PHYC40600 - Physics with Astronomy and Space Science Lab 2;

555 Timer and Raspberry Pi Pico Pulse-Width Modulation

Daragh Hollman

daragh.hollman@ucdconnect.ie

(Dated: March 9, 2024)

The aims of this report were to explore pulse width modulation (PWM) and its applications through the simulation and construction of several example circuits. These included brightness variations of an LED, a simple digital-to-analogue converter. The 555 timer was also investigated both in PWM and in its application in a time-to-amplitude converter circuit. We attempted to construct this circuit physically, but encountered issues with op-amp integration which could not be resolved due to time constraints.

I. INTRODUCTION

Pulse width modulation (PWM) is a useful technique for controlling analog circuits digitally. Through the use of high-resolution internal counters, the duty cycle (the ratio of time the signal is on compared to off) of a square wave can be modulated to reduce the average power supplied to a load to a specific analog signal level [1]. The result is an analogue signal with amplitude at any given time proportional to the width of the pulse. An example PWM signal is shown in figure 1 for varying duty cycle. This switching is usually done at high frequencies so no discernible flickering is observed in the power delivery [2].

Applications of PWM begin with simply adjusting the brightness of lights but extend to less obvious use cases such as in the motor power regulation of light rail (such as the LUAS in Dublin [2]) and in AM radio transmission [3].
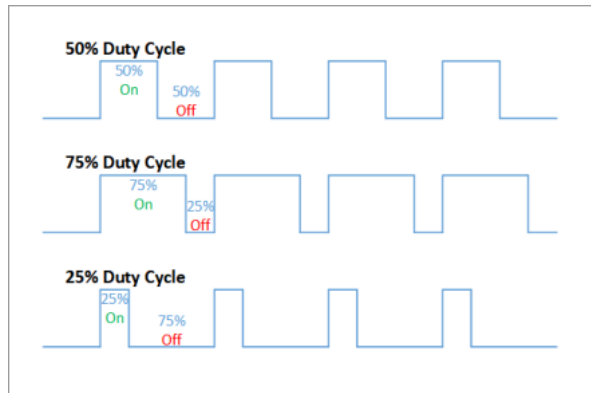

FIG. 1: Examples of difference duty cycles for a square wave [2].

II. PWM WITH A RASPBERRY PI PICO

The Raspberry Pi Pico is a high-performance microcontroller with multi-function digital I/O pins [2, 4]. PWM is one of these functions. A pin diagram along with full documentation can be found at https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html [4]. For ease of reading, this pin diagram is also included in appendix A.

The Pi Pico can be programmed using an implementation of Python 3 designed to run optimally on microcontrollers and other small or otherwise constrained environments called MicroPython [5]. MicroPython contains the MACHINE library from which the PWM function can be used to generate a PWM output on a designated pin. The frequency of the PWM can be set up to frequencies beyond 1 MHz, however in this lab we only use up to a maximum 100 kHz [2]. The duty cycle can be set to an integer value between 0 and 65535 (the 16-bit binary maximum), with the ratio of this value and the maximum yielding the ratio the signal will be on compared to off.

In this section, we first verify the PWM generation from the Pi Pico behaves as expected, and then design scripts to adjust the brightness of the inbuilt LED on the microcontroller. Following this, the combination of the PWM output and a low pass filter will be used to create a simple Digital-To-Analogue Converter (DAC). Lastly, the generation of analogue signals will be described and demonstrated for a triangle and sine wave.

A. PWM Basics and Applications to Varying LED Brightness

To first verify the PWM output from the Pi Pico, a simple script was written to output PWM for a constant frequency and duty. The output of this pin was measured with an oscilloscope and recorded for
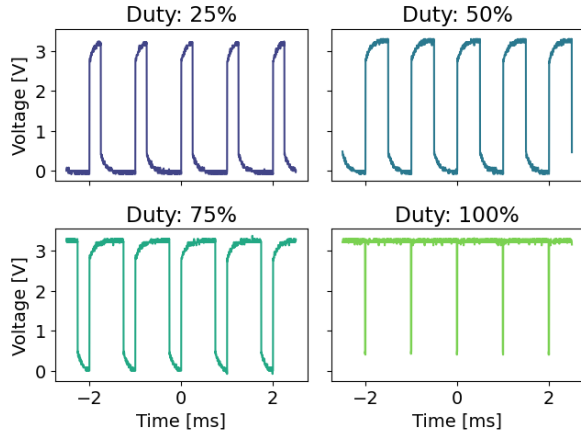
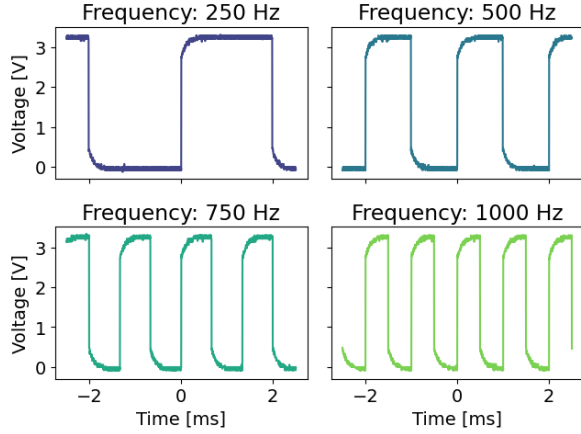**FIG. 2:** PWM output from the Pi Pico showing variations in duty for a constant frequency of 1000 Hz

FIG. 3: PWM output from the Pi Pico showing variations in frequency for a constant duty of $\approx 50\%$ (i.e. $65535/2 \approx 32768$, rounding to an integer value.)

several duty values with a constant frequency, and also for several frequency values for a constant duty value. Variations in duty are plotted in figure 2, and variations in frequency are plotted in figure 3. We can see that the PWM output is what is expected by comparison with figure 1. Frequency adjusts the frequency of the square wave output as a whole, while duty adjusts the amount the square wave is on compared to off.

Instead of passing the PWM output to an oscilloscope, the output was sent to internal LED on the Pi Pico (GPIO pin 25, see pin diagram). A *brightness value* float between 0 (off) and 1 (full brightness) was defined in the script mentioned previously. The simple multiplication of this brightness value with the maximum duty value 65535 is sufficient to control the brightness of the LED from this variable. Note that the duty value must be rounded to an integer as this floating point multiplication will result in decimal values which the Pi Pico cannot receive. For low PWM frequencies, flickering of the LED could be observed (as briefly mentioned in the

introduction). A 2014 study report that the minimum viewing time required for visual comprehension could be as low as 13 ms per frame in a sequence [6]. This corresponds to a minimum frequency 75 Hz. This value is decisively on the upper end of human capabilities (and is described as such in the study which quotes findings between 13 ms and 80 ms), given a common monitor refresh rate of 60 Hz or higher for modern devices [7]. To avoid visible flickering in our LED, a higher frequency of 100 Hz was chosen.

From here, it is easy to adjust this brightness value over time to linearly transition between 0 and 1. A loop was created to increase the brightness in steps of 0.01 for 100 steps, and then decrease again by the same value for a total length of 200 steps. In each step, a small delay $\mathcal{O}(\text{ms})$ could be applied to define the length of time to cycle through the loop. This delay t_{delay} was defined as follows:

$$t_{\text{delay}} = \text{round}_{\text{ms}} \left(\frac{\text{period}}{\text{N}} \right) \tag{1}$$

where the period is the length of time for one full cycle, and N the number of steps in the cycle, all rounded to the nearest millisecond.

Observing the LED it is clear that while the brightness value is being incremented linearly, the light does not appear to change linearly in brightness. It appears to stay brighter for longer than it is dimmer. This is however expected, as we know that the human eye has a logarithmic response to changes in light intensity. This phenomenon is part of what is known as the Weber-Fechner Laws [8]. We can adjust for this, by increasing and decreasing the brightness linearly in log-space. This was not fully achieved for this exercise but a close approximation was implemented which was functionally similar. The brightness was increased with the following equation:

$$\frac{10^t}{10} \tag{2}$$

and decreased with:

$$\frac{10^{-(t+1)}}{10} \tag{3}$$

where t is the position along the respective half of the 200 step cycle from 0 to 1, i.e. $i/100$ where i is the step number. While effective in producing a increase linear in log space, these equations do have the drawback of their bounds. The minimum value, corresponding to $t = 0$ for each is only at 10% of the maximum brightness, and as such the LED will never reach the fully off state. Despite the constraints on the range, the LED was observed to range between the two brightness values linearly (to the eye). The script for producing both of these effects is included in appendix B.

B. Simple DAC

A digital-to-analogue converter (DAC) can be created by passing the PWM output to a low pass filter. The low pass filter acts to attenuate the frequency component of the signal to leave solely the averaged PWM signal as the analogue output [2]. The diagram for such a setup is shown in figure 4 [2]. In a low pass filter the resistance R and capacitance C define the cut-off frequency f_c by the following relation [9]:

$$f_c = \frac{1}{2\pi RC} \tag{4}$$

At this cut-off frequency, a signal with that frequency has attenuated by $-3\,\mathrm{dB}$, and further frequencies beyond this cut-off will attenuate at a rate of $-20\,\mathrm{dB}$ per decade. PWM inputs with signal frequencies below this cut-off frequency will be mostly unattenuated and the output will not be smooth. It is then difficult to use high PWM frequencies as the input for the DAC, as larger resistors and capacitors will be needed to increase the cut-off frequency to provide the same attenuating effect.

Given a test input PWM signal of $20\,\mathrm{kHz}$ and a 50% duty, a combination of R and C was found to produce a signal with less than 5% variation. To achieve this variation, we must first calculate the voltage across the capacitor as it charges as follows:

$$V_{\mathrm{capacitor}} = V_{\mathrm{source}} \left(1 - e^{\frac{-t}{RC}}\right) \tag{5}$$

where t is the time spent charging with V_{source} applied. From here we can solve for a value of RC required. The duty at 50% will yield an analogue $V_{\mathrm{analogue}} = 1.65\,\mathrm{V}$ (half $V_{\mathrm{source}} = 3.3\,\mathrm{V}$), and for this to vary by 5%, we require:

$$V_{\mathrm{capacitor}} = 1.05 \times 1.65\,\mathrm{V} \tag{6}$$

With a charging time per period of $t = \frac{1}{2}\mathcal{T} = 2.5 \times 10^{-5}\,\mathrm{s}$ (for PWM period $\mathcal{T}$), we hence have:

$$V_{\mathrm{capacitor}} = \frac{1.05 \times 1.65\,\mathrm{V}}{3.3\,\mathrm{V}} = \left(1 - e^{\frac{-t}{RC}}\right) \tag{7}$$

$$0.525 = 1 - e^{\frac{-t}{RC}}$$

$$\frac{t}{RC} = -\ln(0.475)$$

$$RC = \frac{t}{0.744} = 3.36 \times 10^{-5}\,\mathrm{s} \tag{8}$$

While any combination of resistance and capacitance with $RC \geq 3.36 \times 10^{-5}\,\mathrm{s}$ will achieve this, we selected a combination of $R = 680\,\Omega$ and $C = 1\,\mu\mathrm{F}$, yielding $RC = 6.8 \times 10^{-4}\,\mathrm{s}$ was chosen based on the components available. This output was verified to have less than 5% variance using the oscilloscope and the results were plotted in figure 5.
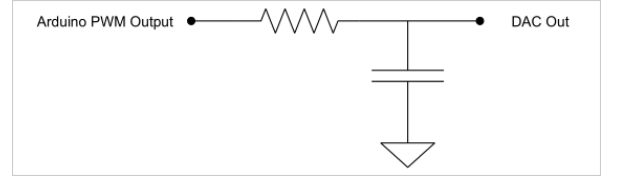

FIG. 4: A low pass filter to convert the PWM input to an analogue output [2]. Note a Pi Pico is used in our case instead of an Arduino, however the principles remain the same.
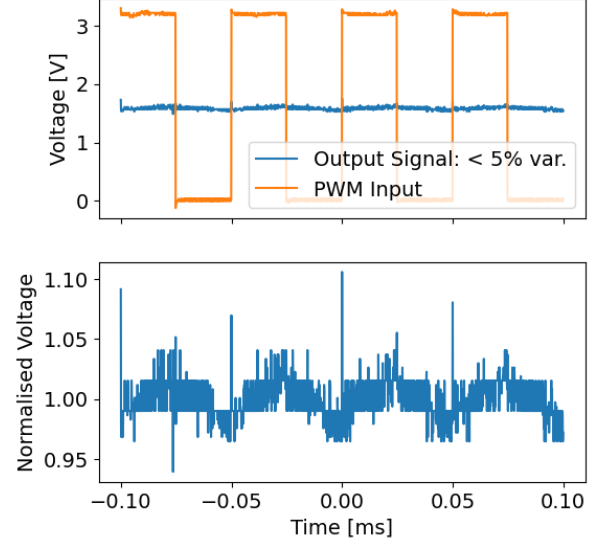

FIG. 5: The output from the simple DAC with a PWM (orange) input of $20\,\mathrm{kHz}$ and $R = 680\,\Omega$, $C = 1\,\mu\mathrm{F}$ to produce an output signal (blue) with variation less than 5%. *Second panel*: The DAC output normalised to the mean output.

It is important that this circuit not be used unbuffered. The output signal from the DAC circuit when connected to another circuit will affect the total resistance of the DAC circuit. The change in RC will cause the cut-off frequency to change, producing an unwanted response from the DAC. This can be amended using a unity gain op-amp to buffer the output signal. A unity gain op-amp is an op-amp circuit which has a voltage gain of 1, meaning the input and output signals are equal voltage [10]. However, one useful property of op-amps for this application is that they typically have very high impedence, and hence, will not draw a significant current from the DAC circuit. The total resistance and hence the RC value will remain unchanged [10].

C. Generating Analogue Output Functions

The combination of the above DAC circuit (to provide an analogue output) and varying the PWM duty cycle (to set the analogue DC level) allows for the creation of specific functions such as a triangular and sine function. This is possible through changing the PWM duty in discrete steps over a cycle [2]. A script was written

to vary the duty similarly to section II.A., instead this time adjusting the duty according to a custom function for a triangular signal or a sine wave. The full script is available in appendix C.

1. Triangular Signal

A triangular signal was generated through the use of a piecewise function:

$$T(i) = \begin{cases} \frac{2i}{N} & i < \frac{N}{2} \\ -\frac{2i}{N} + 2 & i \geq \frac{N}{2} \end{cases}$$

where i is the step along the function with N steps. The period of the signal was set by applying a short delay along each step of the cycle. This delay was defined as a function of the desired as follows:

$$t_{\text{delay}} = \frac{\text{period}}{N} \tag{9}$$

Triangle functions for periods of 100 ms and 50 ms were plotted in figure 6. We notice that these functions don't peak sharply as expected from the function. Through several observations with varying resistances, we find that higher resistance produces sharper peaks, with the downside of producing more noise on the voltage. For producing these signal functions, we used a 1 kΩ resistor as it provided good signal clarity whilst maintaining minimal noise.

2. Sine Wave

Sine wave signals were generated similarly, through the use of the following function:

$$S(i) = \frac{1}{2}\left(\sin\left(2\pi\frac{i}{N}\right) + 1\right) \tag{10}$$

with a frequency set similarly using a delay per time step. Sin functions for frequencies ranging between 10 Hz and 100 Hz were plotted in figure 7. We notice how the amplitude of each sin wave decreases as the frequency increases, and attempting to generate higher frequency sine waves results in large attenuation. This is expected behaviour, as we expect the low pass filter to attenuate high frequency changes in the input signal. 10 sin waves were generated with frequencies between 1 Hz to 1 kHz. The amplitude of each sine wave was determined by subtracting the mean value of each wave from the maximum. These values were plotted against their frequency in decibels with respect to the voltage output of the Pi Pico (3.3 V). See figure 8. We see the expected response from the low pass filter, with rapid attenuation after the cut-off frequency. This frequency was $f_c = 15.915$ Hz and is plotted as an orange vertical line through the data. Where it intersects the data is approximately -1 dB, far

from the expected value of -3 dB. It is clear that the actual cut-off frequency of this circuit must be higher than calculated, potentially due to the use of an unbuffered output affecting the resistance as discussed previously. Tolerance on the components could also play a role but cannot account for the full discrepancy. However regardless, it becomes obvious why high frequency sin waves are being attenuated with this RC configuration.

3. Inaccuracies and Accounting for Processing Time

There is an obvious trade-off between increasing the number of steps in the function and time errors in the output. Increasing the number of steps in the function cycle of course increases the accuracy of the output (purely by simply having more data points to plot over the period), however, as Python is by no stretch a slow programming language, we expect delays in the signal output for each step, causing an increase in the overall length of the period. In generating the above functions we initially observed larger periods ($\mathcal{O}(10\,\text{ms})$) than expected. We attempted to ameliorate these errors with a few methods.

While MicroPython contains functions for measuring the time between two points in the code, unfortunately these are not applicable for these purposes as - despite running relatively quickly - take time themselves to run. Hence, they account for the time taken to run the other commands in the loop, but the time they take to run cannot be accounted for.

Another attempt to correct for this issue was to use inbuilt MicroPython function decorators (NATIVE and VIPER). These cause the MicroPython compiler to send to the microcontroller native CPU opcodes in the place of bytecode. These decorators induce limitations on the possible code but generally require no adaptation to the functions [5]. The MicroPython documentation suggests these both offer performance increases twice as fast as standard. We were able to implement these decorators however they did not provide improvements significant enough to remove the issue. Difficulties were also encountered with respect to soft restarting the script. We believe the while loop used in the script was not able to be interrupted when converted using these decorators. Fortunately, simply de-powering the Pi Pico by unplugging it fixed these issues.

Other possible implementations include the use of another programming language - C and C++ are approximately 200 times faster than Python in most operations [2] - or perhaps the use of third party tools to compile Python to C, see Cython: `https://cython.org/`.

The method eventually implemented in the final version of the script was simply a manual correction by vi-
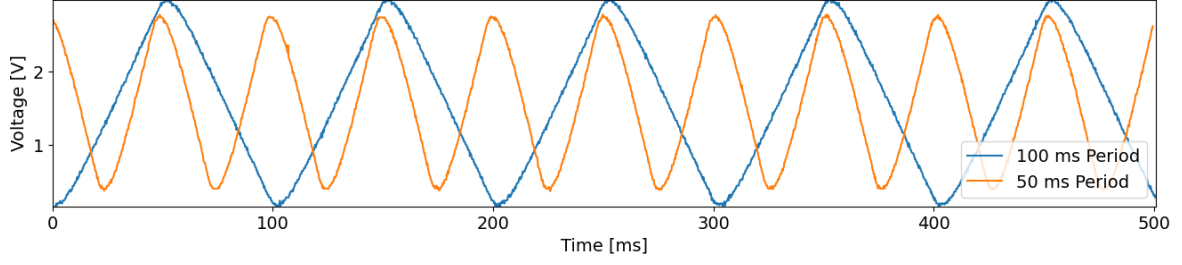
FIG. 6: Two triangle functions generated using the combination of PWM and a Low Pass Filter. The period was varied by making a time delay in updating each step of the curve. We note the difficulty in achieving a sharp peak on this function. Higher resistance values in the RC circuit produced sharper triangles, but also resulted in noisier signals.
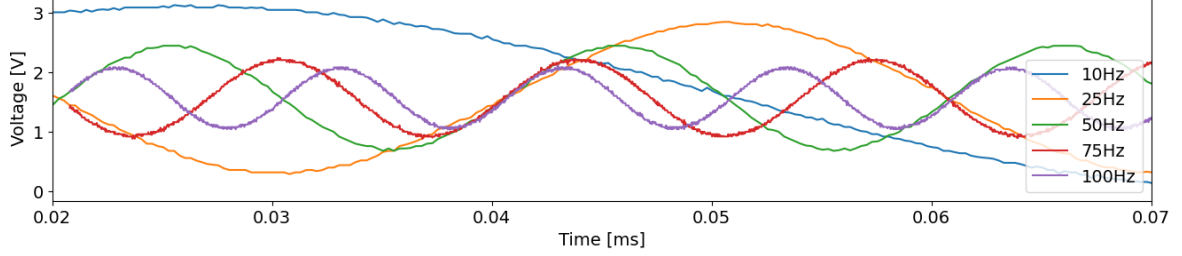

FIG. 7: Several sin functions generated using the combination of PWM and a Low Pass Filter. The frequency was varied with the same method as in the triangle function case.
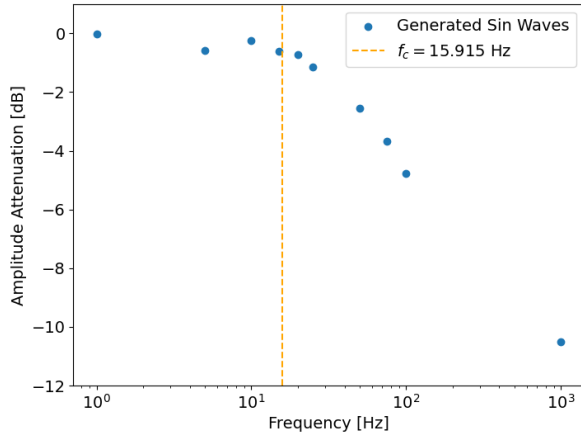

FIG. 8: The amplitude of the output sin wave plotted with respect to the frequency. We see the typical attenuation curve we expect from a low pass filter. Cut-off frequency (for $R = 1\,\mathrm{k\Omega}$ and $C = 10\,\mu\mathrm{F}$) is marked by the vertical line.

N	t_c
10	900
15	400
20	250
25	150
100	30
500	25
1000	25

TABLE I: Table of time delay correction values for function generation using the DAC. N is the number of steps in the function cycle, and t_c is the time correction subtracted from the delay in each step.

sual observation with an oscilloscope. While somewhat crude, it worked effectively and with relatively small errors. As the delay is applied on each step of the function cycle, the correction to the delay must be applied too. Through visual observation, it was quickly apparent that this correction was depended on the number of steps. A table of values was recorded for a range of steps, see table I. These points were plotted in figure 9 and an decaying exponential function was plotted using least squares fitting. A delay correction could be chosen using this function for any given number of steps in the function cycle and applied on each step.

III.   555 TIMER

The 555 Timer is a popular integrated circuit chip for timing, a pin diagram is included in figure 10 [9]. The 555 operates in a simple way; It's output is high when the 555 receives a trigger input (trigger pin voltage less than $\frac{1}{3}V_{\mathrm{cc}}$), and goes low when it receives a threshold input (threshold pin voltage higher than $\frac{2}{3}V_{\mathrm{cc}}$) [9]. When the output is low, the discharge pin connects to ground, and the when the reset pin is low, the output is forced low. The combination of these simple pin functions allows for a large set of complex applications [9].
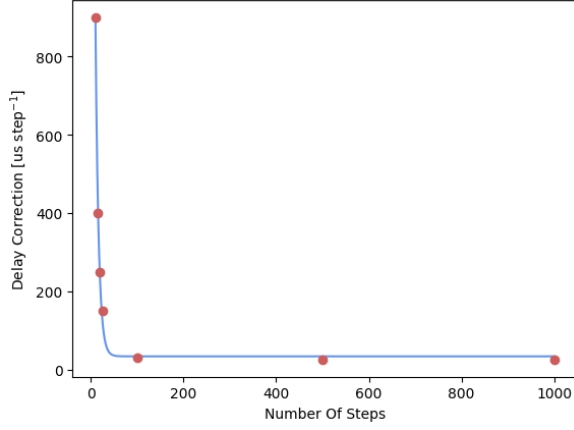
FIG. 9: The time correction values from table I and a least squares exponential fit. This function could then be used to define a time correction for any given number of steps.
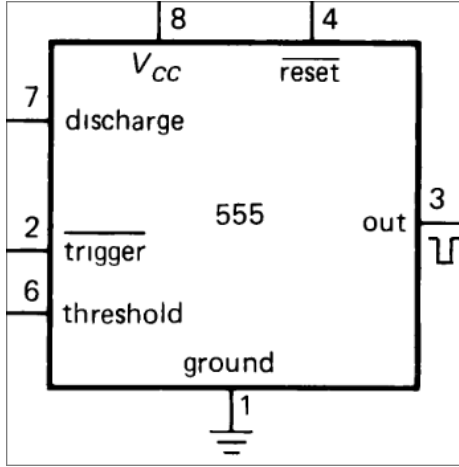

FIG. 10: A pin diagram for the 555 timing IC [9]. Note, pin 5: Control Voltage is excluded in this diagram.

A. 555 Timer as an Astable Oscillator

Based on the configuration of capacitors and resistors connected to the input pins of the 555, three main operating modes are possible: monostable (creating one fixed-width pulse and returning to stable off), bistable (operating as a flip flop, stable both on and off), and astable (constant oscillations) [11]. In this section, we will explore the functionality of an astable oscillator, see figure 11. In this schematic, we see the trigger and threshold pin both connected (through resistors) to V_{cc}. We also see the discharge pin (7) separating this connection. We can infer that as the circuit begins, the V_{cc} supply to the threshold will eventually yield the off output from the circuit, however it is slowed by the charging of the capacitor. Until the capacitor is fully charged, the output will be on. However, when the output switches to off, the discharge will be connected to ground, dropping the voltage to the trigger pin, discharging the capacitor and eventually yielding the on output. The astable function-
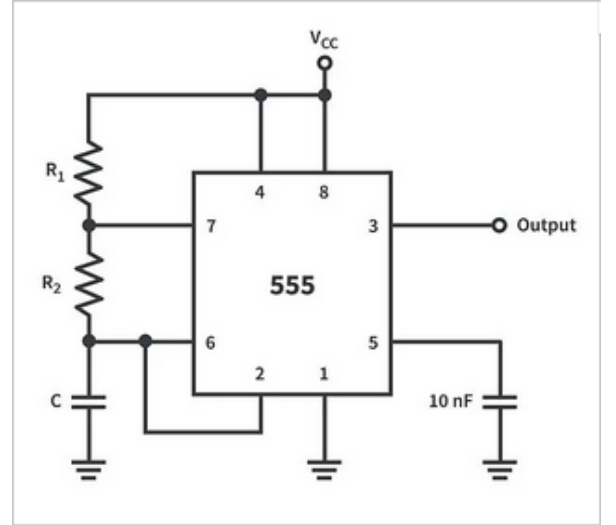

FIG. 11: The configuration of resistors and capacitors required for the 555 to operate in astable mode [11].

ality is created from this repeating cycle [11].

1. Simulations

We performed simulations of the 555 timer as an astable oscillator using TINA [12]. The circuit was set up as shown in figure 12. We observe that while the charging of the capacitor is dependent on both resistors (as both are between it and V_{cc}), the discharging only has to flow through one of the resistors. Hence, we can vary the values of each resistor to vary the length of time the 555 is on compared to off. We also find we can vary the overall frequency of oscillations by changing the capacitance so that the capacitor charges and discharges slower or faster. An example of this varied charging and discharging looks like is shown in figure 13. We can see how changing these resistor values effectively changes the duty of a PWM signal as shown in previous sections.

This PWM feature can be shown more clearly through the use of a potentiometer to vary resistance of one of the previous resistors. A circuit to demonstrate this was simulated in Tina as shown in figure 14 [12]. This circuit was simulated for different potentiometer resistances and capacitors to fully demonstrate the functionality of the circuit. Figure 15 shows simulation outputs showing how changing the value of capacitance will affect the frequency of oscillations in the output. Figure 16 shows simulation outputs showing how changing the value of the potentiometer affects the duty cycle of these oscillations. We see that the use of diodes in the circuit allows for the mapping of equal resistances to the 50% point of the potentiometer. We also see that for a potentiometer value of 0%, we still have an on signal for an appreciable portion of the cycle. This likely due to non-zero resistance minima in non-ideal potentiometers. The weight of this
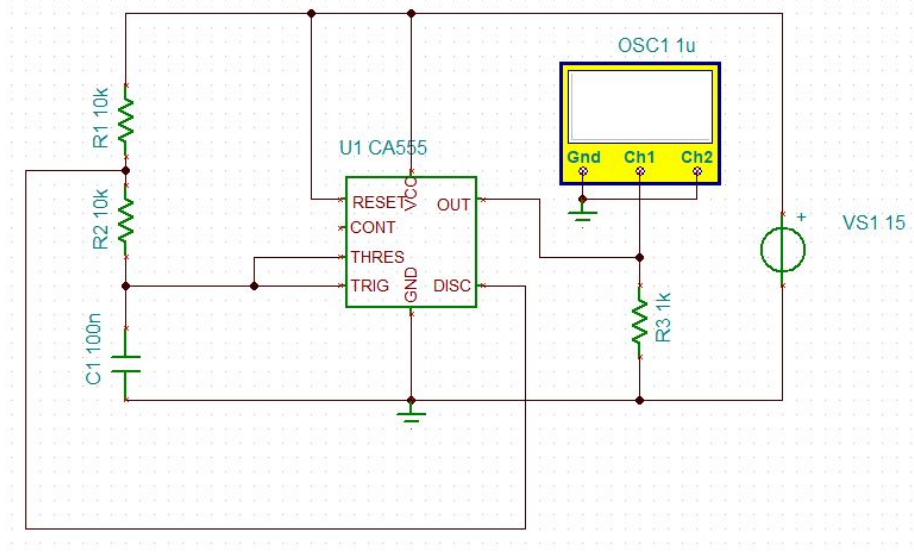
FIG. 12: The 555 Timer as an astable oscillator created using Tina [12]. An oscilloscope is placed at the output of 555 to measure its response.
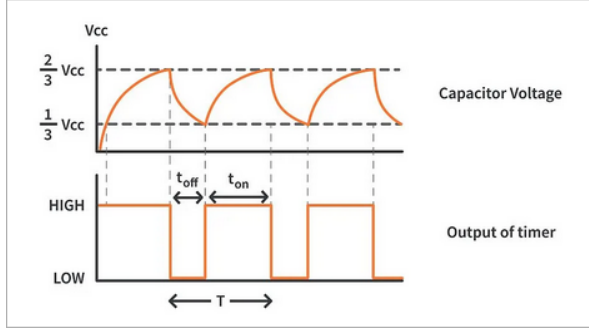

FIG. 13: How the charging and discharging of the capacitor in the figure 11 circuit results in differing off and on pulse widths.

effect can be mitigated by increasing the size of the other resistor between the potentiometer and V_{cc}, causing the on signal width to be smaller relative to the off signal width.

2. Circuit Construction

This circuit was then constructed following the diagram in figure 14. A 5 V external power supply was used to supply the V_{cc} and a simple motor was used to load the circuit. The output of the 555 was passed to the base of a transistor allowing current to flow from the power supply through the motor to ground. This circuit was tested for different RC values. As expected, we found that by varying the potentiometer value, the power to the motor changed and it spun faster or slower. By varying the capacitance in the circuit, the frequency of the 555 output could be changed. The motor response was predominantly the same regardless of frequency, with the most noticeable difference being in the audible tone the motor

produced changing in pitch. We encountered a large inertial resistance to spinning (the motor could first be set to higher potentiometer values to get the motor spinning before returning to the target speed), which seemed to be - while somewhat difficult to notice - more difficult to overcome with higher frequency outputs.

B. Designing a Time-To-Amplitude Converter utilising the 555 Timer

1. Theory

A time-to-amplitude converter (TAC) is a device capable of precisely measuring small time intervals and outputting a pulse with proportional amplitude [13]. The TAC is an important tool with applications in nuclear physics for measuring the decay times of a short-lived particle (such as in measuring muon lifetimes) [9, 14].

A simple circuit example of a TAC is included in figure 17 [13]. A source of constant current is used to charge a capacitor linearly (limited by saturation) by the following relation:

$$V = \frac{It}{C} \tag{11}$$

where V is the voltage across the capacitor, I is the constant current, and t the time [13]. After receiving a start signal, the "start" switch opens and "stop" switch closes allowing the capacitor to start charging. The "stop" switch subsequently closes on a stop signal and the voltage built up on the capacitor is gives the amplitude for the output signal. On a stop signal, the linear gate opens to pass the voltage pulse out, and then closes after some time resulting in a fixed width
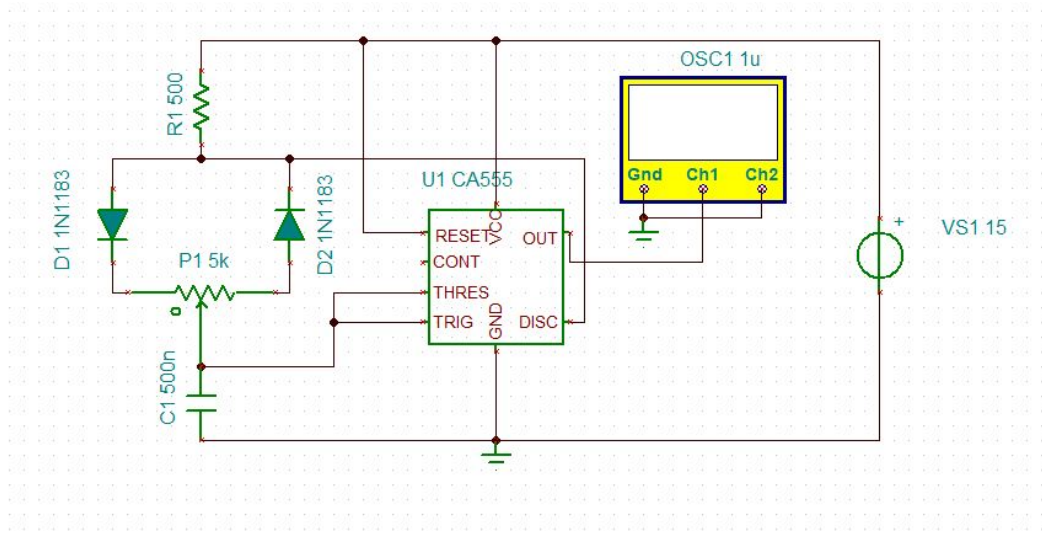
FIG. 14: The 555 timer in astable mode can be set up using a potentiometer to vary the speed of charging relative to discharging of the capacitor, enabling dynamic PWM [2].
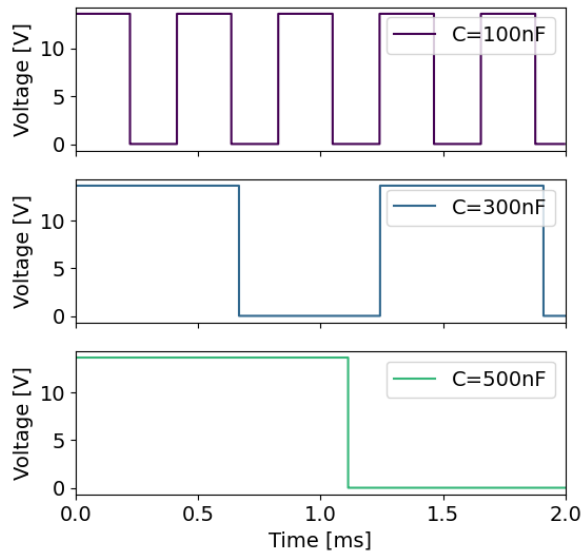

FIG. 15: Tina simulations of varying the capacitance in an astable oscillator circuit, see figure 14.
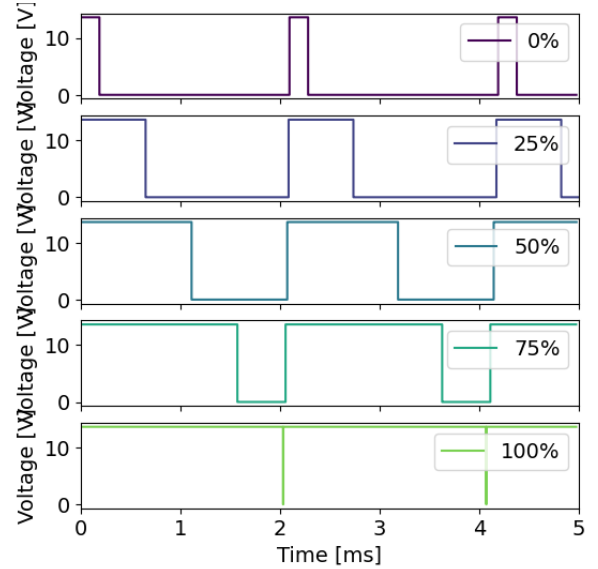

FIG. 16: Tina simulations of varying the charging to discharging rates via a potentiometer in an astable oscillator circuit, see figure 14. We see a linear mapping of potentiometer value to duty.

rectangular pulse output [13]. The capacitor is then discharged and switches are all returned to their initial positions.

While a simple representation, some of these components - namely the constant current source - are non-trivial to construct in a undergraduate lab environment. However, it should also be possible to recreate the functionality of this circuit (at least as a proof of concept) using a bistable 555 timer and an integrating op-amp.

2. Bistable 555 Timer

Discuss what it is, what is different from the astable. Include simple circuit diagram. Explain how the start and stop signals can power transistors to switch on and off.

A 555 timer in bistable mode differs from the astable mode described about in that both the on and off outputs are stable. The 555 effectively acts as a switch and remains in the on or off state as desired. The circuit diagram of the bistable 555 can be found on the left side of figure 18. The leading edge of the start signal will apply
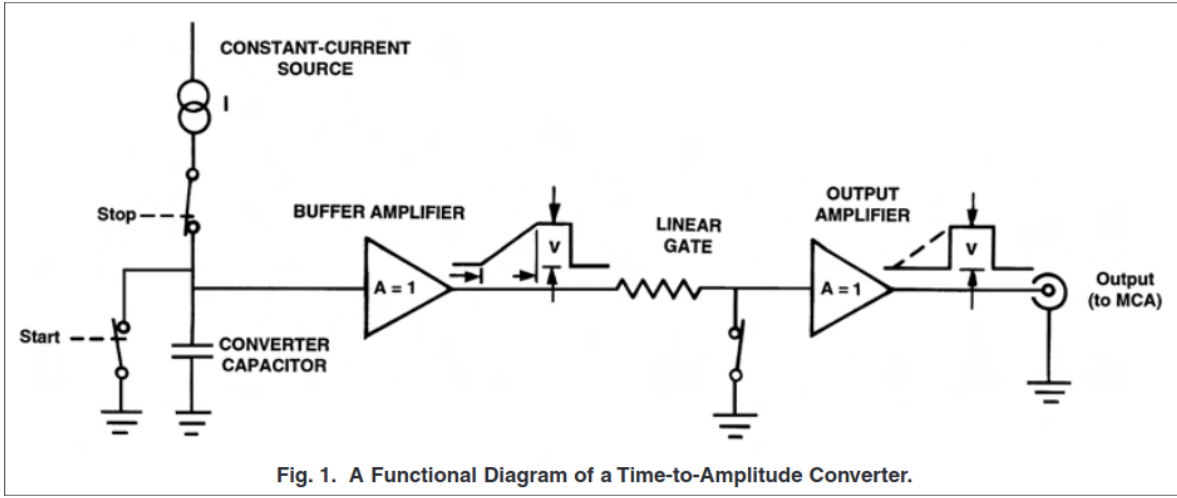
Fig. 1. A Functional Diagram of a Time-to-Amplitude Converter.

FIG. 17: A simplified example of a TAC circuit [13]. Note that toggle switches have been included for simplicity, and TAC circuits generally use transistors as switches for such actions [13].
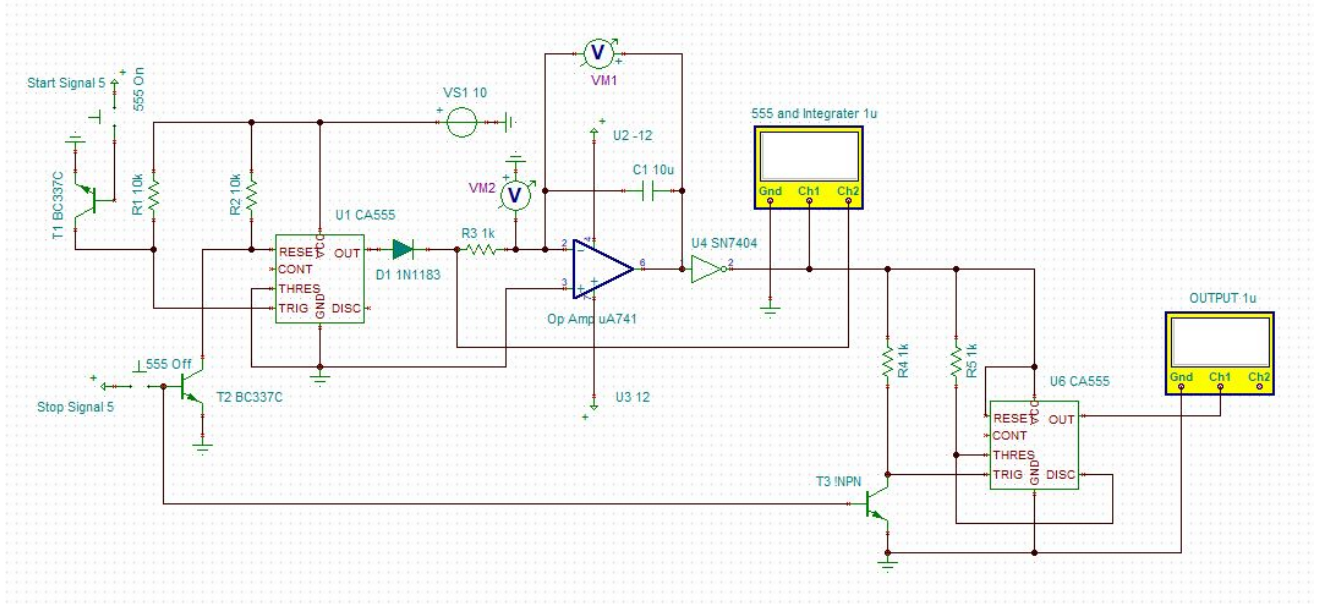

FIG. 18: A full circuit diagram for the TAC simulation in Tina.

a voltage to the base of a transistor, grounding the trigger and causing the 555 output to turn on. Similarly, a stop signal grounds the reset pin via a transistor causing the 555 output to be low. Hence, for the time between the leading edge of the start and stop signals, the 555 output will be high, after which it is low.

3. Integrating Op Amp

The output of the 555 is passed to an integrating op-amp. Op-amps allow for nearly perfect integrators, without the restriction of the output voltage being less than the input [9]. The integrator adds the signal from the inverting input by charging a capacitor in the feedback loop. The voltage of the capacitor with respect to the non-inverting input is then given by the following equation [9]:

$$V_{\text{out}} = -\frac{1}{RC} \int V_{\text{in}} dt + \text{const.} \tag{12}$$

where R and C are the resistance and capacitance of the integrator circuit and V_{in} is the voltage input. This voltage level can then be passed to any means of generating a pulse. The op-amp circuit in our TAC can be found in the centre of figure 18.

4. Monostable 555 Timer

To create the pulse, a monostable 555 Timer was used, see the right hand side of figure 18. A monostable 555 is constructed such that it creates a single fixed with pulse after grounding the trigger pin which has an amplitude equal to the V_{cc}. By passing the integrated voltage to the V_{cc} when a stop signal arrives, an output pulse with fixed width can be created by grounding the trigger.

The circuit as a whole then works in the following way. The start pulse arrives and grounds the trigger pin causing the output of the bistable 555 to go high. As a result, the input of the op-amp integrator is high and the capacitor starts charging as the op-amp integrates the 555 signal. When a stop signal arrives, the monostable 555 is reset and the output goes to off. The op-amp stops integrating further and a fixed with output pulse is created by the monostable 555, with amplitude proportional to the capacitor voltage, and hence, proportional to the time between the stop and start signals.

5. Testing

The circuit was simulated using Tina and some unexpected behaviour occurred. Unwanted capacitor charging and discharging was occurring in the Tina simulations of the circuit. A diode was included between the bistable 555 output and the integrator input to ensure no discharging could occur back through the 555. Other diode positions were tried with respect to the integrator but this position yielded expected output. A voltage inverter was used after the integrator. As shown in equation 12, the voltage output from the integrator is negative with respect to the input, and hence must be inverted to restore a positive voltage signal passed to the monostable 555. The circuit was difficult to demonstrate using Tina as - shown in figure 18 - push buttons were used to generate start and stop signals and hard to create consistency in inputs. Larger R and C values were used to slow the integration time to measure these longer time intervals.

We attempted to construct this circuit physically but unfortunately could not get it functional for manual signal inputs. The bistable 555 timer and transistor switches were implemented and functional, however, adding the integrating op-amp to the circuit proved non-trivial. Despite trying many configurations and thorough troubleshooting (limited due to time constraints) the integrator output remained at $\approx 0\,\mathrm{V}$ regardless of bistable 555 input.

6. Configuration

Should the circuit have worked as expected, several steps would need to taken to convert from a simple proof of concept to a practical device. Primarily, component values must be selected to map integration time to a chosen range of times - i.e. the millisecond range, or microsecond range - for a given amplitude range. The output pulse generation method we use restricts the integrated voltage between $5\,\mathrm{V}$ and $15\,\mathrm{V}$ due to limits on the 555 timer [11]. This can be done through the use of equation 12 as the voltage output of the integrator is inversely proportional to the RC value. The V_{cc} voltage of the bistable 555 also plays a roll in the speed of integration. Other useful functionality would be in the adjustment of the output pulse, specifically its width. This could be varied by switching resistance values in the monostable 555 circuit.

IV. CONCLUSIONS

Pulse width modulation and its applications were investigated and explored through the use of several examples including brightness variations of an LED, a simple digital-to-analogue converter, and the 555 timer. We also showed an example application for the 555 timer in a time-to-amplitude converter circuit. This circuit is explained in detail along with steps for configuration, and was simulated using the Tina circuit simulator. We attempted to construct this circuit physically, but encountered issues with the op-amp integrator and could not resolve them due to time constraints.

[1] BARR Group. Introduction to pulse width modulation (pwm). https://barrgroup.com/blog/introduction-pulse-width-modulation-pwm, 2001. Accessed: 2024-03-02.

[2] UCD School of Physics. 555 timer & raspberry pi pico pulse-width modulation, 2022. Accessed: 2024-03-02.

[3] John Marcon. Understanding pwm transmitters. https://www.radioworld.com/tech-and-gear/understanding-pwm-transmitters, 2017. Accessed: 2024-03-02.

[4] Raspberry Pi. Raspberry pi documentation: Raspberry pi pico and pico w. https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html. Accessed: 2024-03-02.

[5] Damien P. George. Micropython. https://micropython.org/. Accessed: 2024-03-02.

[6] Mary C. Potter, Brad Wyble, Carl E. Hagmann, and Emily S. McCourt. Detecting meaning in rsvp at 13 ms per picture. *Attention, Perception and Psychophysics*, 76(2):270–9, 02 2014.

[7] Intel. What is refresh rate and why is it important. `https://www.intel.com/content/www/us/en/gaming/resources/highest-refresh-rate-gaming.html`. Accessed: 2024-03-02.

[8] Christian Maes. Statistical mechanical foundation of weber–fechner laws. *Journal of statistical physics*, 182(3), 2021.

[9] Paul Horowitz and Winfield Hill. *The art of electronics; 3rd ed.* Cambridge University Press, Cambridge, 2015.

[10] eCircuit Center. Rc low-pass filter with op amp buffer. `http://www.ecircuitcenter.com/Circuits/opfil1/opfil1.htm`, 2010. Accessed: 2024-03-02.

[11] CircuitBread. 555 timer - 1. introduction to 555 timers. `https://www.circuitbread.com/tutorials/555-timer-1-introduction-to-555-timers`, 2022. Accessed:2024-03-05.

[12] DesignSoft. Tina: Circuit simulator for analog, digital, mcu, and rf circuits. `https://www.tina.com/`. Accessed:2024-03-05.

[13] ORTEC. Time-to-amplitude convertors and time calibrator. `https://virgilio.mib.infn.it/brofferio/didattica/datasheet/NoteOrtec-TAC-Time-Calibrator.pdf`. Accessed: 2024-03-07.

[14] R. E. Hall, D. A. Lind, and R. A. Ristinen. A simplified muon lifetime experiment for the instructional laboratory. *American journal of physics*, 38(10):1196–1200, 1970.

Appendix A: Pi Pico Pin Diagram

Raspberry Pi Pico Pinout
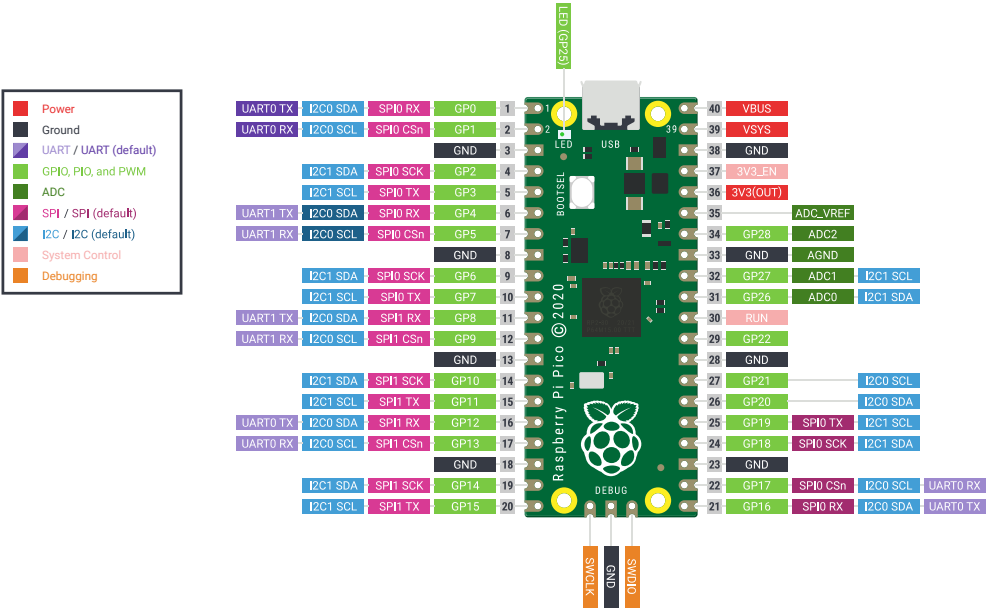
Appendix B: Linear / Log Brightness Script

```python
from machine import Pin, PWM
from time import sleep_ms
import math

# Pin 25 for inbuilt LED
pwm = PWM(Pin(25))

# Set frequency in Hz
pwm.freq(100)

# Initialise brightness value
brightness = 0

# Evenly spaced brightnesses in linear or log space
isLog = True

# specify the cycle length
cycleLength = 4 # seconds

# 200 steps in one cycle, converted to ms. Needs to be an int for sleep_ms() function
delay = round((cycleLength / 200) * 1000)

def GetBrighter(t, b, log=False):
    # Loop 100 points for each cycle

    i = 0
    while i < 101:
        print(f"{b:0.2f}")

        pwm.duty_u16(round(b * 65025))

        sleep_ms(t)

        if log:
            b = math.pow(10, i/100) / 10
        else:
            b += 0.01
        i += 1

    return b

def GetDimmer(t, b, log=False):
    # Loop 100 points for each cycle
    i = 0
    while i < 101:
        print(f"{b:0.2f}")

        pwm.duty_u16(round(b * 65025))

        sleep_ms(t)

        if log:
            b = math.pow(10, - i/100 + 1) / 10
        else:
            b -= 0.01
        i += 1

    return b

while True:

    if brightness <= 0.5:
        print("getting brighter")
        brightness = GetBrighter(delay, brightness, log=isLog)

    elif brightness >= 0.5:
        print("getting dimmer")
        brightness = GetDimmer(delay, brightness, log=isLog)

    else:
        break
```

Appendix C: Function Generator Script

```python
from machine import Pin, PWM
import math
from time import sleep_us

pwm = PWM(Pin(15))

period = 100 # ms

frequency = 1000 # Hz
sinPeriod = 1000 / frequency # ms

print(sinPeriod)

# How many points in the curve
# Note: there is a tradeoff in the accuracy of the curve,
# and the accuracy of the period due to the time Python takes to loop through all these steps
numSteps = 100
function = "sin"

# Delay correction!
# Table of correction values
stepValue = [10, 15, 20, 25, 100, 500, 1000]
correctionValue = [900, 400, 250, 150, 30, 25, 25]

# Delay correction function
def DelayCorrectionFunction(numberOfSteps):
    # Values determined using scipy's curve_fit and the above table of correction values
    return round(3799.4 * math.exp(-0.14904 * numberOfSteps)  + 25)

delayCorrection = DelayCorrectionFunction(numSteps) # us, subtacted from the delay of each step

# Set frequency in Hz
pwm.freq(25000)

# Set duty value between 0 and 1
dutyPercentages = []

# From 0 to numSteps inclusive
for i in range(numSteps + 1):

    if function == "sin":
        val = (2 * 3.141 * i / numSteps)
        dutyPercentages.append( (math.sin(val) + 1) / 2 )

    if function == "triangle":
        if i < (numSteps / 2):

            # multiply by two to increase duty range from 0 to 1 instead of 0 to 0.5
            val = (2 * i / numSteps)

        else:
            val = (-2 * i / numSteps + 2)

        dutyPercentages.append(val)


# Debug prints
print(dutyPercentages)

if function == "sin":
    delay = 1000 * (sinPeriod / numSteps) # us

else:
    delay = 1000 * (period / numSteps) # us

print(delay + delayCorrection)
print(delayCorrection)

dutyValues = []

for value in dutyPercentages:
    dutyValues.append(round(value * 65025))


def CycleDuties(dutyPercents, delay, delayCorrection):
    while True:
        for value in dutyPercents:
            pwm.duty_u16(value)
            sleep_us(delay - delayCorrection)
```

```
CycleDuties(dutyValues, round(delay), delayCorrection)
```

Appendix D: Python Analysis Notebooks

Task 1: PI PICO PWM Exercises

1a - Basics

```python
import numpy as np
import matplotlib.pyplot as plt

from matplotlib import cm
from glob import glob

cmap = cm.get_cmap('viridis')
plt.rcParams.update({'font.size': 14})
```

```
/tmp/ipykernel_1845292/4121011056.py:9: MatplotlibDeprecationWarning: The get_cmap function was deprec
ated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]``
` or ``matplotlib.colormaps.get_cmap(obj)`` instead.
  cmap = cm.get_cmap('viridis')
```

```python
dataPaths = glob("../data/pwmTests/*")

constantFrequencyPaths = [path for path in dataPaths if "f1000" in path]
constantDutyPaths = [path for path in dataPaths if "duty50" in path]

constantFrequencyPaths.sort()
constantDutyPaths.sort()

# Fix path sorting... moving first element to end
constantFrequencyPaths.append(constantFrequencyPaths.pop(0))
constantDutyPaths.append(constantDutyPaths.pop(0))
```

```python
# Plotting changes in duty while keeping frequency=1000 Hz constant
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
axes = axes.flatten()

# Evenly spaced colours in contrast
colours = np.linspace(.2, .8, len(constantFrequencyPaths))

for i, path in enumerate(constantFrequencyPaths):

    # Isolating the duty percentage from path
    duty = path.split("_")[-1]
    duty = duty[4:-4]

    data = np.loadtxt(path, skiprows=2, delimiter=",")

    axes[i].plot(data[:,0] * 1000, data[:,1], color=cmap(colours[i]))
    axes[i].set_title(f"Duty: {duty}%")

plt.tight_layout()
axes[2].set_xlabel("Time [ms]")
axes[3].set_xlabel("Time [ms]")
axes[0].set_ylabel("Voltage [V]")
axes[2].set_ylabel("Voltage [V]");
```
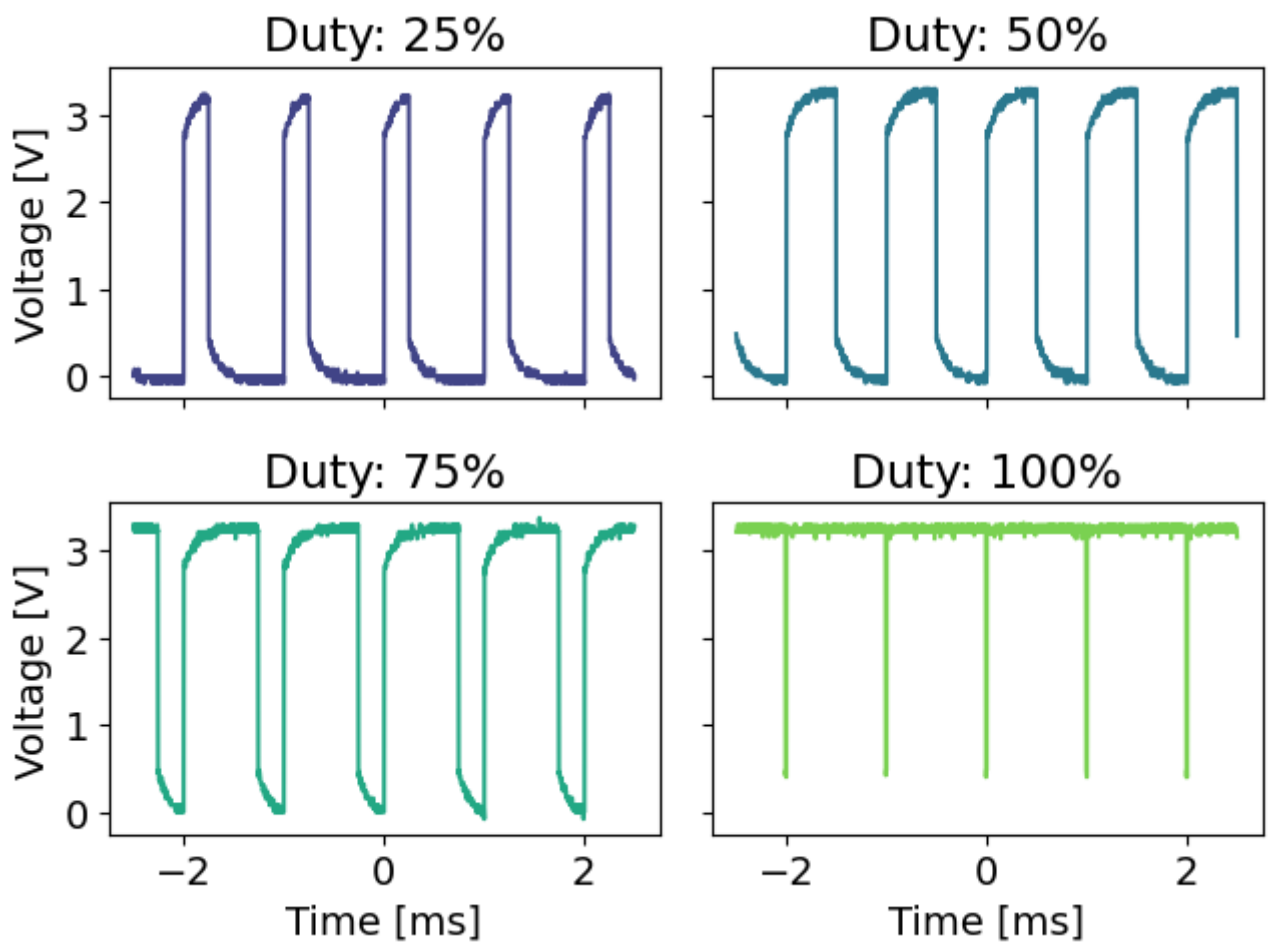
Duty: 25% Duty: 50%

Duty: 75% Duty: 100%

```
In [ ]:  # Plotting changes in frequency while keeping duty=50% constant
         fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
         axes = axes.flatten()

         # Evenly spaced colours in contrast
         colours = np.linspace(.2, .8, len(constantFrequencyPaths))

         for i, path in enumerate(constantDutyPaths):

             # Isolating the duty percentage from path
             frequency = path.split("_")[-2]
             frequency = frequency[1:]

             data = np.loadtxt(path, skiprows=2, delimiter=",")

             axes[i].plot(data[:,0] * 1000, data[:,1], color=cmap(colours[i]))
             axes[i].set_title(f"Frequency: {frequency} Hz")

         plt.tight_layout()
         axes[2].set_xlabel("Time [ms]")
         axes[3].set_xlabel("Time [ms]")
         axes[0].set_ylabel("Voltage [V]")
         axes[2].set_ylabel("Voltage [V]");
```
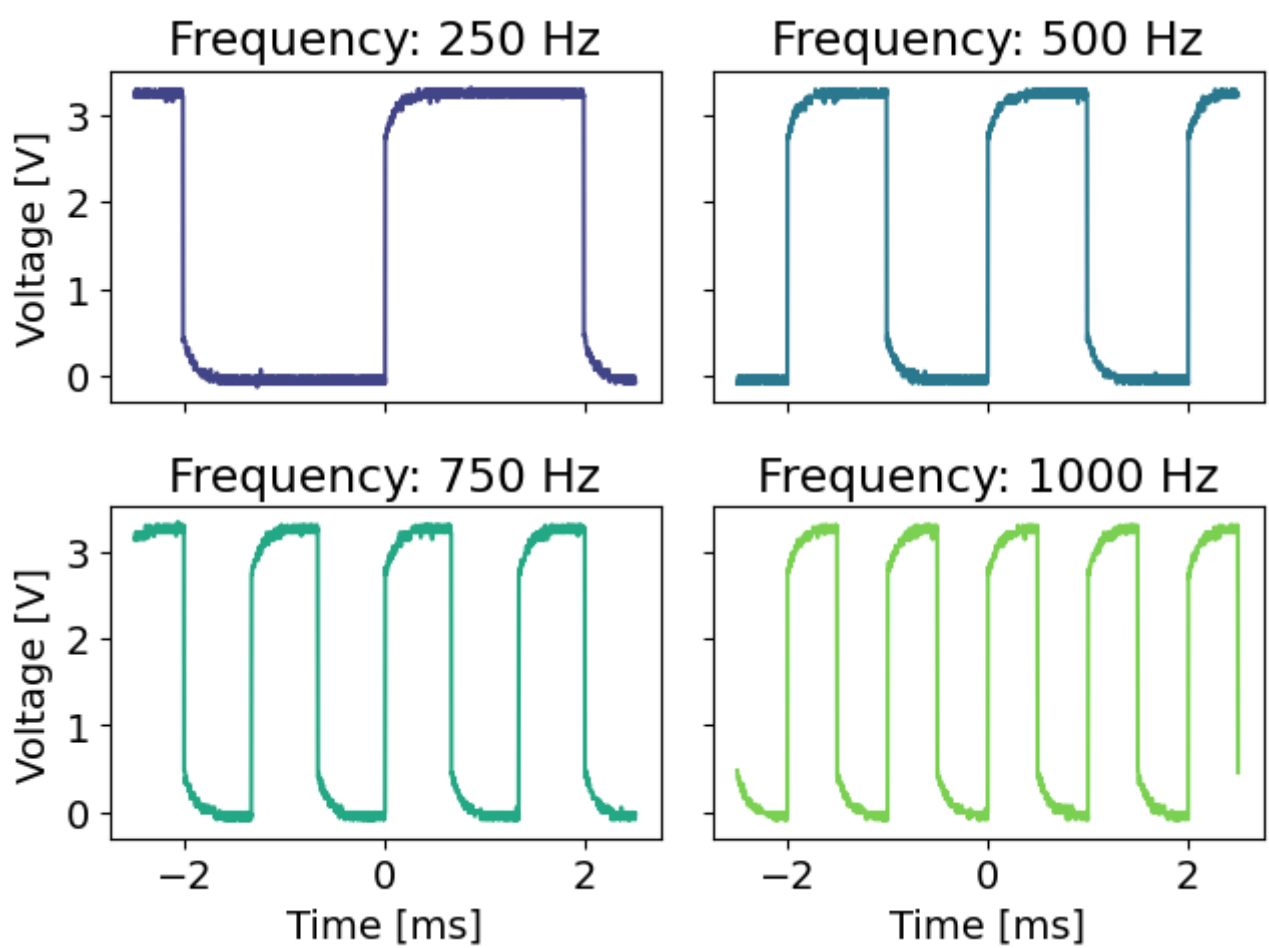
Frequency: 250 Hz   Frequency: 500 Hz

Frequency: 750 Hz   Frequency: 1000 Hz

```python
In [ ]:  def Increase(x, A):
             return A**x / A

         def Decrease(x, A):
             return A**-(x-1) / A

         xRange = np.linspace(0, 1, 100)

         A=10
         increasing = [Increase(x, A) for x in xRange]
         decreasing = [Decrease(x, A) for x in xRange]

         plt.plot(xRange, increasing)
         plt.plot(xRange, decreasing)

         plt.margins(0)
```
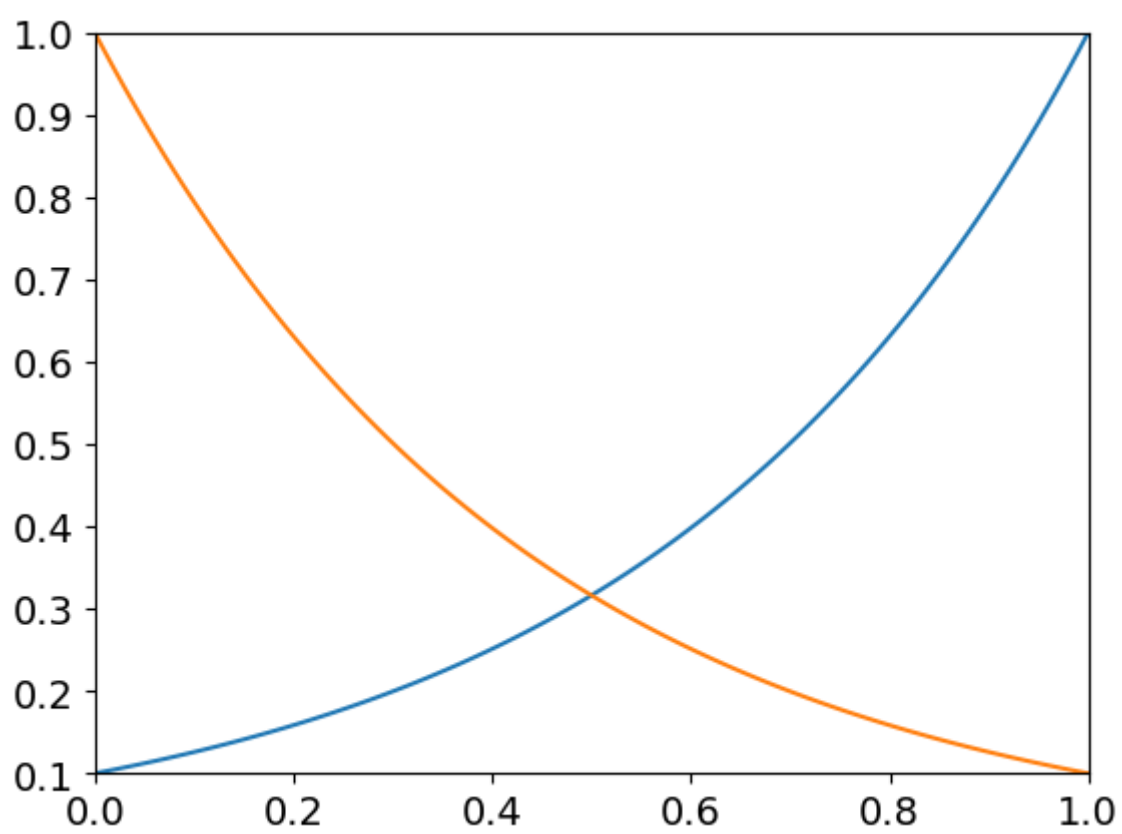
Task 1: PI Pico PWM Excercises

```python
import numpy as np
import matplotlib.pyplot as plt

from natsort import natsorted

from matplotlib import cm
from glob import glob

cmap = cm.get_cmap('viridis')
plt.rcParams.update({'font.size': 14})
```

```
/tmp/ipykernel_509405/1263976303.py:9: MatplotlibDeprecationWarning: The get_cmap function was depreca
ted in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]``
or ``matplotlib.colormaps.get_cmap(obj)`` instead.
  cmap = cm.get_cmap('viridis')
```

1b - Simple DAC

```python
data = np.loadtxt("../data/dacVariations/dacVariations_R680_1uF.csv", skiprows=2, delimiter=",")
```

```python
high = max(data[:,2])
low = min(data[:,2])

print((high - low) / np.mean(data[:,2]))
```

```
2.1459382257920043
```

```python
fig, axes = plt.subplots(2, 1, figsize=(6,6), sharex=True)

ax0, ax1 = axes

ax0.plot(data[:,0]*1000, data[:,1], label="Output Signal: < 5% var.")
ax0.plot(data[:,0]*1000, data[:,2], label="PWM Input")

ax0.set_ylabel("Voltage [V]")
ax1.set_ylabel("Normalised Voltage")

ax1.plot(data[:,0]*1000, data[:,1]/np.mean(data[:,1]))

plt.xlabel("Time [ms]")
ax0.legend(loc="lower right");
```
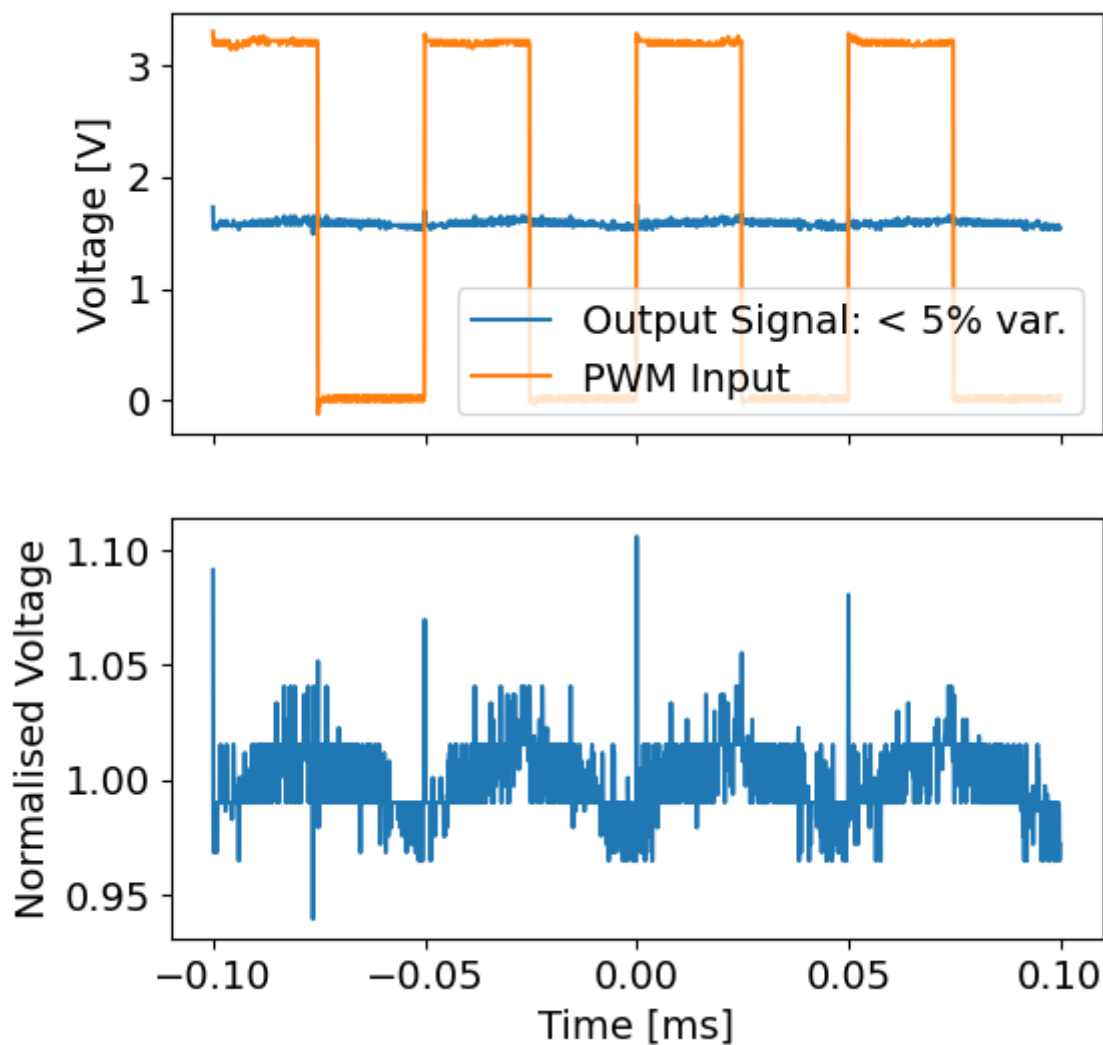
1c - Generating analogue output functions

```
In [ ]:  dataPaths = glob("../data/trianglePeriods/*")
         print(dataPaths)
```

```
['../data/trianglePeriods/50ms.csv', '../data/trianglePeriods/100ms.csv']
```
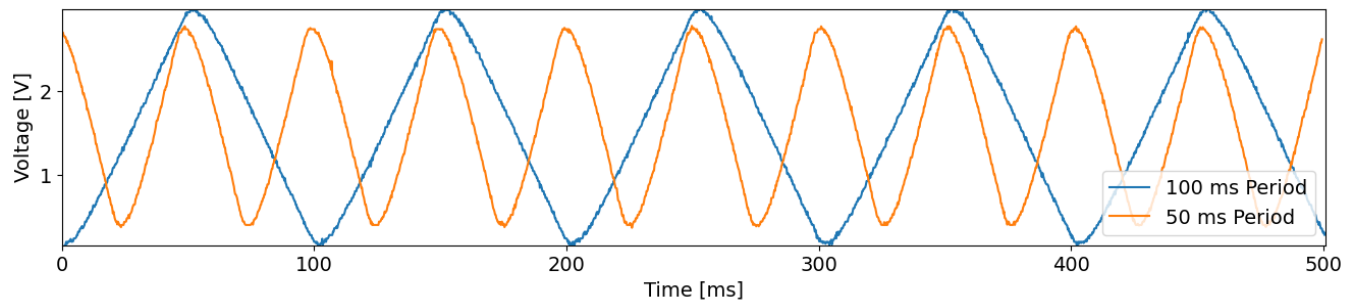
```
In [ ]:  fig, ax = plt.subplots(figsize=(16,3))

         labels = ["100 ms Period", "50 ms Period"]

         for i, path in enumerate(dataPaths):
             data = np.loadtxt(path, skiprows=2, delimiter=",")

             # Alligning start points
             if i == 0:
                 ax.plot(data[:,0]*1000+227, data[:,1], label=labels[i])
             else:
                 ax.plot(data[:,0]*1000+238, data[:,1], label=labels[i])


         ax.set_xlabel("Time [ms]")
         ax.set_ylabel("Voltage [V]")
         ax.legend(loc="lower right")
         ax.margins(0);
```

```
In [ ]:  dataPaths = glob("../data/sinFrequencies/*")
         dataPaths = natsorted(dataPaths)
         print(dataPaths)
```

['../data/sinFrequencies/sin1Hz.csv', '../data/sinFrequencies/sin5Hz.csv', '../data/sinFrequencies/sin10Hz.csv', '../data/sinFrequencies/sin15Hz.csv', '../data/sinFrequencies/sin20Hz.csv', '../data/sinFrequencies/sin25Hz.csv', '../data/sinFrequencies/sin50Hz.csv', '../data/sinFrequencies/sin75Hz.csv', '../data/sinFrequencies/sin100Hz.csv', '../data/sinFrequencies/sin1000Hz.csv']

```
In [ ]:  fig, ax = plt.subplots(figsize=(16,3))

         # Extracting just the frequency from the file name
         labels = [path.split("/")[-1][3:-4] for path in dataPaths]

         for i, path in enumerate(dataPaths):

             # Choosing to plot only a select few
             if i in [0, 1, 3, 4, 9]:
                 continue

             data = np.loadtxt(path, skiprows=2, delimiter=",")

             ax.plot(data[:,0], data[:,1], label=labels[i])

         ax.set_xlim(0.02, 0.07)
         ax.set_xlabel("Time [ms]")
         ax.set_ylabel("Voltage [V]")
         ax.legend(loc="lower right")
         ax.margins(x=0);
```
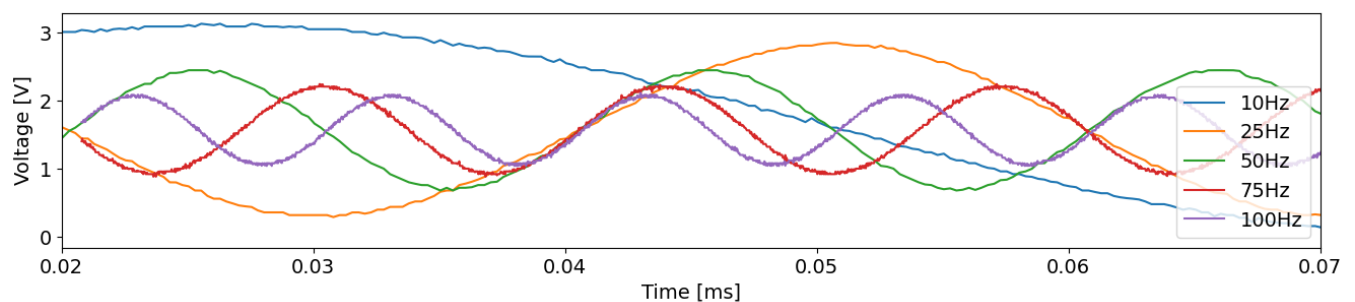
```python
In [ ]:  def DetermineAmplitude(data):
             # Rough guess of the amplitude through subtracting the mean from the max value
             # Could use curve_fit but initial parameters of frequency get difficult

             centreValue = np.mean(data[:,1])

             amplitude = np.max(data) - centreValue

             return amplitude


         sinAmplitudes = []

         for path in dataPaths:

             data = np.loadtxt(path, skiprows=2, delimiter=",")

             sinAmplitudes.append(DetermineAmplitude(data))

         def ExponentialCurve(x, a, b, c):
             y = []

             for point in x:
                 y.append(a * exp(- b * point) + c)

             return y

         #pars, cov = curve_fit(ExponentialCurve)

         frequencies = [int(el[:-2]) for el in labels]
         decibles = 10*np.log10(np.array(sinAmplitudes) / 1.65) # Pico IO voltage is fixed at 3.3V

         fig, ax = plt.subplots(figsize=(8, 6))

         ax.scatter(frequencies, decibles, label="Generated Sin Waves")
         ax.vlines([15.915], ymin=-12, ymax=1, ls="dashed", color="orange", label="$f_c=15.915$ Hz")

         ax.set_xscale("log")
         ax.set_xlabel("Frequency [Hz]")
         ax.set_ylabel("Amplitude Attenuation [dB]")
         ax.legend()
         ax.margins(y=0)
```
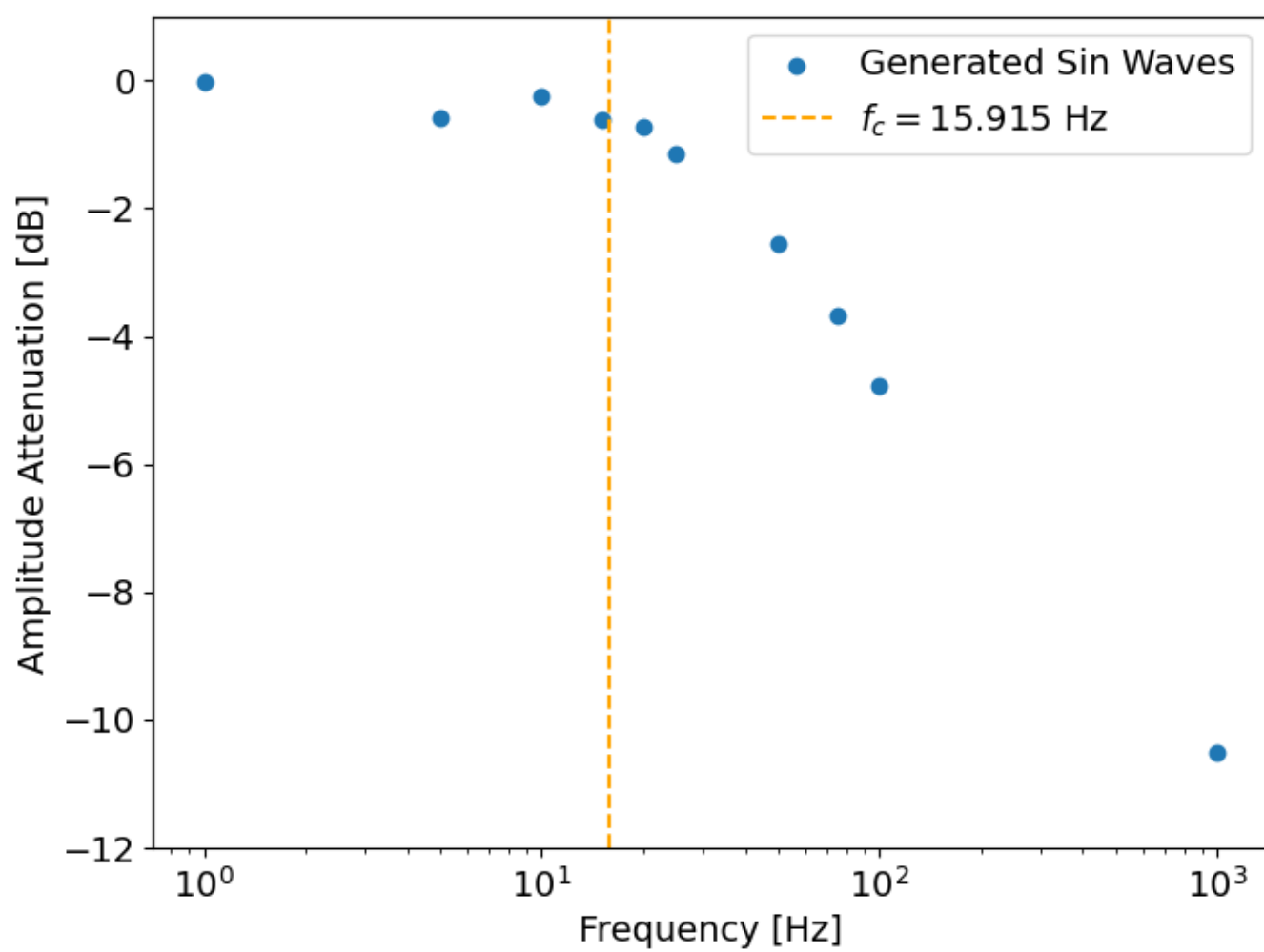
Task 1c, calibrations

```
In [ ]:  import numpy as np
         import matplotlib.pyplot as plt

         from scipy.optimize import curve_fit
```

```
In [ ]:  stepValues = [10, 15, 20, 25, 100, 500, 1000]
         correctionValues = [900, 400, 250, 150, 30, 25, 25]

         correctionValuesBySteps = [a / b for a, b in zip(stepValues, correctionValues)]
```

```
In [ ]:  def ExpFunc(x, a, b, c):
             return a * np.exp(-b * x)  + c

         pars, cov = curve_fit(ExpFunc, stepValues, correctionValues)
         print(pars)
```

```
[8106.82039178  -58.0198913 ]
```
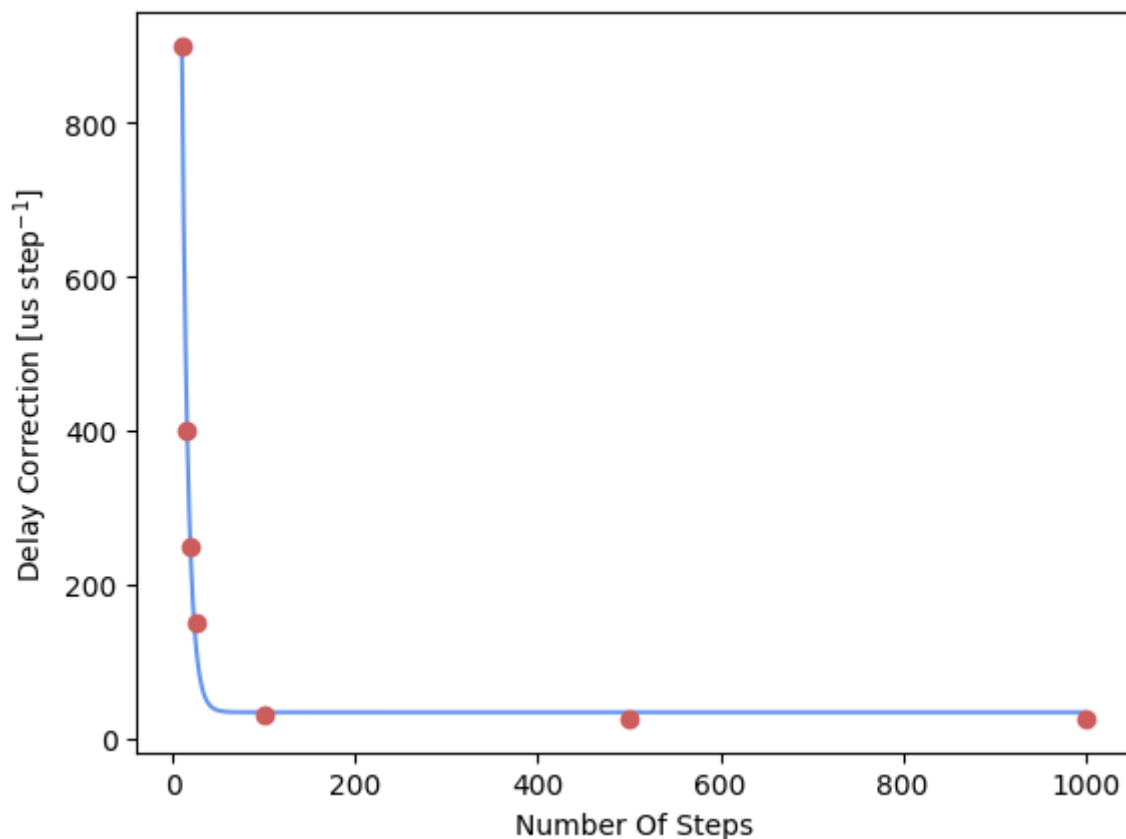
```
In [ ]:  plt.scatter(stepValues, correctionValues, color="indianred", zorder=5)

         xRange = np.linspace(10, 1000, 1000)

         plt.plot(xRange, ExpFunc(xRange, pars[0], pars[1], pars[2]), color="cornflowerblue")

         plt.xlabel("Number Of Steps")
         plt.ylabel("Delay Correction [us step$^{-1}$]")
         #plt.xlim(0, 200)
```

```
Out[ ]:  Text(0, 0.5, 'Delay Correction [us step$^{-1}$]')
```

Task 2: 555 Timer

```python
import numpy as np
import matplotlib.pyplot as plt

from natsort import natsorted

from matplotlib import cm
from glob import glob

cmap = cm.get_cmap('viridis')
plt.rcParams.update({'font.size': 14})
```

```
/tmp/ipykernel_810156/1263976303.py:9: MatplotlibDeprecationWarning: The get_cmap function was depreca
ted in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]``
or ``matplotlib.colormaps.get_cmap(obj)`` instead.
  cmap = cm.get_cmap('viridis')
```

```python
dataPaths = glob("../data/potentiometer/frequencyVariation/*")
dataPaths = natsorted(dataPaths)
print(dataPaths)
```

```
['../data/potentiometer/frequencyVariation/100nF.txt', '../data/potentiometer/frequencyVariation/300n
F.txt', '../data/potentiometer/frequencyVariation/500nF.txt']
```

```python
fig, axes = plt.subplots(3, 1, figsize=(6,6), sharex=True)

# Extract labes from file paths
labels = [f"C={el.split("/")[-1][0:5]}" for el in dataPaths]

for i, path in enumerate(dataPaths):

    data = np.loadtxt(path, skiprows=1)

    axes[i].plot(data[:,0]*1000, data[:,1], color=cmap(i/len(axes)), label=labels[i])

    axes[i].set_ylabel("Voltage [V]")
    axes[i].legend(loc="upper right")

plt.xlim(0, 2)
plt.xlabel("Time [ms]")
```
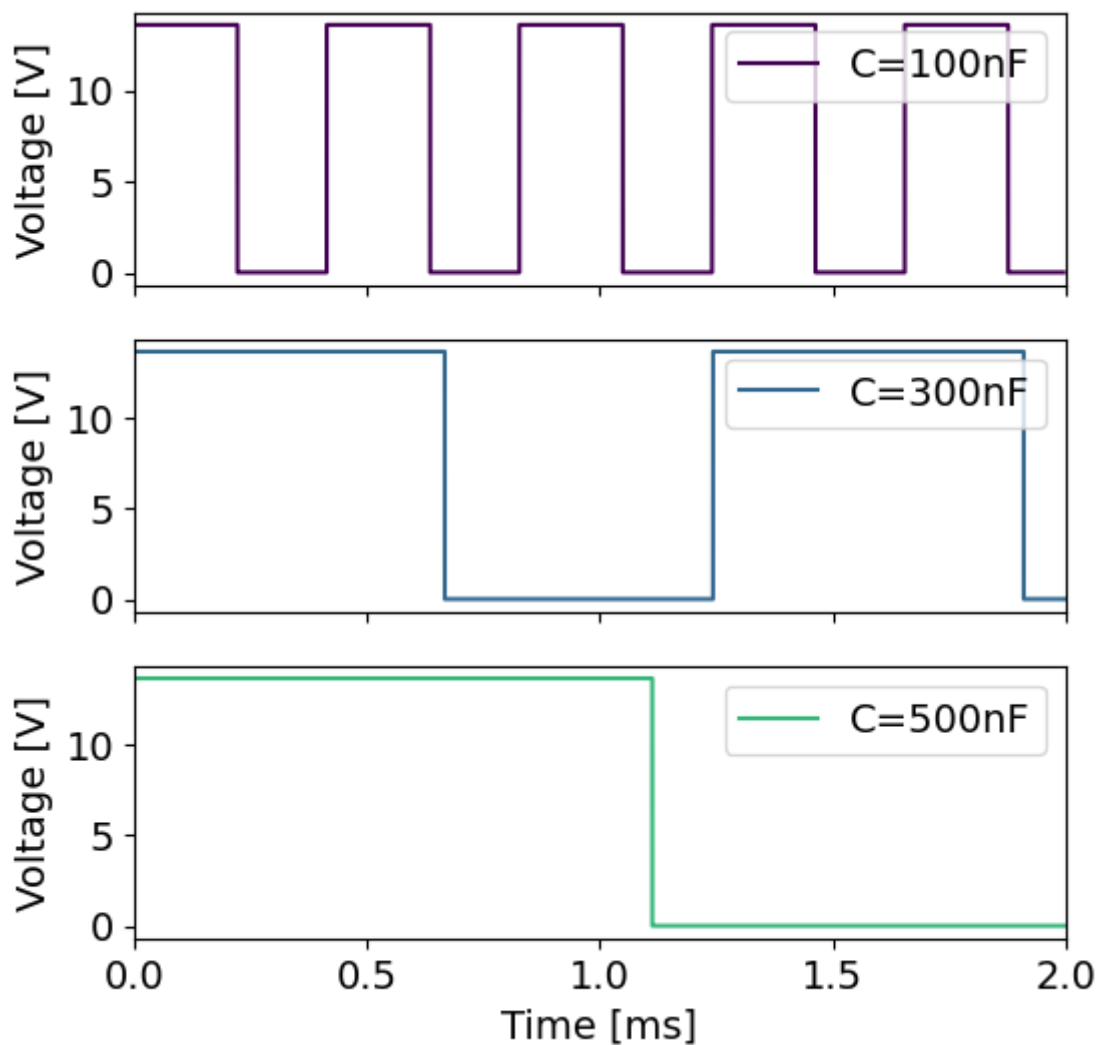
```
Text(0.5, 0, 'Time [ms]')
```

```
dataPaths = glob("../data/potentiometer/potentiometerVariation/*")
dataPaths = natsorted(dataPaths)
print(dataPaths)
```

['../data/potentiometer/potentiometerVariation/p0.txt', '../data/potentiometer/potentiometerVariation/p25.txt', '../data/potentiometer/potentiometerVariation/p50.txt', '../data/potentiometer/potentiometerVariation/p75.txt', '../data/potentiometer/potentiometerVariation/p100.txt']

```
fig, axes = plt.subplots(5, 1, figsize=(6,6), sharex=True)

labels = ["0%", "25%", "50%", "75%", "100%"]

for i, path in enumerate(dataPaths):

    data = np.loadtxt(path, skiprows=1)

    axes[i].plot(data[:,0]*1000, data[:,1], color=cmap(i/len(axes)), label=labels[i])

    axes[i].set_ylabel("Voltage [V]")
    axes[i].legend(loc="upper right")

plt.xlim(0, 5)
plt.xlabel("Time [ms]")
```

Text(0.5, 0, 'Time [ms]')