbar.update() # Since we start one time step in
105
106     while (rk4.successful() and rk4.t <= c.T_F and ts_index < len(time_steps_s)):
107         # Multiple checks to verify validity of integration process
108
109         # Calculate voltages on op amp inputs (using last voltage) and op amp output
110         invert_inp_V[ts_index] = c.VREF + (cap_charge_C[ts_index   1] / c.C)
111         ninvrt_inp_V[ts_index] = c.VREF + ((output_volts[ts_index   1]   c.VREF) * c.R2 / (c.R2 +
                c.R3))
112         output_volts[ts_index] = op_amp(ninvrt_inp_V[ts_index], invert_inp_V[ts_index])
113
114         rk4.set_f_params(c, output_volts[ts_index]) # Set parameters for the differential eq.
115         rk4.integrate(time_steps_s[ts_index]) # Run the numerical integration
116         cap_charge_C[ts_index] = rk4.y[0]   # Extract capacitor charge from the integrator
117
118         ts_index += 1   # Move on to the next time step
119         pbar.update()
120 print(flush=True) # Flush the buffer after completing too
121
122
123 ''' ===== Plot/Output Section ===== '''
124
125 # Compute information about the oscillation charactaristics of the circuit
126 f_Hz, T_s, duty_cycle, t_high, t_low = computeFrequencyParams(c)
127
```

```
print(f"\nThe circuit oscillates with a frequency of {f_Hz:.3f} Hz (Period = {T_s:.3f} s).")
print(f"The duty cycle is {duty_cycle:.1%} ({t_high:.3f} s high / {t_low:.3f} s low).")

plt.ion()   # Enable interactive mode
fig1 = plt.figure() # Create a graph figure for manual control

ax1 = fig1.add_axes((0.1, 0.22, 0.8, 0.7)) # Space for the legend, etc

# Plot the values on the left voltage axis
ax1.set_xlabel('Time, seconds')
ax1.set_ylabel('Volts', color='tab:blue')
ax1.plot(time_steps_s, reference_V, ':', label='Voltage Reference [V]', color='tab:gray', alpha
    =0.5)
ax1.plot(time_steps_s, output_volts, ' ', label='Output [V]', color='tab:purple', alpha=0.75)
ax1.plot(time_steps_s, ninvrt_inp_V, '  ', label='Non inverting input [V]', color='tab:green')
ax1.plot(time_steps_s, invert_inp_V, ' ', label='Inverting input [V]', color='tab:blue')
plt.yticks(np.linspace(np.floor(c.VEE), np.ceil(c.VCC), int(np.ceil(c.VCC)   np.floor(c.VEE)) +
    1))
plt.grid(ls='  ', lw=0.5)
ax1.tick_params(axis='y', labelcolor='tab:blue')

ax2 = ax1.twinx() # Create graph with twinned x axis

# Plot the charge on the capacitor on the right axis
ax2.set_ylabel('Capacitor Charge, Coulombs', color='tab:orange')
ax2.plot(time_steps_s, cap_charge_C, ' .', label='Capacitor Charge [C]', color='tab:orange',
    alpha=0.75)
ax2.tick_params(axis='y', labelcolor='tab:orange')

plt.title("Simulation Data vs. Time")

# Add a legend to the bottom of the figure using bbox_to_anchor
fig1.legend(ncol=3, loc="upper right", bbox_to_anchor=(0., 0.02, 1., 0.1), mode="expand")

plt.show()


''' ===== Data File Output Section ===== '''

# Columnate data into a single numpy array for CSV output
columnated = np.column_stack((time_steps_s, reference_V, output_volts,
                              ninvrt_inp_V, invert_inp_V, cap_charge_C))
np.savetxt('output.csv', columnated, delimiter=', ', comments='', fmt='%e',
           header='Time [s], Reference [V], Output [V], Non Inverting Input [V], Inverting Input
               [V], Capacitor Charge [C]',)
```

ENGR133_Project_circuitSim_funcs_ekessel.py

```python
import sys
import numpy as np
import json
from types import SimpleNamespace # Container where values can be accessed like members

# Expected constant parameters and their units
CONST_NAMES = {'C':'Farads',
               'R1':'Ohms',
               'R2':'Ohms',
               'R3':'Ohms',
               'GAIN':'dB',
               'VCC':'Volts',
               'VEE':'Volts',
               'VREF':'Volts',
               'T_0':'Seconds',
               'T_F':'Seconds',
               'DT':'Seconds'}

# Validate the contents of a JSON file
def validateJSONconstants(path):
    with open(path) as fileobj:
        json_file = json.load(fileobj)
        # Use generator to make an iterable that is all true iff the json file has the needed values
        if not all(key in json_file for key in CONST_NAMES):
            # Raise an error to be handled
            raise ValueError("JSON file missing neccesary parameters")

# Convert decibels to unitless voltage amplification
def dBtoAmp(decibels): return (10 ** (decibels / 20))

# Use a lambda to create a function that behaves like an op amp
def makeOpAmpFunc(gain, vcc, vee = 0):
    if vcc <= vee:
        raise ValueError("Vcc must have a higher potential than Vee")
    return lambda ninv, inv : (np.clip((gain * (ninv   inv) + (vcc + vee) / 2), vee, vcc))

# Input values and calculate values
def userInputCalculate(defaults_fpath):
    with open(defaults_fpath) as fileobj: # Pre load defaults
        c = json.load(fileobj, object_hook = lambda d : SimpleNamespace(**d))

```

```
    # Ask the user to pick between complete manual input or RC const calculation
    modes = ["Input circuit values.", "Calculate RC constant for frequency/period."]
    print("\nPlease select a mode or 0 to exit:")
    for idx, option in enumerate(modes):
        print(f" {idx + 1}) {option}")

    while True:
        instr = input("Enter a mode  > ")
        try:
            inval = int(instr)
            if not (0 <= inval <= len(modes)):
                raise ValueError(f"Invalid entry: Please enter a number 0 {len(modes)}")
            break
        except ValueError as e:
            print(f"Error: {e}")

    if inval == 0: # Exit
        print("Exiting...")
        sys.exit()
    elif inval == 1: # Enter values manually
        print("For each of the values below, enter a positive value or leave blank for the
            default:")
        for val_name in CONST_NAMES:
            unit_name_len = len(CONST_NAMES[val_name]) # For making it look good
            instr = input(f" {val_name:<4} (Default = {c.__getattribute__(val_name):7} [{
                CONST_NAMES[val_name]}]) {(8   unit_name_len) * ' '}> ")
            try:
                new_val = float(instr)
                c.__setattr__(val_name, new_val) # Set an attribute using its name as a string
            except ValueError:
                pass # Just ignore bad inputs
    else:   # Compute RC values based on oscilation period/frequency
        print("In order to calcuate for a specific frequency or period, the default values will
            be used for most constants.")
        valid_cnames = ['GAIN', 'T_0', 'T_F', 'DT'] # Constants that user may adjust
        print("For each of the values below, enter a positive value or leave blank for the
            default:")
        for val_name in valid_cnames: # Same as above but for different constant list
            unit_name_len = len(CONST_NAMES[val_name]) # For making it look good
            instr = input(f" {val_name:<4} (Default = {c.__getattribute__(val_name):7} [{
                CONST_NAMES[val_name]}]) {(8   unit_name_len) * ' '}> ")
            try:
                new_val = float(instr)
                c.__setattr__(val_name, new_val)
            except ValueError:
```

```
82                  pass
83
84          while True: # Determine whether or not user wants to calculate using frequency or period
85              instr = input("Please select [f]requency or [p]eriod  > ").lower()
86              if instr == 'f':
87                  prompt = "frequency [Hz]"
88                  convert = lambda inp : 1 / inp # Converter function from Hz to sec
89                  break
90              elif instr == 'p':
91                  prompt = "period [s]"
92                  convert = lambda inp: inp      # Pass through function (sec  > sec)
93                  break
94              print("Please enter 'f' or 'p' only.")
95
96          while True: # Get desired constant parameter from user
97              instr = input(f"Enter the desired {prompt}  > ")
98              try:
99                  inval = float(instr)
100                 if inval <= 0:
101                     raise ValueError("Value must be positive")
102                 period = convert(inval) # Store as period using converter function
103                 break
104             except ValueError as e:
105                 print(f"Error: {e}")
106
107         RC = period / (2 * np.log(3)) # Calculate needed RC constant (easy w/ assumptions)
108         print(f"The needed RC time constant is {RC:.3f} [Ohm Farads].")
109
110         while True: # Determine whether or not user wants R or C to be a known value
111             instr = input("Please select whether to fix [r]esistance or [c]apacitance  > ").lower
                ()
112             if instr == 'r':
113                 prompt = "resistance [Ohms]"
114                 calculate = lambda R : (R, RC / R) # Takes R and makes a tuple (R, C) using RC
115                 break
116             elif instr == 'c':
117                 prompt = "capacitance [Farads]"
118                 calculate = lambda C : (RC / C, C) # Takes C and makes a tuple (R, C) using RC
119                 break
120             print("Please enter 'r' or 'c' only.")
121
122         while True: # Get the desired constant from the user
123             instr = input(f"Enter the desired {prompt}  > ")
124             try:
125                 inval = float(instr)
```

```
126                 if inval <= 0:
127                     raise ValueError("Value must be positive")
128                 R, C = calculate(inval) # Correctly calculate R and C from the input and RC
129                 break
130             except ValueError as e:
131                 print(f"Error: {e}")
132 
133         c.C = C     # Update the constant params
134         c.R1 = R
135 
136     return c # Completed constant values
137 
138 # Compute information about the oscillation of the circuit based on the constants
139 def computeFrequencyParams(c):
140     # Lots of math... see report for explanation
141     t_high = c.R1 * c.C * (np.log(1   (((c.VEE   c.VREF) * c.R2) /
142                                   ((c.R2 + c.R3) * (c.VCC   c.VREF))))
143                            np.log(1   (c.R2 / (c.R2 + c.R3))))
144     t_low  = c.R1 * c.C * (np.log(1   (((c.VCC   c.VREF) * c.R2) /
145                                   ((c.R2 + c.R3) * (c.VEE   c.VREF))))
146                            np.log(1   (c.R2 / (c.R2 + c.R3))))
147     period = t_high + t_low
148     frequency = 1 / period
149     duty_cycle = t_high / period
150 
151     return frequency, period, duty_cycle, t_high, t_low
```

ENGR133_Project_circuitSim_defaults_ekessel.json

```
{
"C":     0.000001,
"R1":    100000,
"R2":    10000,
"R3":    10000,
"GAIN":  100,
"VCC":   5.0,
"VEE":   0.0,
"VREF":  2.5,
"T_0":   0.0,
"T_F":   1.0,
"DT":    0.001
}
```