Bachelor's thesis

Embedded Software and IoT

2024

Henry Pekkermann

Mesmerizing kinetic sand art table

TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Embedded Software and IoT

2024 | Total number of pages: 45

Henry Pekkermann

Mesmerizing kinetic sand art table

After seeing a coffee table that draws patterns on sand, it seemed interesting enough to get into project design and implementation. The goal was to engineer a mechanic that could move a magnet under the surface to move a ball on top of the sand surface.

This was developed by using a cost-effective scara-type robotic arm moved by two stepper motors and controlled by an Arduino nano board. The user interface was designed using the Django web server framework running on a Raspberry PI. The patterns are drawn by a .thr file that contains a list of polar coordinates that are transmitted to Arduino through a USB serial connection. The motor movement was calculated with inverse kinematics for a 2-link planar robotic system.

This was a personal project to gain experience and the end product was implemented in “good enough” working condition.

Keywords:

embedded system, raspberry pi, arduino nano, mechanical design

Content

Illutrations

Tables

List of abbreviations (or) symbols

CNC	Computer Numerical Control
CSS	Cascading Style Sheets
DC	Direct Current
FIFO	First In, First Out
HTML	HyperText Markup Language
LED	Light Emitting Diode
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
PWM	Pulse Width Modulation
RGB	Red, Green, Blue
RGBW	Red, Green, Blue, White
SPR	Steps Per Revolution
USB	Universal Serial Bus

1 Introduction

This paper introduces a kinetic sand art coffee table and follows the build and development of the project.

The goal is to engineer a robotic arm to solve the problem of moving the ball on the surface. This is a personal project and the end product needs to be good enough, meaning that it needs to work as intended and be quiet enough to not be annoying.

The thesis oversees planning and developing the mechanics, electronics, and software for the robot arm. It concentrates on the firmware for the microcontroller that controls the motors for the arm. This only touches the topic of mechanical design and does not cover building the actual table.

Illustration 1: Finished project

2 Kinetic Sand Art Table

On the 42nd of September 2016, a Kickstarter campaign was launched by Bruce Shapiro, which introduced Sisyphus – The Kinetic Art Table[1]. The Kickstarter project was successfully funded in 24th of October 2016 with 1,992 backers raising $1,924,018. It quickly became famous on social media, hence the source of the idea in this thesis project.

Sisyphus Industries[2] introduced a coffee table that constantly draws new patterns and shapes and erases old ones. It uses a robotic arm under a glass surface to move a magnet which moves a metal ball on the surface. Between the metal ball and glass surface is very fine sand for the ball to draw the patterns, and fabric for the sand not to damage the glass and make it quieter. The LEDs around the ring are there to enhance the height disparity for the patterns.

3 System design

Overall design went through three iterations, because of problems. This chapter will explain what problems were encountered and how they were solved.

3.1 Version 1

The first version was mainly for testing stepper motor and LED driving. It used Raspberry PI and a stepper motor shield for Raspberry PI[3]. The motor shield had two integrated DRV8825 drivers and was controlled with the PI using Python.

The problem with this design was that the DRV8825 drivers were extremely loud and could not be changed.

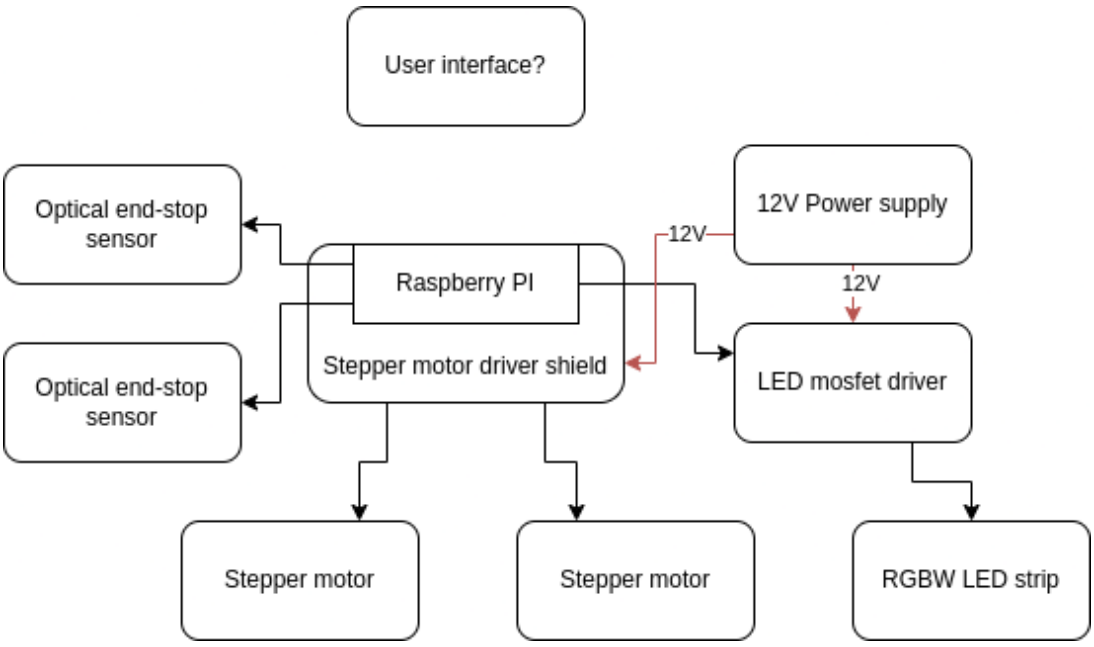

Illustration 2: System design Version 1

3.2 Version 2

The second design started by improving the stepper motor drivers. Research into the topic showed that TMC2208 were more advanced and quieter stepper motor drivers, but a new carrier board was needed. A CNC shield by Protoneer[4] suited perfectly for housing the drivers and only needed an Arduino nano or mega to drive it which was cheaper than Raspberry PI.

When designing the user interface, simplicity was kept in mind, and a serial Bluetooth module was used to talk with Serial Bluetooth Terminal[5] -application on a phone. Communication was text-based and users needed to query menus and track lists by sending text commands. It used Arduino Mega as a processor and SD-card reader to save and read the coordinate tracks.

The text-based user interface was slow and prone to misspelling and was disregarded because of that.

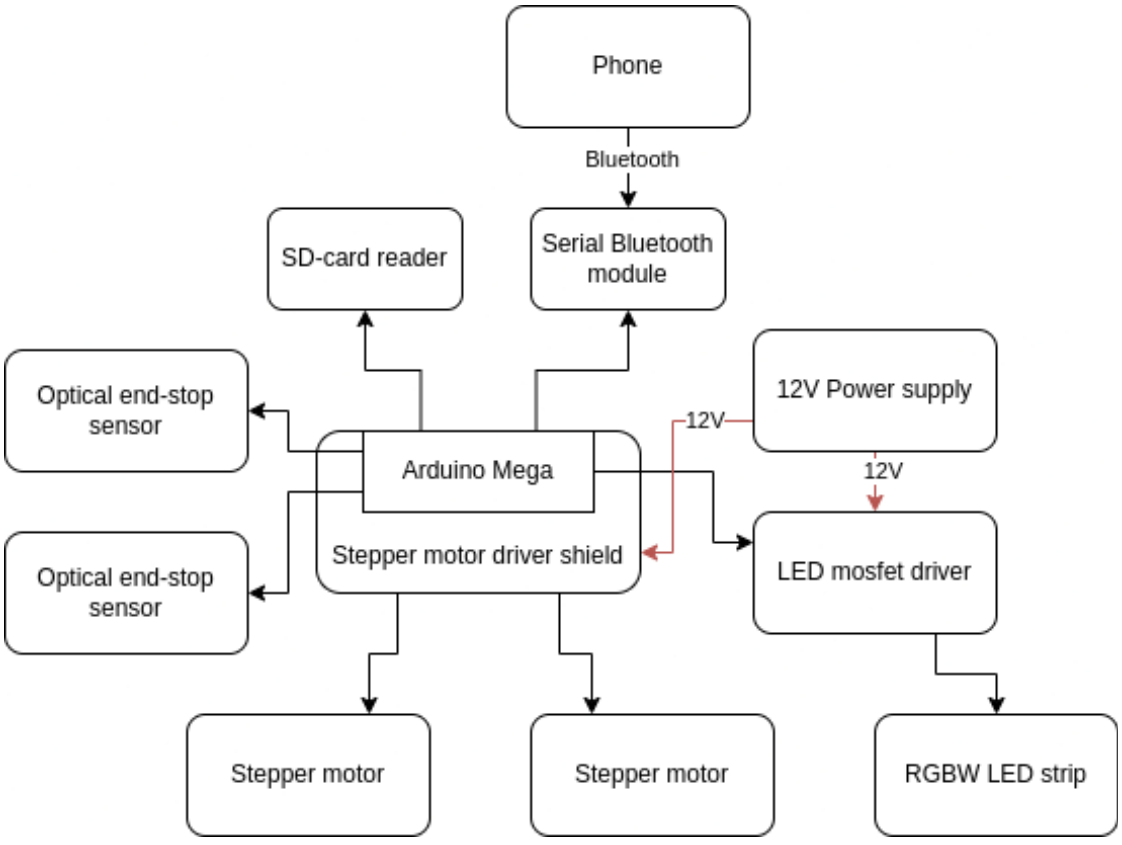

Illustration 3: System design Version 2

3.3 Version 3

Version 3 used a Django web server[6] for user interface which offered support for any device that has a web browser installed. Django was also chosen because of framework familiarity. This upgrade meant bringing back Raspberry PI for the design but also using Arduino to control stepper motors.

Arduino Mega was switched to Arduino Nano, because of size difference and unused pins. The CNC shield was also switched to V4[7] because it has Arduino Nano support and 3 stepper motor driver places instead of 4. At this time, the Raspberry PI Pico with the RP2040 chip had just been released, hence the use of the Arduino Nano RP2040 version.

This version required an extra DC-to-DC converter to convert 12 volts to 5 volts for the Raspberry PI. All things considered, this version was perfect for the project use.

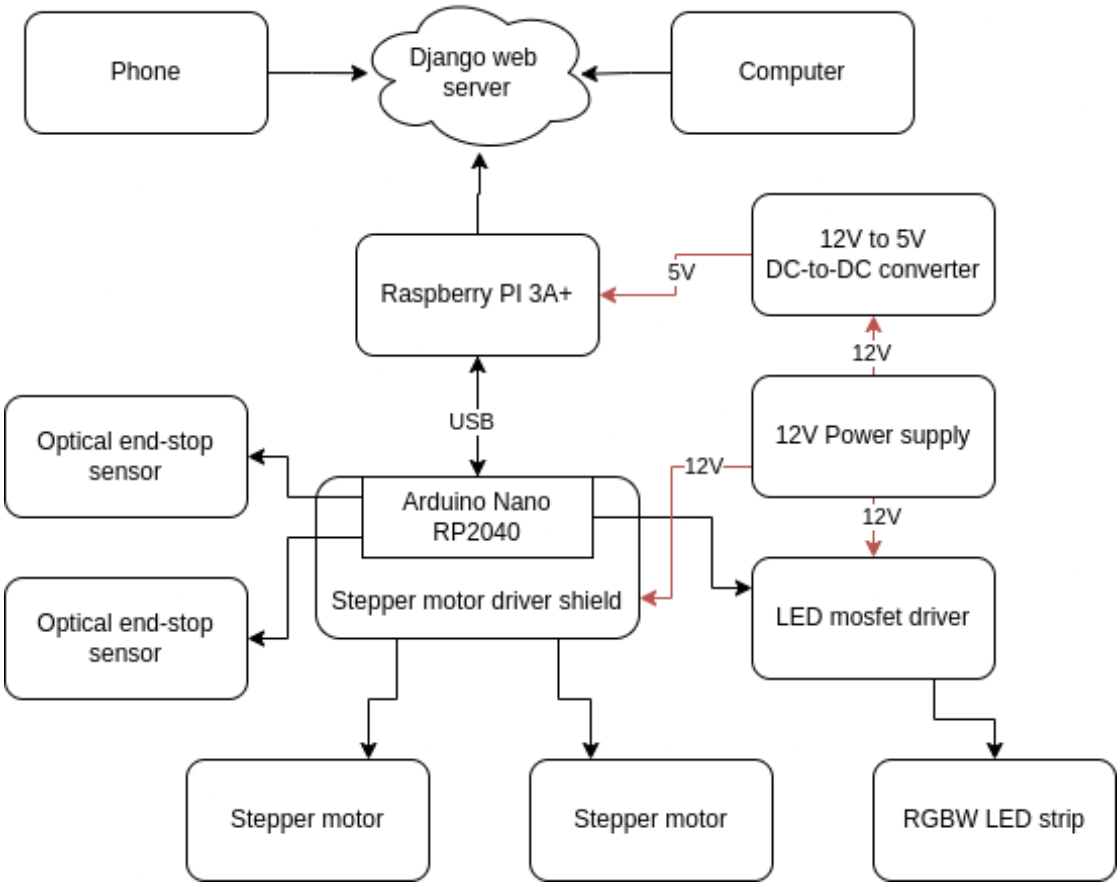

Illustration 4: System design Version 3

4 Mechanics Design

The 3D design followed Rob Dobson's SandBot[8](Illustration 5) model which is designed to be a scara-type robot arm with 360 degrees of freedom on each joint. This design was chosen for its low cost since it could be 3D printed and didn't require any expensive parts like rails. The initial design was a near replica of the SandBot, except the motor mounts were also 3D printed and the axis running through the middle was an aluminium rod. This was designed using Fusion360[9].

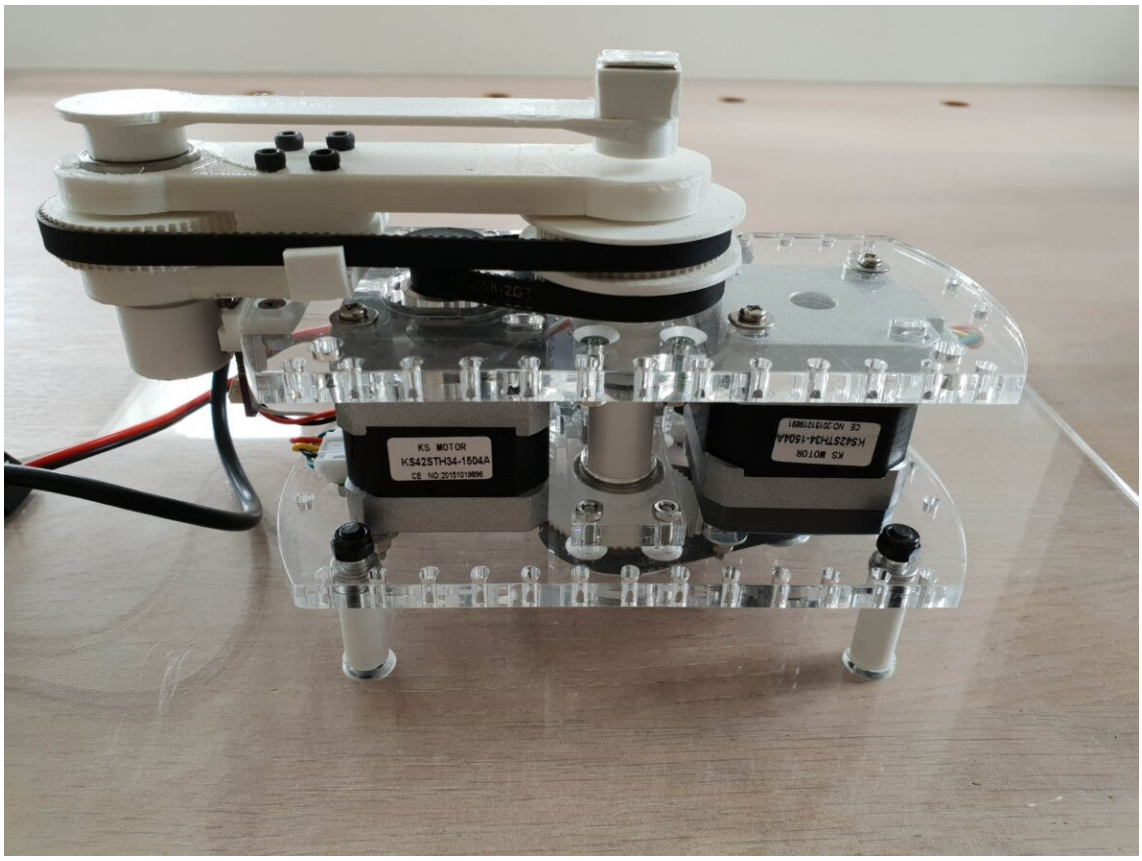

Illustration 5: Rob Dobson's SandBot

This design had a flaw of the arm bending down by the belt of the second arm, meaning the magnet would not touch the surface all the time. This was solved by doubling the first arm and running the second arm belt between the first arm(Illustration 6).

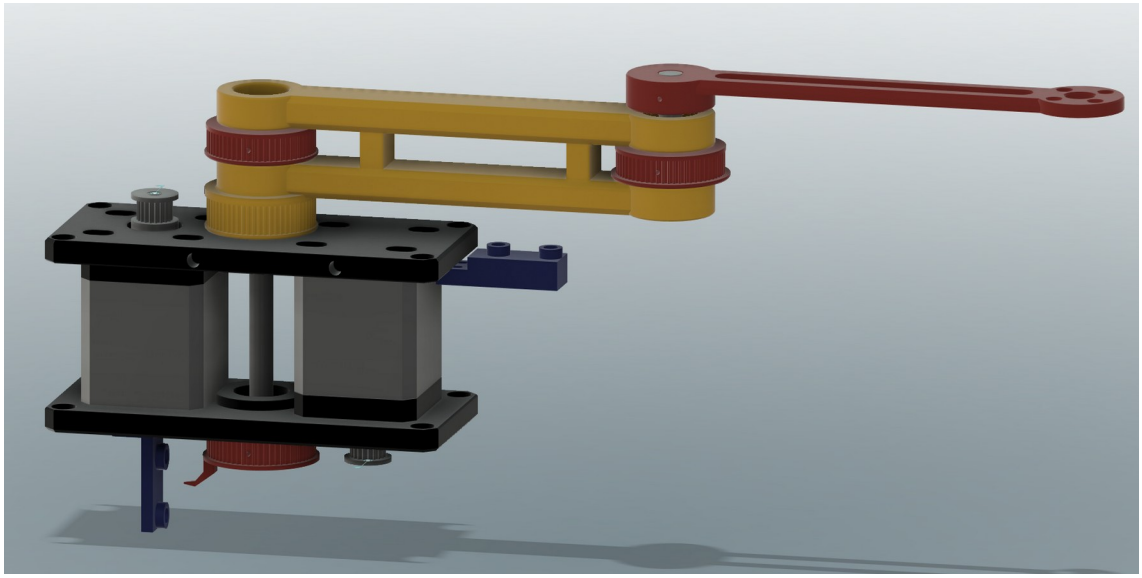

Illustration 6: Final 3D design

An issue with the new design was that the upper bearing on the middle axis needed to be ceramic so that the magnet would not be affected by the bearing. Another issue was the position of the end stop sensors. This was solved by having bigger holes where the sensor holder connects so it has room to wiggle for calibration.

The robot has a working area of 360 degrees with a radius of 282mm. Each arm was calibrated to be 141mm since there was no tensioner for the timing belt and that was the best tension The gear ratio of 1:3 was chosen for its simplicity and the gears not becoming too big. Height calibration was implemented by having four long screws on each corner which connect to the table itself and allows the user to screw nuts up and down to raise a corner.

Illustration 7: Robotic arm inside the table

5 Electronics

The electronics needed to be easily available and cheap enough for a student's budget. Note that this project was built before the Raspberry PI shortage.

5.1 Robotic arm electronics design

The electronics for the robotic arm had to have the following requirements:

- A microcontroller that can run two motors, an LED strip, two sensors, and communicate with the Raspberry PI
- Motors that can move indefinitely
- Drivers for the two motors
- Sensors for moving motors to the initial position
- LED strip and LED driver

Stepper motors with NEMA 14 form factor were chosen for this project, because they are cheap, can run continuously, are widely available, and are precise. The downside of stepper motors is that they require fairly high voltage and a driver board. But stepper motor drivers are cheap and are supported with many carrier boards for different controllers. For the drivers, TMC2208 modules were used for their quiet stepping mode and ability to microstep. Microstep functionality allows the motor to divide one step into multiple steps to gain accuracy but lose torque.

The LED strip chosen was a 12-volt RGBW (Red, Green, Blue, White) strip. This offered an easy control with a PWM (Pulse Width Modulation) signal from the controller. The downside is the need for driver hardware, which was solved by building a custom MOSFET array for a cheap solution.

The Arduino family has an abundance of support from the community that is creating software libraries, hardware modules and shields, and custom third-

party boards. Arduino Nano had a small form factor and enough pins to control all the connected hardware and a community-made CNC shield[7] that can drive up to 3 stepper motors with the ability to connect external stepper motor drivers. The shield was able to be connected straight to 12 volts with the ability to convert the voltage to Arduino voltage level.

For the sensors, optical end-stop sensors were perfect by providing a clean digital signal when activated. The sensors work by sending a light signal from one end and sensing that light signal from another end and activating by something blocking the light from reaching the sensor.

5.2 Web server electronics design

The web server only had the requirement of being able to run the Django web server wirelessly and able to communicate with the Arduino. This was done by using Raspberry PI 3A+ for its low price and ease of use. The Raspberry offered a Linux operating system possibility to run the web server and wireless connectivity to the router for networking.

5.3 Power supply

The power supply had the requirements of providing enough current for the whole system and providing the right voltages for each component. Arduino with the carrier board and the LED strip both required 12 volts to operate while the Raspberry PI ran on 5 volts. This was solved by choosing a 12-volt power supply and a DC-to-DC buck converter to convert 12 volts to 5 volts.

6 Software

6.1 Web Software

The control of the hardware was done through the Django web server version 4.2.5 running on the Raspberry PI. The web server had the following requirements:

- Functionality to upload .thr (polar coordinate track) files to the web server and save the track data in a database.
- The server needs the capability to convert the uploaded .thr files into image (.png) format
- There should be an ability to play and queue tracks from the database.
- The web server should be able to communicate with Arduino through a USB serial.
- There should be functionality to control the color and brightness of the LEDs or to choose LED fade tracks.
- The web server should provide functionality to adjust settings on Arduino.

The implementation of the web server's front end was started by sketching out the needed web pages for the home screen, LED control, tracklist, and options. These pages were written with HTML and CSS. The home page shows the current track playing and play/pause/stop control. The LED control was implemented using iro.js[10] API version 5.5.2 to generate one color wheel for red, green, and blue control and two sliders for white and RGB brightness. This page sends a dictionary of values to the backend which then sends commands to the Arduino which controls LEDs.

The tracklist page contains the tracks in the database in a list format with generated pictures. This page contains another tab to visualize the queued tracks and a button to clear the queue. The page has an "upload" button which

takes the user to the file upload page. Only .thr files are accepted, other file types are ignored.

The settings page contains options to:

- Set LED fade
- Move the robot arm to the home position
- Calibrate the home position
- Update coordinates from the database to the Arduino
- Stop motors
- Poweroff the Raspberry PI
- Slider to change the motor speed
- Slider to change the LED fade speed
- Slider to change the LED fade intensity
- Slider to change the LED fade saturation

The backend was written with Python version 3.9.2 and used SQLite[11] version 3.34.1 as a database. When uploading a file to the server, it checks if it is a .thr file and rejects other types. The file gets then passed to a Python script that uses matplotlib[12] version 3.7.2 to plot the coordinates into a graph and saves it as a .png file(Table 1). Both the .thr and the .png files get saved into a specified folder and the path to them gets uploaded into a list in the database.

Table 1: thr2png.py

```
from matplotlib import pyplot as plt
import sys

# Creates a .png file from .thr file
def drawFile(file):
    # Craetes a base with almost no borders
    plt.figure(figsize=(20, 20), frameon=False)
```

```
    ax = plt.axes([0, 0, 1, 1], projection='polar')
    # Turns off grids and axis markings
    plt.axis('off')
    plt.grid(visible=None)

    # Inverting the y axis and turining the graph 90 degrees, because .thr does
this
    ax.set_theta_direction(-1)
    ax.set_theta_zero_location("N")

    with open(file) as f:

        theta = []
        rho = []
        counter = 0
        for line in f:
            # discard if line commented
            if "#" in line or len(line) < 5 or "/" in line:
                continue

            else:
                comp = line.split()
                theta.append(float(comp[0]))
                rho.append(float(comp[1]))
                counter += 1


        plt.polar(theta, rho, marker='o', color='blue', markersize=1, linewidth=3)
        # Draws a green dot as starting point
        plt.polar(float(theta[0]), rho[0], marker='o', color='green', markersize=30)
        # Draws a red dot as end point
        plt.polar(float(theta[-1]), rho[-1], marker='o', color='red', markersize=30)

        name = file.split('.')
        plt.savefig(name[0]+'.png')

        return counter
```

Communication between Raspberry PI and the Arduino was implemented through a USB cable with a custom Python script(Table 2) using the pyserial[13] version 3.5 library. This was used to send coordinates and commands over.

Table 2: writeSerial.py

```
import serial, time
import struct, sys

def split_float(f):
        # Convert a float to a bytes object
        float_value = f
        bytes_value = struct.pack('f', float_value)

        # Convert the bytes object to a list of integers
        int_values = [b for b in bytes_value]

        return int_values

def split_int(d):
        # Convert a float to a bytes object
        value = d
        bytes_value = struct.pack('i', value)

        # Convert the bytes object to a list of integers
        int_values = [b for b in bytes_value]

        return int_values

def startSerial():
    # Try to start serial
    try:
        global ser
        ser = serial.Serial('/dev/ttyACM0', 115200, timeout=0.0002)
        ser.reset_input_buffer()
```

```
        time.sleep(1)
    except:
        print("Failed to start serial connection")

def resetBuffer():
    ser.reset_input_buffer()

def writeToSerial(command, data):
    try:

        # Check if serial is not open
        try:
            ser
        except:
            startSerial()

        byte = []
        time.sleep(0.01)

        # Checks if data is a float or int
        if isinstance(data, float):
            byte = split_float(data)
        elif isinstance(data, int):
            byte = split_int(data)

        b = [command, len(byte)] + byte
        # print(b)

        ser.write(bytes(b))
    except:
        print("Failed to write to serial")
```

```
def waitForResponse():
    # Loops while no input
    while ser.inWaiting() < 1:
        pass

    input = ser.readline()
    return input.decode('UTF-8')

def readSerial():
    # Loops while no input
    while ser.inWaiting() < 1:
        pass

    input = ser.readline()
    return input.decode('UTF-8')
```

When a track gets played, it launches another Python script that opens the specified .thr file and reads coordinates line by line ignoring comment lines. It calculates the distance between the current position and the next position, if it is too long, it divides it into a list of coordinates to smooth long straight lines(Table 3).

Table 3: divideCoords.py

```
import math


# Distance * length_of_the_arms to get units
# 0.03 seems a good distance
def checkDistance(coord1, coord2, distance):
    # Check if the line point are in the outer edge
    if coord1[1] > 0.97 and coord2[1] > 0.97:
        return False
```

```
    # Changing polar coords to X and Y
    x1, y1 = polarToXY(coord1[0], coord1[1])
    x2, y2 = polarToXY(coord2[0], coord2[1])

    # Getting the change of coords
    deltaX = x2 - x1
    deltaY = y2 - y1

    # Getting the distance between the points
    d = math.sqrt(deltaX**2 + deltaY**2)

    # If distance is over twice the wanted distance
    if d > distance * 2:
        return True
    else:
        return False

# Change from polar to cartesian
def polarToXY(theta, r):
    x = r * math.cos(theta)
    y = r * math.sin(theta)

    return x, y

# Changes from cartesian to polar
def xyToPolar(x, y):
    r = round(math.sqrt(x**2 + y**2), 5)
    th = round(math.atan2(y, x), 5)

    return th, r
```

```
def divideBy(coord1, coord2, distance):
    # Changing polar coords to X and Y
    x1, y1 = polarToXY(coord1[0], coord1[1])
    x2, y2 = polarToXY(coord2[0], coord2[1])

    # Getting the change of coords
    deltaX = x2 - x1
    deltaY = y2 - y1

    # Getting the distance between the points
    d = math.sqrt(deltaX**2 + deltaY**2)

    # Calculating how many points it divides to
    num_points = int(d / distance)

    # If number of point is 0 then exit
    if num_points == 0:
        return False

    # Calculating the distance between points
    r = d / num_points

    # Getting the angle from point A to point B
    theta = math.atan2(deltaY, deltaX)

    # Starting the coord list with point A
    new_coords = [[x1, y1]]

    # Calculating all the points between
    for i in range(0, num_points):
        x_new = new_coords[-1][0] + r * math.cos(theta)
        y_new = new_coords[-1][1] + r * math.sin(theta)
```

```
    new_coords.append([x_new, y_new])

# And adding the point B
new_coords.append([x2, y2])

coords_polar = []

# Changing the points to polar
for i in new_coords:
    th, r_new = xyToPolar(i[0], i[1])
    coords_polar.append([th, r_new])

    if th == coord2[0]:
        coords_polar.append(coord2)
        break

return coords_polar
```

The user interface turned out simple and clear how to control the functionality. It might not be the prettiest, but it gets the jobs done.

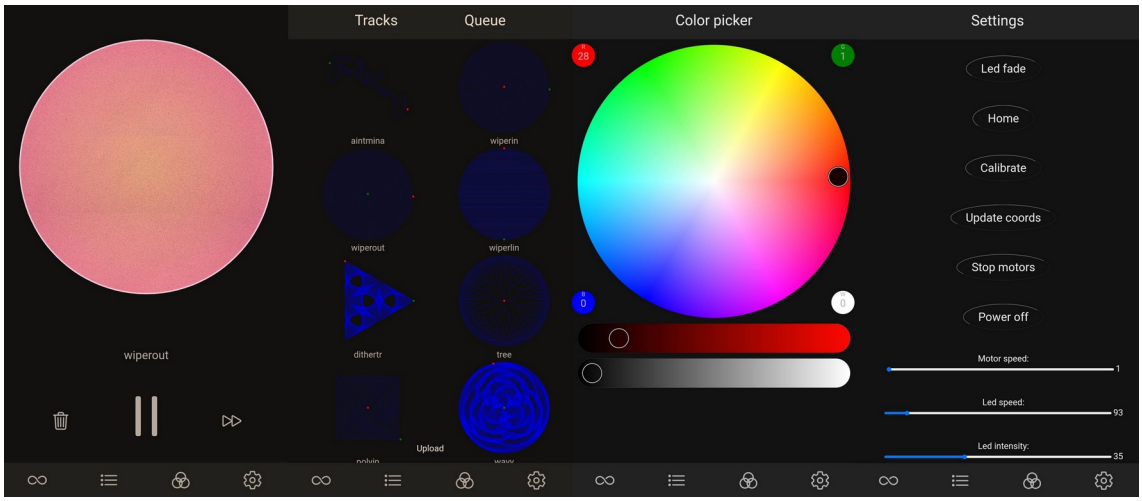

Illustration 8: Four main pages of the web server

6.2 Firmware

Firmware for Arduino nano RP2040 was written using the Pico-SDK[14] framework version 1.5.0 and C++ programming language. The firmware had the following requirements:

- The first core needs to listen to coordinates through a USB serial communication.
- Combine floating point values from 4 bytes.
- Check if the coordinates are valid.
- Pass coordinates to the second core.
- The second core needs to calculate angles with inverse kinematics.
- It needs to be able to calculate how many steps the motors need to be moved and move the motors accordingly.

- Then calculate the position where the motors actually moved and pass it to the first core.
- The first core listens for coordinates from the second core.
- Answer through serial where the robotic arm actually moved.
- Listen to commands.
- Drive the LEDs and iterate fade if required.

This was implemented by first testing out USB serial communication and the communication between the two cores. Communication between cores used a FIFO(First In, First Out) buffer which the Pico-SDK framework had functions to use. USB data transfer uses a function to split and combine integers and floating point values to arrays for byte transfer(Table 4).

Table 4: main.cpp - Serial communication

```
/*
        Read serial input
*/
uint16_t read_serial(uint8_t *buffer) {
        uint16_t buffer_index = 0;
        while (1) {
                int c = getchar_timeout_us(100);
                if (c != PICO_ERROR_TIMEOUT &&  buffer_index <
BUFFER_LENGTH) {
                        buffer[buffer_index++] = c;
                }
                else {
                        break;
                }
        }
        return buffer_index;
```

```
}

/*
        Combine n number of bytes to float
*/
float combine_float_bytes(uint8_t *bytes) {
        uint32_t value = 0;
        int n = bytes[0];

        for (int i = 1; i <= n; i++) {
                value |= bytes[i] << (8 * (i - 1));
        }

    return *reinterpret_cast<float*>(&value);
}

/*
        Combine n number of bytes to int
*/
int combine_int_bytes(uint8_t *bytes) {
        int result = 0;
        int n = bytes[0];

        for (int i = 1; i <= n; i++) {
                result |= static_cast<int32_t>(bytes[i]) << ((8 * (i - 1)));
        }
        return result;
}

/*
        Splits float into bytes for serial
*/
```

```
void split_float_to_bytes(float value, unsigned char* bytes) {
        unsigned char* int_bytes = reinterpret_cast<unsigned char*>(&value);

        for (int i = 0; i < 4; i++) {
                bytes[i] = *(int_bytes + i);
        }
}
```

Next was to write the stepper motor driver and test the motors. This was as simple as toggling the driver STEP pin back and forth for a single step with a 1 millisecond delay between the toggles. The driver had an ENA pin to enable torque on the motors and a DIR pin which set the direction of the steps. The driver also had a function to read the end stop sensors(Table 5).

Table 5: driver.cpp - Motor object

```
/*
        Constructor for the motor object
*/
motor::motor(char dir_pin, char step_pin, char enable_pin, char sensor_pin) {
    dir = dir_pin;
    enable = enable_pin;
    step = step_pin;
    sensor = sensor_pin;

    // Initiating the pins
    gpio_init(dir);
    gpio_init(enable);
    gpio_init(step);
    gpio_init(sensor);
    gpio_set_dir(dir, GPIO_OUT);
    gpio_set_dir(enable, GPIO_OUT);
```

```
    gpio_set_dir(step, GPIO_OUT);
    gpio_set_dir(sensor, GPIO_IN);
}

/*
        Checks the input from the sensor
*/
bool motor::sensorCheck() {
    int val = gpio_get(sensor);
    if (val) {
        return true;
    }
    else if (!val) {
        return false;
    }
}

/*
        Disables the motor
*/
void motor::Stop() {
    gpio_put(enable, 1);
}

/*
        Enable the motor
*/
void motor::Enable() {
    gpio_put(enable, 0);
}

/*
```

```
        Sets the direction pin and enables the motor
*/
void motor::setDirection(int steps) {
    if (steps >= 0) {
        gpio_put(dir, 1);
    }
    else if (steps < 0) {
        gpio_put(dir, 0);
    }
}

/*
        Step function
*/
void motor::Step(int steps, float stepdelay) {
    for (int i = 0; i < abs(steps); i++) {
        gpio_put(step, 1);
        sleep_ms(stepdelay);
        gpio_put(step, 0);
        sleep_ms(stepdelay);
    }
}
```

The driver also needed to move the motors simultaneously, so ”dualSteps” and ”equalSteps” functions were created. The function "dualSteps" drives two motors concurrently with different ratio stepping. The function ”equalSteps” moves the motors in equal steps at the same time. To move the motors to the origin(coordinate 0,0), it needed a homing function that moved the motors until the arm activated the end stop sensors(Table 6).

Table 6: driver.cpp - Driving two motors

```
/*
      Function to divide the steps
      motor1 needs to have more steps!
*/
void dualSteps(int step1, motor motor1, int step2, motor motor2, float speed) {
  int v = step1 / step2;      // Variable to figure out step ratio
  float v_f = float(step1) / float(step2);
  v_f -= float(v);          // Variable to figure out the remainder of step ratio
  int counter = 0;
  int step2_counter = 0;
  float extra_counter = 0.0;

  for (int i = 0; i < step1; i++) {
    motor1.Step(1, speed);
    counter++;
    if ((counter >= v) && (step2_counter < step2)) {
      if (extra_counter > 1.0) {
        extra_counter -= v;
        continue;
      }
      motor2.Step(1, speed);
      step2_counter++;
      counter = 0;
      extra_counter += v_f;
    }
  }

  // Checking if step2 did all the steps
  while (step2_counter < step2) {
    motor2.Step(1, speed);
```

```
        step2_counter++;
    }
}

/*
        Moves both of the motors at the same time
*/
void equalSteps(int step, motor motor1, motor motor2, float speed) {
    for (int i = 0; i < step; i++) {
        motor1.Step(1, speed);
        motor2.Step(1, speed);
    }
}

/*
        Moves the motors in order until signal from sensor
*/
void home(motor motor1, motor motor2) {
    motor1.setDirection(1);
    while (!(motor1.sensorCheck())) {
        motor1.Step(1, 1);
    }

    motor2.setDirection(1);
    while (!(motor2.sensorCheck())) {
        motor2.Step(1, 1);
    }

    motor1.setDirection(-1);
    motor1.Step(100, 5);

    motor1.setDirection(1);
```

```
    while (!(motor1.sensorCheck())) {
        motor1.Step(1, 10);
    }

    motor2.setDirection(-1);
    motor2.Step(100, 5);

    motor2.setDirection(1);
    while (!(motor2.sensorCheck())) {
        motor2.Step(1, 10);
    }
}
```

Implementing the math for the robotic arm required solving inverse kinematics for a 2-link planar robot system(Illustration 9) with the following[15]:

1. $\theta_2 = \cos^{-1}\left(\frac{d^2 - l_1^2 - l_2^2}{2 \times l_1 \times l_2}\right)$ where $d^2 = x^2 + y^2$ or r^2 in polar coordinates

2. $\theta_1 = \tan^{-1}\left(\frac{y}{x}\right) - \tan^{-1}\left(\frac{l_2 \times \sin(\theta_2)}{l_1 + l_2 \times \cos(\theta_2)}\right)$

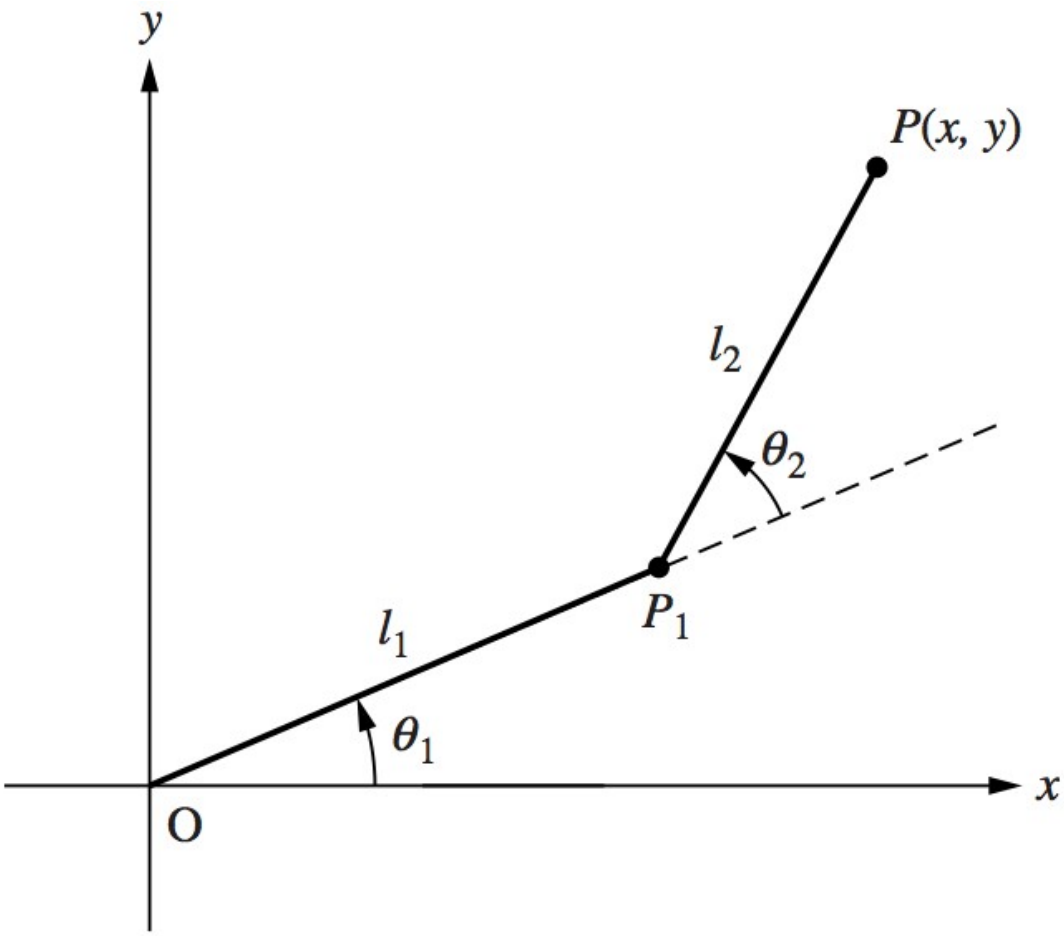

Illustration 9: 2-link planar robot system

The implementation into code was simple enough by creating the ”polarGetTheta2” function that calculates θ_2 from polar coordinates and the ”polarGetTheta1” function that calculates θ_1 from θ_2 and polar coordinates(Table 7).

Table 7: Inverse Kinematics in code

```
float polarGetTheta2(float theta, float r) {
    if(r == 0) {
        return M_PI;
    }
```

```
    // Math (had to break it up to multiple variables)
    float s1 = pow(r, 2) - 0.5;
    float q2 = acos(s1 / 0.5);

    return q2;
}

/*
        Get angles for arm 1 from polar coordinates
*/
float polarGetTheta1(float theta2, float theta, float r, bool inverted, float
theta1_old) {
    if(r == 0) {
        return theta1_old;
    }

    // Checking if denominator is zero
    float s2 = 0.5 + 0.5 * cos(theta2);
    if (s2 == 0.0) {
        s2 = 0.000001;
    }

    // Math
    float s1 = 0.5 * sin(theta2);
    float alpha = atan(s1 / s2);
    float q1 = 0;

    if(!inverted) {
        q1 = theta - alpha;
    }
    else {
        q1 = theta + alpha;
```

```
    }

    return q1;
}
```

Getting stepper motor steps required some additional steps. First, it needed to calculate the change in angles for both motors from the previous angle. Then it had to calculate how many steps the stepper motor needed to take taking into account SPR (steps per revolution) and microstepping(Table 8).

Table 8: Calculating change angles and steps

```
/*
        Calculating change of angles
*/
float deltaAngles(float theta1, float theta_old) {
    float delta_theta = theta1 - theta_old;

    while (delta_theta > 5) {
        delta_theta -= (2*M_PI);
    }
    while (delta_theta < -5) {
        delta_theta += (2*M_PI);
    }

    return delta_theta;
}

/*
        Calculating steps from delta angles
*/
int steps(float theta, int microstepping) {
```

```
    if (theta == 0.00000) {
        return 0;
    }

    unsigned long steps_per_revolution = 600 * microstepping;
    float s1 = (steps_per_revolution / (2 * M_PI));
    int step = round(s1 * theta);

    return step;
}
```

The model design meant that every time arm1 moved, arm2 moved in the opposite direction the same amount arm1 moved. Hence it was needed to account for this movement in code(Table 9).

Table 9: Accounting for arm1 movement

```
// Accounting the arm2 spin
step2 += step1;
```

To make sure that the robot does not lose its position over time, this movement was needed to be done in reverse. First by getting the joint angles from the steps taken and saving them for the next movement and then calculating polar coordinates from the joint angles(Table 10).

Table 10: Forward Kinematics in code

```
/*
        Calculating new arm angle from steps taken
*/
float thetaFromSteps(int step, int microstepping) {
    if (step == 0) {
        return 0.0;
```

```
    }

    unsigned int steps_per_revolution = 600 * microstepping;
    float theta = step / (steps_per_revolution / (2 * M_PI));

    return theta;
}

/*
        Caluculating the angle from arm angles
*/
float thetaFromArms(float q1, float q2) {
    float x = 0.5 * cos(q1) + 0.5 * cos(q1 + q2);
    float y = 0.5 * sin(q1) + 0.5 * sin(q1 + q2);

    if(x == 0 && y == 0) {
        return 0.0f;
    }

    float theta = atan2(y, x);

    return theta;
}

/*
        Calculating th r from arm angles
*/
float rFromArms(float q1, float q2) {
    float x = 0.5 * cos(q1) + 0.5 * cos(q1 + q2);
    float y = 0.5 * sin(q1) + 0.5 * sin(q1 + q2);

    if(x == 0 && y == 0) {
```

```
    return 0.0f;
  }

  float r2 = pow(x, 2) + pow(y, 2);

  return sqrt(r2);
}
```

The complete use of these functions can be seen in the following main.cpp code block(Table 11). The use of two cores in this case is a bit excessive since it blocks the execution of code when communicating, but it allows a coordinate queue system to be implemented in the future.

Table 11: Complete stepper motor code

```
        // Getting angles for the arms
        float theta2 = polarGetTheta2(angle, r);
        float theta1 = polarGetTheta1(theta2, angle, r, inverted, theta1_old);

        // Getting the change of angles
        float delta_theta1 = deltaAngles(theta1, theta1_old);
        float delta_theta2 = deltaAngles(theta2, theta2_old);

        // Getting the steps needed
        int step1 = steps(delta_theta1, microstepping);
        int step2 = steps(delta_theta2, microstepping);

        // Accounting the arm2 spin
        step2 += step1;

        // Setting the direction of the motors
        motor1.setDirection(step1);
```

```
            motor2.setDirection(step2);

            // Moving the motors
            if ((abs(step1) > abs(step2)) && (step2 != 0)) {
                dualSteps(abs(step1), motor1, abs(step2), motor2,
draw_speed);
            }
            else if ((abs(step1) < abs(step2)) && (step1 != 0)) {
                dualSteps(abs(step2), motor2, abs(step1), motor1,
draw_speed);
            }
            else if (abs(step1) == abs(step2)) {
                equalSteps(abs(step1), motor2, motor1, draw_speed);
            }
            else if (step2 == 0 && step1 != 0) {
                motor1.Step(abs(step1), draw_speed);
            }
            else if (step1 == 0 && step2 != 0) {
                motor2.Step(abs(step2), draw_speed);
            }

            // Get arm angles from the steps
            theta1_old += thetaFromSteps(step1, microstepping);
            theta2_old += thetaFromSteps(step2 - step1, microstepping);

            // Check if angles are over 2PI
            if (theta1_old >= (2*M_PI)) {
                theta1_old -= (2*M_PI);
            }
            else if (theta1_old <= -(2*M_PI)) {
                theta1_old += (2*M_PI);
            }
```

```
angle_old = thetaFromArms(theta1_old, theta2_old);
r_old = rFromArms(theta1_old, theta2_old);
```

7 Closing chapter

The thesis presented the theory and implementation of developing a robotic arm for a kinetic sand art coffee table project that can draw patterns on sand. The mechanics were quickly explained to use scara-type robotic arm which was 3d printed to save on cost. The electronics and the system design chapters covered how the best electronics system were chosen out of three prototypes.

The main topic was the software implementation that concentrated on developing the firmware for the Arduino that controls the motors in the robotic arm. This explained how the robotic arm can be moved with inverse kinematics and stepper motors.

The goal to engineer a working solution was successfully achieved. Ignoring the few mechanical issues not covered by the thesis, the project turned out to work perfectly and was quiet enough to not be annoying.

References

1: Sisyphus kickstarter, Bruce Shapiro, 2016, https://www.kickstarter.com/projects/1199521315/sisyphus-the-kinetic-art-table/description (Accessed: 20.03.2024)

2: Sisyphus Industries, , 2016, https://sisyphus-industries.com/ (Accessed: 20.03.2024)

3: Stepper Motor HAT for Raspberry Pi, , , https://www.waveshare.com/stepper-motor-hat.htm (Accessed: 20.03.2024)

4: Arduino CNC shield V3, Bertus Kruger, 2015, https://blog.protoneer.co.nz/arduino-cnc-shield/ (Accessed: 20.03.2024)

5: Serial Bluetooth Terminal, Kai Morich, , https://play.google.com/store/apps/details?id=de.kai_morich.serial_bluetooth_terminal&hl=en_US&gl=US&pli=1 (Accessed: 20.03.2024)

6: Django web server, Adrian Holovaty, Simon Willison, 2005, https://www.djangoproject.com/ (Accessed: 20.03.2024)

7: Arduino CNC shield V4, , , https://wiki.keyestudio.com/Ks0152_keyestudio_CNC_Shield_V4 (Accessed: 20.03.2024)

8: Rob Dobson's sandbot, Rob Dobson, 2017, https://robdobson.com/2018/08/a-new-sandbot/ (Accessed: 20.03.2024)

9: Fusion360, , , https://www.autodesk.com/products/fusion-360/overview?term=1-YEAR&tab=subscription (Accessed: 20.03.2024)

10: iro.js, James, , https://github.com/jaames/iro.js (Accessed: 20.03.2024)

11: sqlite3, D. Richard Hipp, 2000, https://www.sqlite.org/ (Accessed: 20.03.2024)

12: matplotlib, Michael Droettboom, et al., 2003, https://matplotlib.org/ (Accessed: 20.03.2024)

13: pyserial, Chris Liechti, 2003, https://pypi.org/project/pyserial/ (Accessed: 20.03.2024)

14: Pico-SDK, , , https://www.raspberrypi.com/documentation/pico-sdk/ (Accessed: 20.03.2024)

15: Inverse Kinematics, Jon Woolfrey, 2018, https://www.youtube.com/watch?v=RH3iAmMsolo (Accessed: 20.03.2024)