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 3, 2024)

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].

The 555 Timer is a...

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.

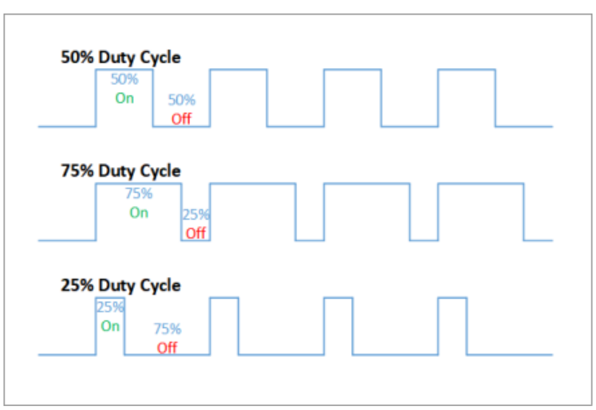


FIG. 1: Examples of difference duty cycles for a square wave [2].

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 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

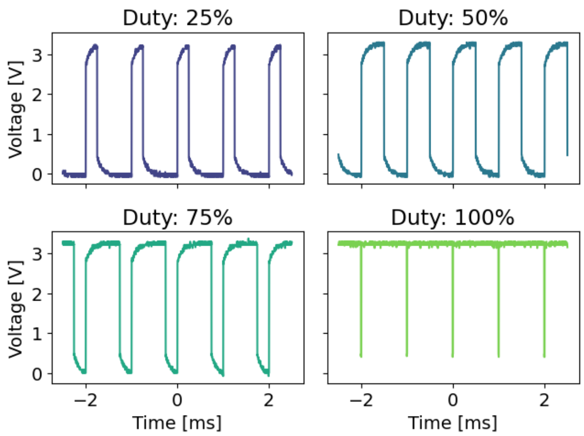


FIG. 2: PWM output from the Pi Pico showing variations in duty for a constant frequency of 1000 Hz

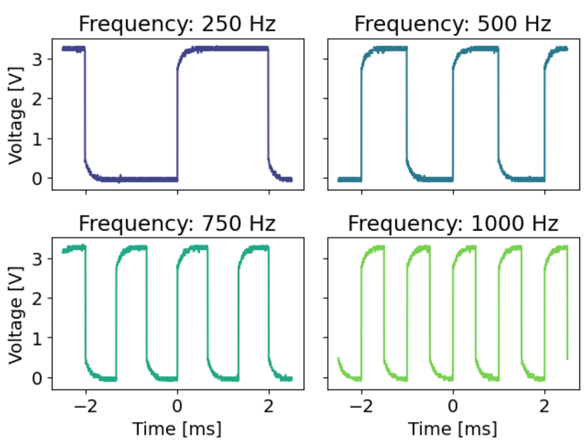


FIG. 3: PWM output from the Pi Pico showing variations in frequency for a constant duty of $\approx 50\%$ (i.e. $65535/2 \approx 32768$, rounding to an integer value.)

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 $\mathcal{O}(\mathrm{ms})$ could be applied to define the length of time to cycle through the loop. This delay t_{delay} was defined as follows:

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

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 attennuating effect.

Given a test input PWM signal of 20 kHz and a 50% duty, a combination of R and C was found to produce a signal with less than 5% variation. To have less than 5% variation, we need to attenuate the signal by a factor of 1/20. Coverting this to decibels we have:

$$\begin{aligned} \mathrm{dB} &= 10\log_{10}\left(\frac{\text{Signal In}}{\text{Signal Out}}\right) \\ &= 10\log_{10}(20) \\ &\approx 13\,\mathrm{dB} \end{aligned} \tag{5}$$

hence we must reduce the signal by 13 dB to achieve this. Based on attenuation slope of $-20\,\mathrm{dB}$ per decade we reach this attentuation for frequencies $1.5 \cdot f_c$. Therefore, for a PWM frequency of 20 kHz, we require a cut-off frequency:

$$f_c \leq \frac{20\,\mathrm{kHz}}{1.5} \approx 13\mathrm{kHz} \tag{6}$$

To achieve this, a combination of $R = 270\,\Omega$ and $C = 0.1\,\mu\mathrm{F}$, yielding $f_c \approx 5.9\,\mathrm{kHz}$ was chosen based on the components available. NEED TO TAKE LAB DATA FOR THIS. This output was verified using the oscilloscope and plotted in figure **??**.

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.

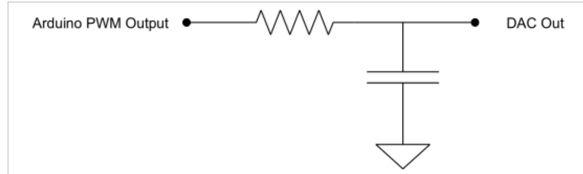


FIG. 4: A low pass filter to convert the PWM input to an analogue output [2]. Note a Pi Pico is used in our case instead of an Arduino, however the principles remain the same.

FIG. 5: The output from the simple DAC with a PWM input of 20 kHz and $R = 270\,\Omega$, $C = 0.1\,\mu\mathrm{F}$

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:

$$t_{\mathrm{delay}} = \frac{\mathrm{period}}{N} \tag{7}$$

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

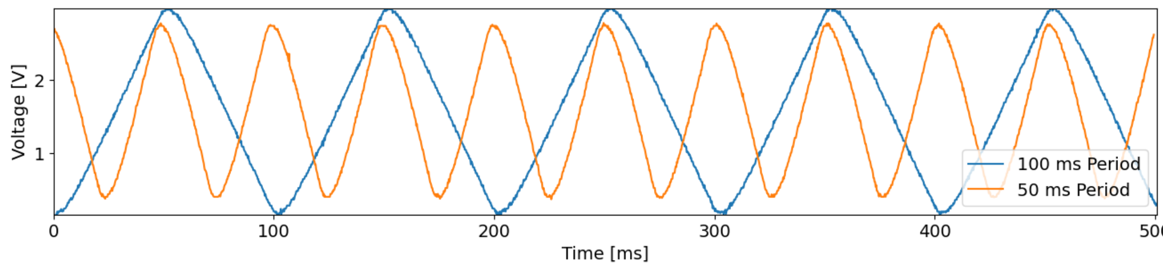


FIG. 6: Two triangle functions generated using the combination of PWM and a Low Pass Filter. The period was varied by making a time delay in updating each step of the curve. We note the difficulty in achieving a sharp peak on this function. Higher resistance values in the RC circuit produced sharper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

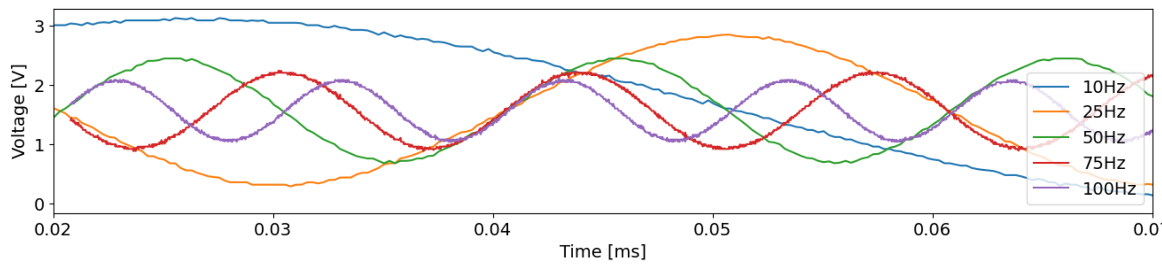


FIG. 7: Several sin functions generated using the combination of PWM and a Low Pass Filter. The frequency was varied with the same method as in the triangle function case.

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 $1\,\mathrm{k}\Omega$ 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:

$$S(i) = \frac{1}{2}\left(\sin\left(2\pi\frac{i}{N}\right) + 1\right) \tag{8}$$

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 $f_c = 15.915$ and is plotted as an orange vertical line through the data.

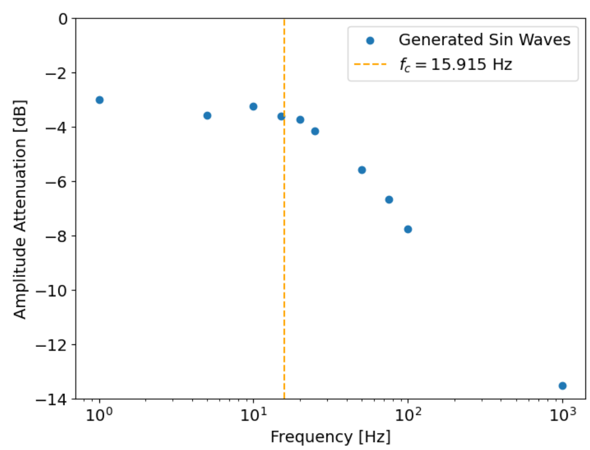


FIG. 8: The amplitude of the output sin wave plotted with respect to the frequency. We see the typical attenuation curve we expect from a low pass filter. Cut-off frequency (for $R = 1\,\mathrm{k}\Omega$ and $C = 10\,\mu\mathrm{F}$) is marked by the vertical line.

Where it intersects the data is approximately -3.75 dB, close the expected value of -3 dB. It is hence 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 ($\mathcal{O}(10\,\mathrm{ms})$) 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 visual 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.

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.

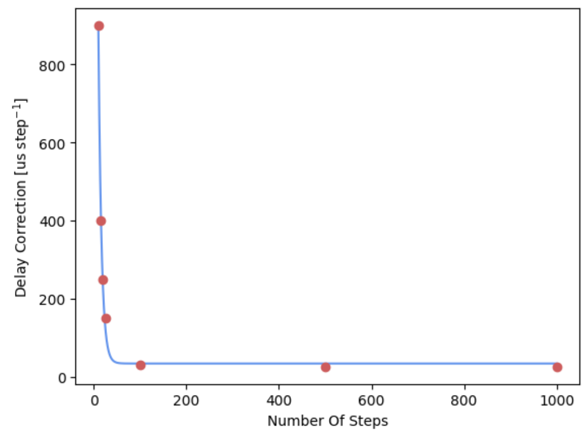


FIG. 9: The time correction values from table I and a least squares exponential fit. This function could then be used to define a time correction for any given number of steps.

III. 555 TIMER

A. 555 Timer as an Astable Oscillator

1. Simulations

2. Circuit Construction

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

[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.

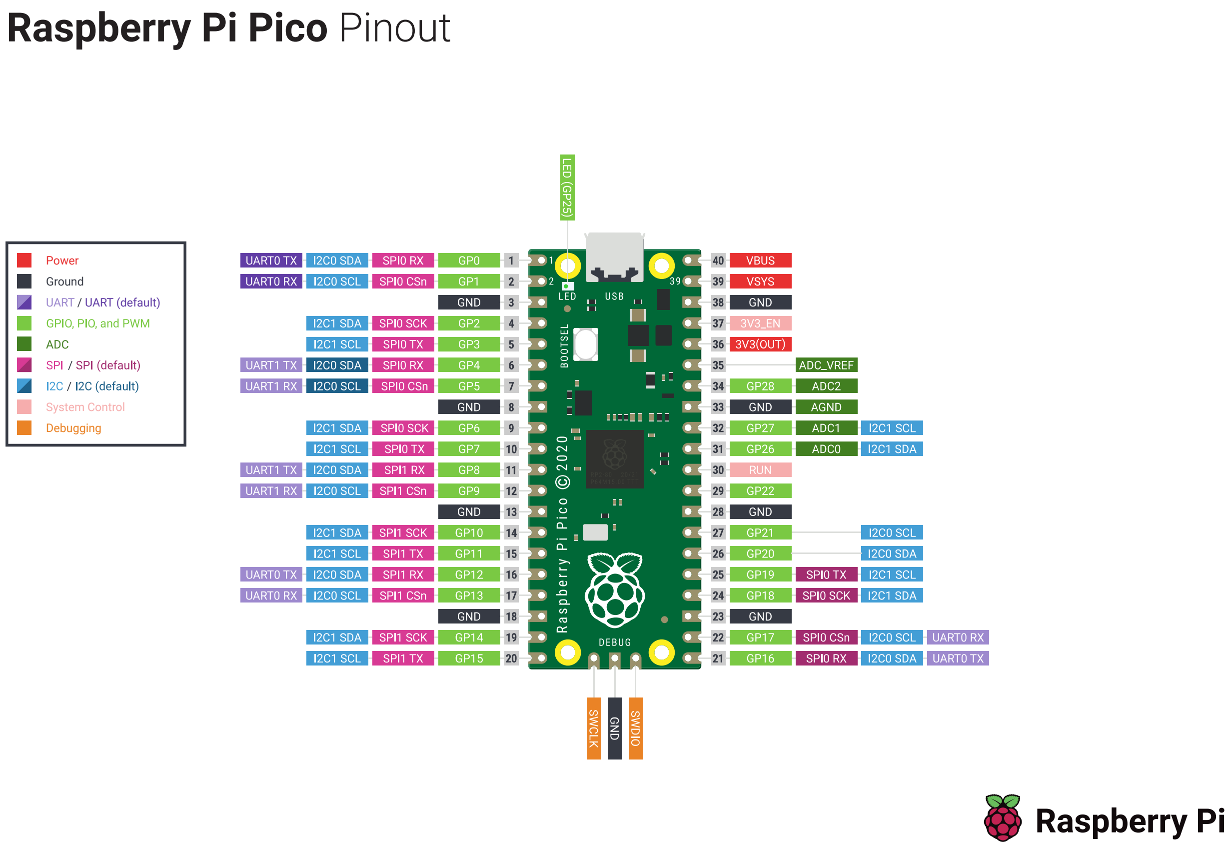
[7] Intel. What is refresh rate and why is it important. https://www.intel.com/content/www/us/en/gaming/resources/highest-refresh-rate-gaming.html. Accessed: 2024-03-02.

[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.

Appendix A: Pi Pico Pin Diagram



Appendix B: Linear / Log Brightness Script

```
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 Notebook