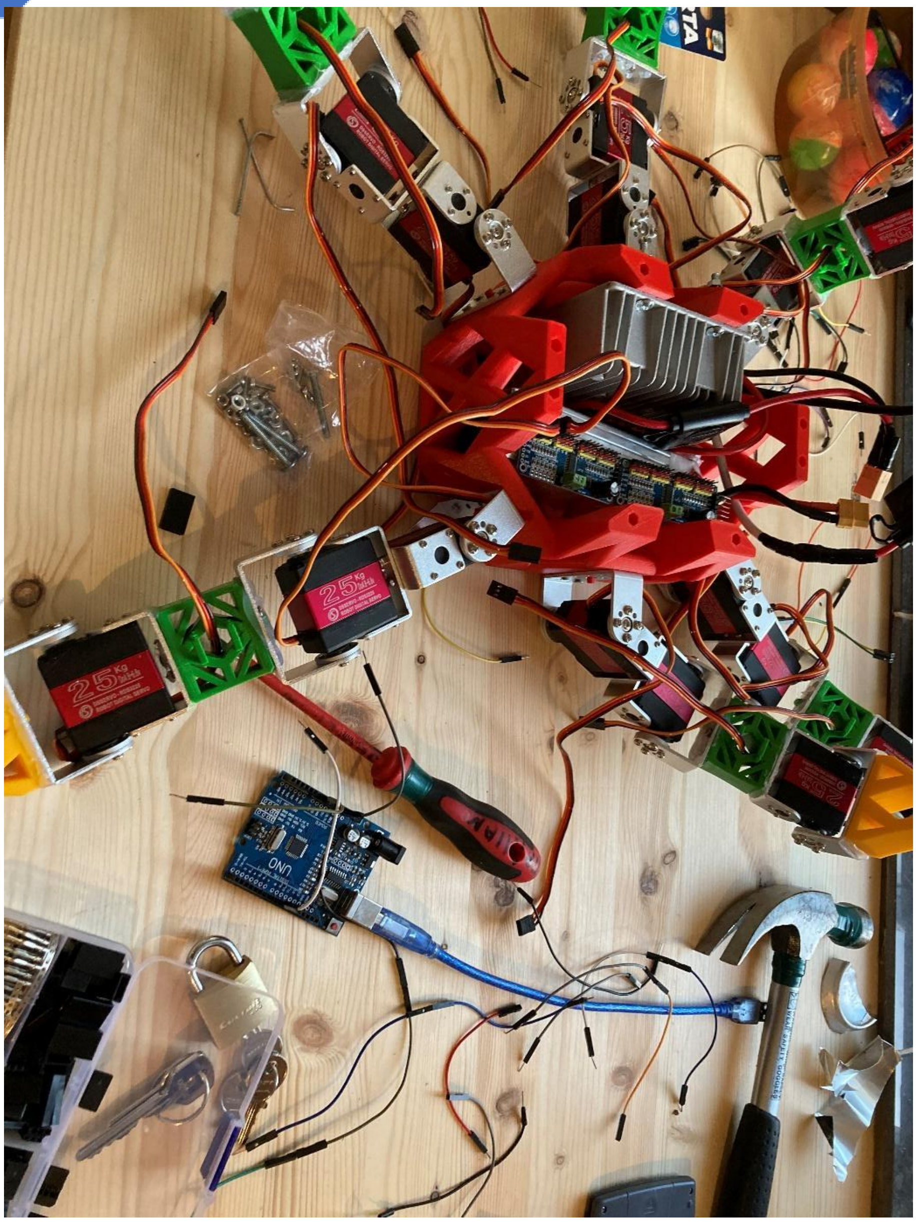
Robot Hexapod

Docs rev1

10.01.2023

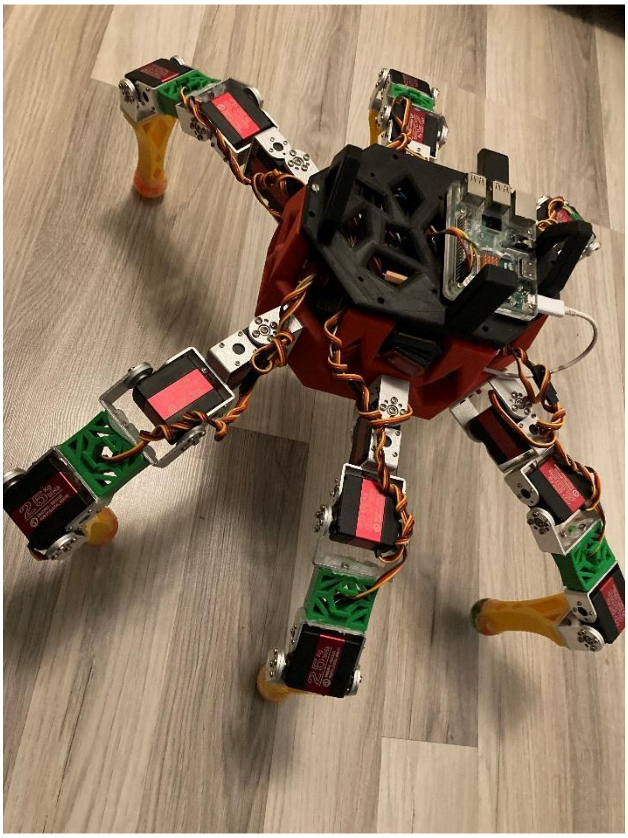


By Damian Widzisz

Contents

Introduction

This is a document that explains how my and my brother's hexapod robot works, specific information about electronics, mathematics, and programming.



The robot is a six-legged spider-like robot made for fun by my brother and me. It was created only for learning and entertaining purposes. The code and everything regarding the robot has an MIT license which means that everything is open source, so if you want to create your robot, you might find this document helpful. Or just if you want to know how it works.

The fun thing about this robot is that it doesn't have any sensors directly connected to the PC. Every movement is done 100% by math.

My brother left me alone to figure out how to program the robot and which controller to use, how to wire everything, and so on. Many parts at home could be used for it. I think the configuration I chose is pretty good.

I wanted to figure out as much mathematics alone as possible, but at the beginning, I didn't know how to start. It is my first big coding project, so I learned how to move one leg using inverse kinematics on the internet—the rest of the math I have done all by myself.

I finished the project over a year and a half ago, but recently I decided to create documentation on it. If I worked on this project now, I would do it very differently and make a better-optimized code and pretty much everything much better. I will go over some ideas for improving the design in the documentation.

You can also find a simulation code in this project's GitHub repository. How it works and how it is helpful is explained there.

Here is the link to it: https://github.com/DaJMaN4/Hexapod-robot-with-raspberry-pi

The link to YouTube videos of this robot walking:

This is all for the introduction. I hope it will be fun to read.

Mathematics

The mathematics chapter is about explaining in an easy way how all the math that makes the robot walk works.

I will explain how to move a single leg, make the robot move in a chosen direction, and make the robot move in all possible directions using a signal from a joystick.

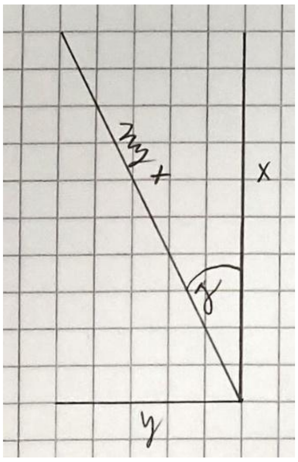
As I said in the introduction, a big part of mathematics I have come to by myself, I didn't try to make it as optimized as possible but made it work. There are many ways to improve and lower the CPU power needed to calculate movements. So it is not the best program possible.

At the end of the chapter, I will reflect on my thoughts and give you some ideas for improving math.

Moving a leg in x and y coordinates

Mathematics I used to make this robot move is called inverse kinematic. It is a branch in mathematics used for determining an angle between at least two lines knowing the length of all lines and coordinates of the endpoint (the endpoint of at least two lines). This is mainly used for animations and robotics.

Knowing this is not very important if you want to make an arm or a leg move. You can find many libraries that do it for you, and robots from many companies include software that makes programming easy. Usually, all you need to know is how to use it, but if you want to know how the mathematics under movement work, read the text below.



$$\gamma = a\tan(\frac{x}{y})$$

$$new\,x = \sqrt{x^2 + y^2}$$

The first thing that a computer must calculate is gamma (a Greek letter similar to “y “). The gamma is the angle of the servo that moves the leg to the right and left.

You can choose how to say in which direction gamma will move (right or left). I will go deeper in the conclusion of this chapter. Here, I will just explain how I have done it.

The x is the coordinate that says how far straight from the center of the mentioned servo is the endpoint of the leg. How far the robot will reach.

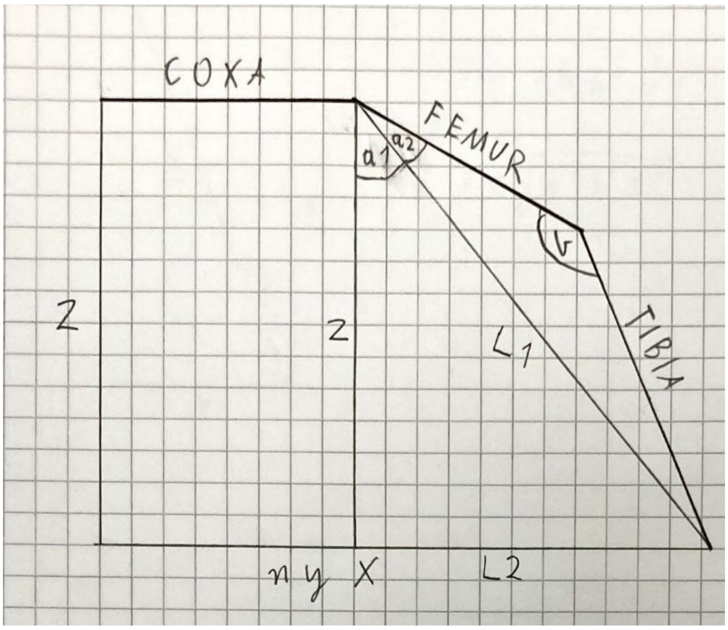
Y coordinate can be positive and negative, but it cannot be equal to zero because, in my later calculations, I will divide by y. If y is positive, the leg is moving in one direction, and if it is negative, it will move in the other direction. The bigger the numbers, the farther from the center the endpoint will be (in millimeters).

Combine the two coordinates, and you get 2D visualization of the leg, which is a right triangle. It is easier to see if you move the y line between ny x and x at the top. Using trigonometry, you can find the value of angel gamma using the inverse tangent function.

The last thing I have done is to find new x (ny x). It is necessary because the following calculations will use x; as you might see, the new x is longer than the x. Remember that x is the distance that the endpoint will move to. So, imagine there is no new x, and x is a constant. By changing the angle y, the endpoint will not move just to the right. It will go right and down, moving like a fragment of an ellipse, not a straight line. So, to make the leg move in straight lines, x must be corrected. I do it using Pythagoras.

Moving a leg in z and x coordinates

Here, I will explain how to move a robotic arm/leg in x and z coordinates.



$$L2 = ny\,x - coxa$$

$$L1 = \sqrt{L2^2 + z^2}$$

$$a2 = a\cos\left(\frac{L^2 + FEMUR^2 - TIBIA^2}{2 \cdot TIBIA \cdot FEMUR}\right)$$

$$a1 = a\tan\left(\frac{z}{L1}\right)$$

$$b = a\cos\left(\frac{TIBIA^2 + FEMUR^2 - L^2}{2 \cdot TIBIA \cdot FEMUR}\right)$$

My robot's lengths of legs and coordinates are in millimeters, but any measurement system can be used.

The constants in the robot are the length of the joints: coxa, femur, and tibia. Between them are servos that move the leg. The figure above shows a graphic representation of a robotic leg in z and x coordinates. Changing the coordinates will change the angle of the servos.

The first calculation is finding the length of L2. It's just ny x – coxa. Using Pythagoras and knowing the z coordinate and L2, I can find L1. Using inverse tangens, I find a1 by dividing z with L1 and reversing tangus.

All side lengths are known in the triangle with lines L1, tibia, and femur, but no angles. That’s why I used the cosine rule.

With it, we can find any angel in a triangle when we know the lengths of the sides. That’s how I find a2 and b. Now just add a1 and a2 together, and that's all.

In the code, it looks like this. The functions atan and acos calculates the value of angles as radian, so I convert it to degrees by multiplying the number result with 57.296. It must be converted because the function that moves servos takes as input degree. The subtraction of 90 at the end is for the servos to move correctly, my servos can move just 180 degrees, and the 0-degree position of a servo is when the angle between the femur and tibia is 90 degrees. In other words, degrees are rotated in - 90 degrees.

```
gamma = math.atan(x / y) * 57.296

L1 = x - coxa
L = math.sqrt(z**2 + L1**2)

a1 = math.acos(z/L) * 57.296
a2 = math.acos((L**2 + femur**2 - tibia**2)/(2*femur*L)) * 57.296
a = a1 + a2
b = math.acos((tibia**2 + femur**2 - L**2)/(2*tibia*femur)) * 57.296 - 90
```

```python
def calculate():
    xc = x - coxa
    if xc < 0:
        gamma = math.atan((xc / -1) / y) * 57.296

        L1 = xc / -1
        L = math.sqrt(z ** 2 + L1 ** 2)

        a1 = math.asin(z / L) * 57.296
        a2 = math.acos((L ** 2 + femur ** 2 - tibia ** 2) / (2 * femur * L)) * 57.296
        a = a1 + a2 - 90
    else:
        gamma = math.atan(xc / y) * 57.296

        L1 = xc
        L = math.sqrt(z ** 2 + L1 ** 2)

        a1 = math.acos(z / L) * 57.296
        a2 = math.acos((L ** 2 + femur ** 2 - tibia ** 2) / (2 * femur * L)) * 57.296
        a = a1 + a2
    b = math.acos((tibia ** 2 + femur ** 2 - L ** 2) / (2 * tibia * femur)) * 57.296 - 90
    if gamma < 0:  # map() function
        gamma = (gamma - (-90)) * 90 / -90 + 90
        gamma = ((gamma - 90) / -1) + 90
    return gamma,a,b
```

The program's calculations are slightly different depending on whether the xc is positive or negative. That's to make moving legs under the coxa line and in minus coordinates possible. These movements aren't essential in this scenario, like in systems with other robots like Quadpods, but this allows the robot to do larger steps.

When x is smaller than coxa, after subtracting it with coxa, you will get a negative number, and then different calculations occur, the differences are:

x is divided by -1 to get the correct number from atan function. The value L1 is divided by -1 because it is always negative, but it must be positive to work in the calculation.

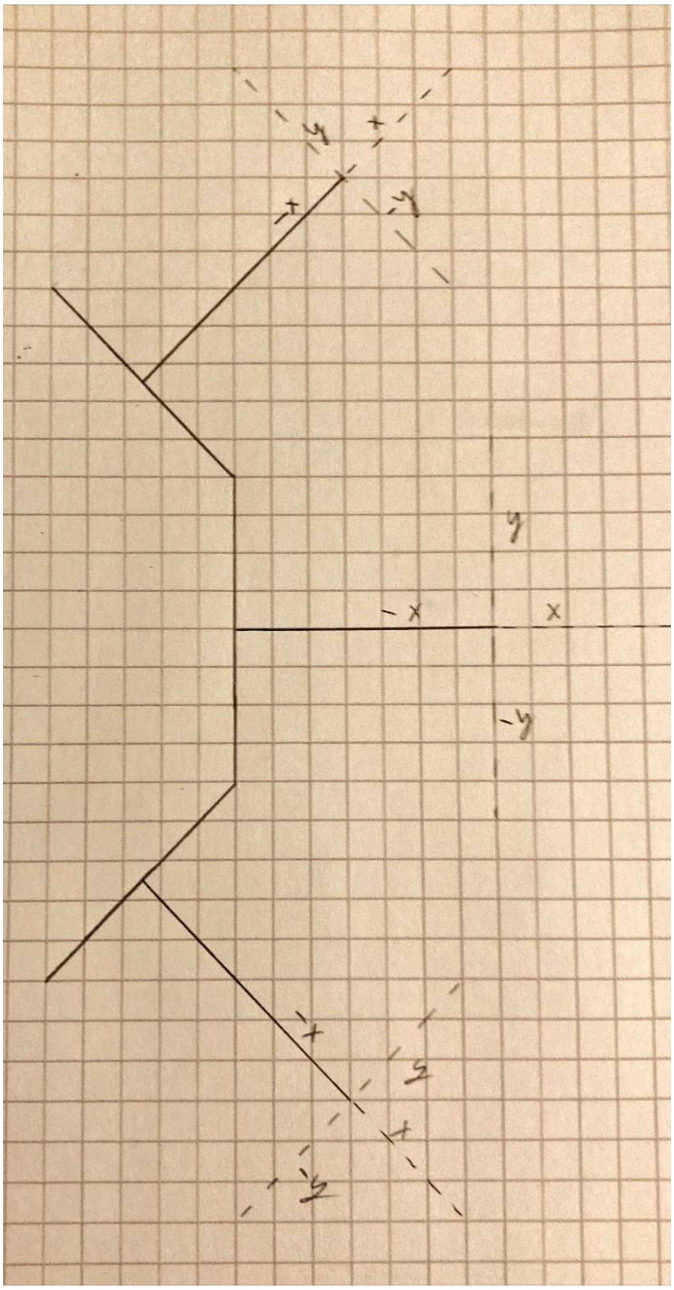
In the end, the beta angel is subtracted by 90 because of the positioning of a servo in the leg. For the same reason as before.

The last thing that happens is mapping the gamma value if it's less than 0. That occurs when y is negative, but it still is the same number as when y is positive and has the same length from 0. Precisely if y is 20 and x is 40 then gamma will be 63 degrees, if y is -20 and x 40, then it will be -63 degrees, but it should be positive and be bigger than 90, it should be the same length fra the 90 which is the initial gamma, the gamma that is when y is close to 0.

For getting the same difference between the initial gamma I used that function, so if the return is -63 that function will convert it to 117.

Making all the legs move

Here I will explain how to make all the legs move in the same direction; I will show an example of moving all legs such that the robot moves to the front and explain how to code walking.



The picture to the right shows the right side of the robot. The shape to the left represents the robot's main body, the lines are the legs of the robot, and little graphs represent coordinates for moving the legs in x and y axes, as explained before.

In this example, z has two states, up and down. While walking, always three legs are up, and three other legs are down on the floor. These two pairs of three legs move in opposite directions, just like your leg are moving opposite to each other while you walk.

The coordinates must change to make the robot go forward so that each leg on the floor moves at the same speed in the same direction.

Assume that n is a constant and follow this math these xx, yy xxx, and yyy are for moving the legs in their positions. xx is for +x, xxx for -x and so on.

$$xx = xx - n$$
$$yy = yy - n$$
$$yyy = yyy + n$$
$$xxx = xxx + n$$

The leg at the top gets yyy and xxx to move up, the one in the middle receives just yyy, and the lowest gets xx and yyy.

This way, all three visible legs move in the same direction with the same speed.

Now, mirror this for the other side, and all six legs will move in the same direction.

To make the robot walk, I made two definitions, each for one state of walking. The first state is when three legs are on the ground. Rest is in the air, the other definition just changes legs, so those on the ground are now in the air.

So, for the legs on the ground, use the method I told you before and calculate gamma, beta, and alfa. Move the legs in the air in the opposite direction. Now define the maximum value of xx and xxx. Each time it gets this value, change which legs are on the ground, and then change which numbers you subtract and add with n with each other. This makes the legs move in the opposite direction.

If you make the variable n bigger, the robot's speed will increase, but not its smoothness of walking, so it cannot be too big because then the robot will not be able to walk.

Moving the robot in all possible directions using signal from a joystick

The PC gets two 8-bit signals, one is x, and the other is y. Together they show where precisely the joystick is positioned. Here I will explain how to use the signal from a joystick to move the robot in all possible directions.

```
if event.code == 0:
    xh = (event.value / 255 - 0.5) * 2
    if xh > 0 and xh < 0.1:
        xh = 0
    elif xh < 0 and xh > -0.1:
        xh = 0
elif event.code == 1:
    yh = (event.value / 255 - 0.5) * 2
    if yh > 0 and yh < 0.1:
        yh = 0
    elif yh < 0 and yh > -0.1:
        yh = 0
```

event.code means which coordinate from the joystick has changed, x or y.

Then I refactor the value to get a number between -1 and 1 from the original 0 to 255. This makes later calculations easier. I also declare that if the value is between -0.1 and 0.1, it will be 0. That's to make the robot go in straight lines while directing the joystick somewhere to the right/left/forward/back. Because if the robot is moving in these directions, then the PC needs to calculate less, which means that the robot moves faster. That's also why I didn't do anything to make the robot move in all directions at the same speed because if I had done it, then moving not in the previously mentioned directions would be too shaky.

New values will be applied when the robot switch which legs are on the ground. I implemented that to prevent un-synchronizing.

```
if xx == 160 and xxx == 160 and xx2 == 160 and xxx2 == 160:
    yyh = yh
    xxh = xh
```

Then depending on values from the joystick, the robot will calculate and return lists of angles of all the legs, a – alfa, b – beta, and at the end, gamma. There are two such lists, one for movement in the x direction and one for the y direction, forward and to the side.

```
if 0 > yyh:
    if up == 1: #straight
        xx = xx - n
        yy = yy - n
        yyy = yyy + n
        xxx = xxx + n
    elif up == 2:
        xx = xx + n
        yy = yy + n
        yyy = yyy - n
        xxx = xxx - n
    alist1, blist1, gammalist1 = move()
elif 0 < yyh:
    if up == 1: #back
        xx = xx + n
        yy = yy + n
        yyy = yyy - n
        xxx = xxx - n
    elif up == 2:
        xx = xx - n
        yy = yy - n
        yyy = yyy + n
        xxx = xxx + n
    alist1, blist1, gammalist1 = move()
```

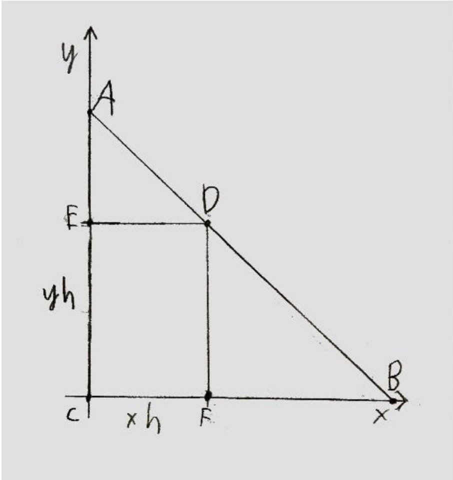
```
if 0 > xxh:
    if up == 1: #left
        xx2 = xx2 - n
        yy2 = yy2 - n
        yyy2 = yyy2 + n
        xxx2 = xxx2 + n
    elif up == 2:
        xx2 = xx2 + n
        yy2 = yy2 + n
        yyy2 = yyy2 - n
        xxx2 = xxx2 - n
    alist2, blist2, gammalist2 = move2()
elif 0 < xxh:
    if up == 1: #right
        xx2 = xx2 + n
        yy2 = yy2 + n
        yyy2 = yyy2 - n
        xxx2 = xxx2 - n
    elif up == 2:
        xx2 = xx2 - n
        yy2 = yy2 - n
        yyy2 = yyy2 + n
        xxx2 = xxx2 + n
    alist2, blist2, gammalist2 = move2()
```

```python
if gammalist1 != [] or gammalist2 != []:
    number = 0
    yyyh = yh
    xxxh = xh
    if xxxh < 0:
        xxxh = xxxh/-1
    if yyyh < 0:
        yyyh = yyyh/-1
    if xxxh == yyyh:
        h = 0.5
    elif yyyh == 0:
        h = xxxh
    elif xxxh == 0:
        h = yyyh
    else:
        if xxxh < yyyh:
            h = yyyh / (yyyh + xxxh)
        elif xxxh > yyyh:
            h = xxxh / (yyyh + xxxh) / -1 + 1
```

If yyh and xhh are not equal to zero, the user does want to move forward/back/right/left.

When that happens, both gammalist1 and gammalist2 will not be empty. New variables yyyh and xxxh will be used for calculation here. If they are negative, then the function will convert them to positive.

Worth noticing that yh is yyyh, the same as xh and xxxh.



In the drawing to the right. The whole picture represents the point to which the robots will move, which is D and shows yh and xh.

The next step is creating variable h, the relationship between the length line AD and line AB in a case when xxxh is bigger than yyyh. Otherwise, it will represent the relation between the lines DB and AD divided by -1 plus 1. More precisely, this is the relation between the output from calculation1 and calculation2. The h will always be a value between 0 and 1.

```python
number = 0
yyyh = yh
xxxh = xh
if xxxh < 0:
    xxxh = xxxh/-1
if yyyh < 0:
    yyyh = yyyh/-1
if xxxh < yyyh:
    h = (yyyh*math.sqrt(2))/(math.sqrt((yyyh*xxxh+(yyyh*yyyh)/2+(xxxh*xxxh)/2)*2)*math.sqrt(2))
elif xxxh > yyyh:
    h = (xxxh*math.sqrt(2))/(math.sqrt((yyyh*xxxh+(yyyh*yyyh)/2+(xxxh*xxxh)/2)*2)*math.sqrt(2))/-1 + 1
elif xxxh == yyyh:
    h = 0.5
if yyyh == 0:
    h = xxxh
elif xxxh == 0:
    h = yyyh
```

Previously I used this code instead. It does the same but is less optimized. As I was writing the documentation, I realized that what I wrote was very bad. So, I changed it. By the way, it is a good reminder that many things can be done more simpler than you might think at the beginning.

```python
if gammalist1 != [] and gammalist2 != []:
    while number != 6:
        outalist.append((alist1[number]-alist2[number])*h+alist2[number])
        outblist.append((blist1[number]-blist2[number])*h+blist2[number])
        outgammalist.append((gammalist1[number]-gammalist2[number])*h+gammalist2[number])
        number = number + 1
    kit2.servo[4].angle = outgammalist[0]
```

The last thing is applying the relation h in the function. This returns a number that is between x and y by h%. When h = 0%, the output is y, and when h = 100%, it is x.

You can think about it like one list moving robot in one direction and the other in the second direction. You can combine those lists and set them with an analog value to say how much it will go in each direction, for example, 20% forward and 80% to the right.

After the while is done, new positions are sent to all servos.

Conclusion

For me, the goal was to make the robot work, I only used a little bit of time on optimization, almost nothing, and that's wrong because if I had optimized it, the robot's speed would increase. So here I will explain in points what I would have done differently, and write some ideas on optimization, etc.

1. I could store the values of movement in straight/back/right/left directions and then just get them if I needed it instead of calculating them always when I needed them. I couldn't do that to move the robot in all possible directions because that would be impossibly much data to store.

2. Using NumPy to optimize the calculating of the servos would significantly reduce the needed computation consumption by the program.

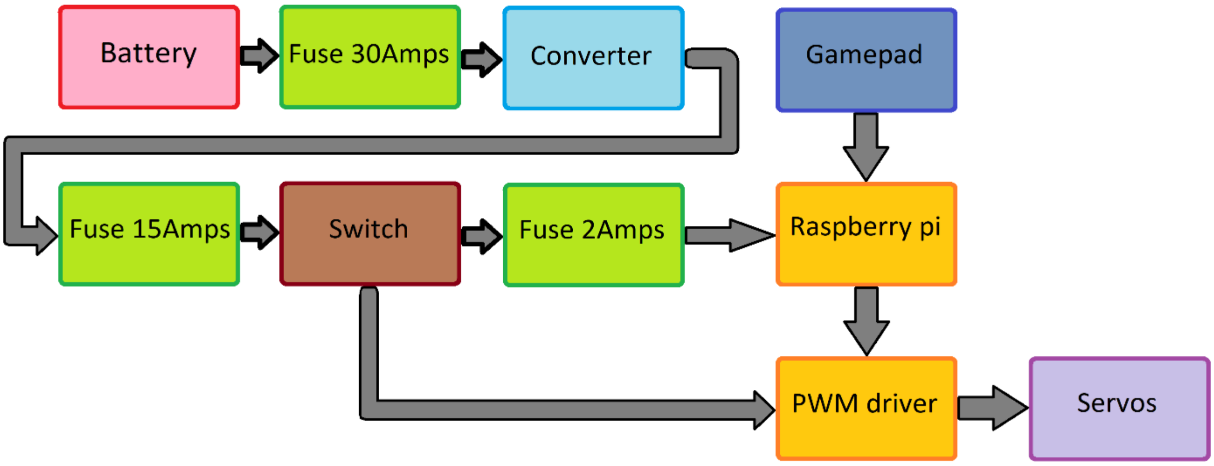
   Precisely I would create the lists as vectors in NumPy and perform all the computations as vectors

3. Draw everything that happens and try to find any shortcuts. There is a good probability that I missed some black swans.

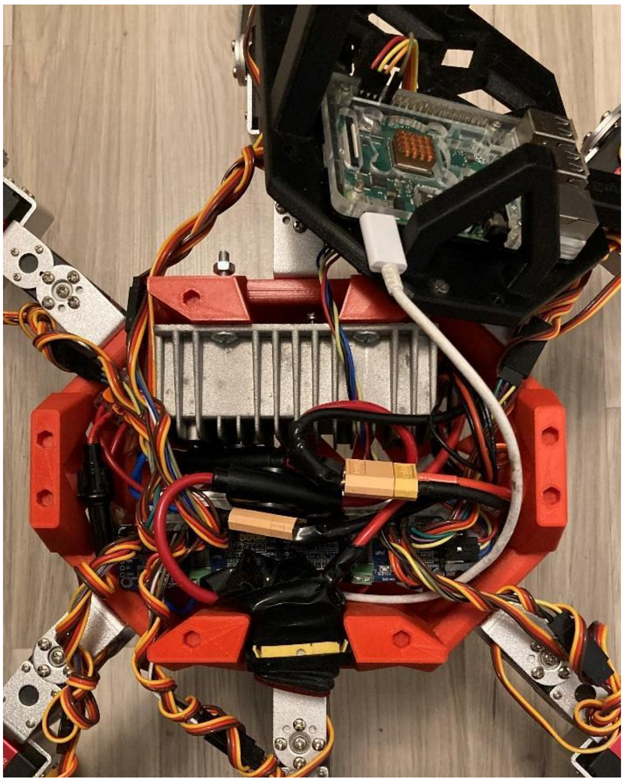
Circuit

Basic explanation

This is a circuit with low voltage. 4 cell lithium battery is connected to a converter that lowers the voltage to 5V. This voltage supplies a PWM driver and PC, which is a raspberry pi. All servos are connected to the PWM driver. The driver receives commands from the PC and moves the servos. Below is a much more detailed explanation with my thoughts on upgrades.



Detailed explanation

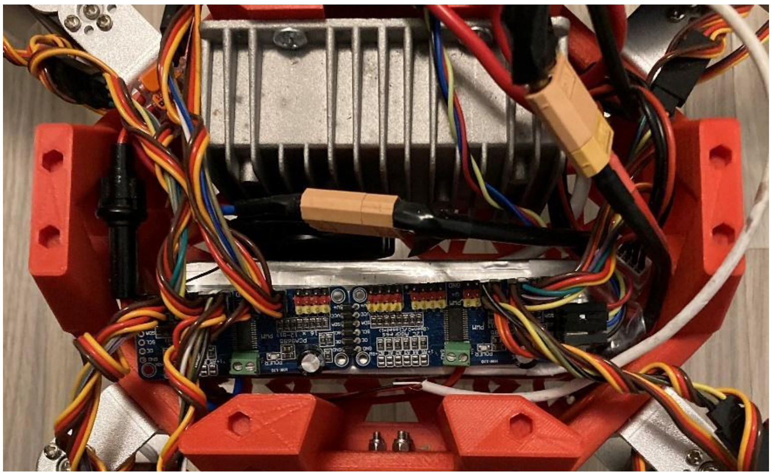


Four cell lithium-ion battery is protected by a 30-amp fuse and connected to the voltage converter. As you might know, this isn't the optimal solution. Using servos with the same nominal voltage as the battery is best. Then you don't need any converter, so the robot doesn't gain additional weight or lose power on converting voltage.

The battery is again protected by a fuse, with 15AMP between the converter and the connections to the PC and PWM driver. Peak current can be high because, at higher torques, servos draw a lot of power. If all the servos were on max torque, then the 30-amp fuse would easily melt, but they usually don't need much current.

If something is in the way of the servos, something that they can't push, they would use their maximum torque to try to make it and could destroy something or harm someone, even if it's your hand, they are rated for 25 kg cm, so it is not very safe for fingers. To make it safe, I would use different servos to send information back to the PC about its torque, temperature, etc. Then I could set the maximum torque on the servos, which made them save. But such servos are much more expensive. We used inexpensive servos.

The gamepad has a stop button that resets the whole program so that it doesn't send any new commands to servos but doesn't turn off the current. The only way to turn them off entirely is to switch the red button on the servo.



PC has its fuse, rated for 2AMP, and most probably, it is not needed and makes little difference, but I wanted to be sure that I couldn't destroy the PC somehow.

All servos have three wires, one for ground, one for voltage, and one for communication. Pretty standard for those types of servos. Communication is a one-way PWM signal. The controller says what position will all the servos have, but it doesn't even know if the servos are connected.

The controller is an adafruit PWM driver with 16 connections to servos. Another same controller can be connected to it in series by soldering them together, but then you need to give it another address by soldiering the address part of the board.

Which I2C addresses the controllers will have. The PC and controller communicate via the I2C communication protocol. The good thing about this protocol is that it can connect many devices to the same two wires, from PC to controllers, go four wires, two for I2C, one for 5V, and one for Ground.

The raspberry PI needs 5V in order to operate. A USB connection delivers it. I should have connected it to the supply pins at the raspberry pi, but I didn't know about them then.

Parts:

Amount	Name
1	LI-po 4 cell Battery
1	Converter 10-25V to 5V
1	Raspberry pi 3B+
1	Fuse 2A
1	Fuse 15A
1	Fuse 30A
2	PWM Driver
1	Switch
18	Servos
2	Fuse pockets

Docs about the PWM driver, this is a newer version, the older is not available anymore:

https://www.adafruit.com/product/815

Docs about the servos:

Program

The program is relatively complicated. It is my first real code, so you can see that many things could be done easier and better, more optimized, and so on. It has zero bugs and works well, but now as I possess more knowledge, I would have done it differently.

Block explanation

Going Through the Program

Libraries and variables

```
1  from adafruit_blinka.board.pyboard import X3
2  from board import I2C
3  from evdev import InputDevice, categorize, ecodes
4  from adafruit_servokit import ServoKit
5  kit = ServoKit(channels=16,i2c=None, address=0x40)
6  kit2 = ServoKit(channels=16,i2c=None, address=0x41)
7  import time
8  import math
9  import os
10 import sys
11 number = 0
12 for _ in range(6):
13     kit.servo[number].set_pulse_width_range(500, 2500)
14     number = number + 1
15 number = 4
16 for _ in range(12):
17     kit2.servo[number].set_pulse_width_range(500, 2500)
18     number = number + 1
19 gamepad = InputDevice('/dev/input/event0')
```

I first import libraries. I use: adafruit_blinka.board.pyboard – this is an extension to communicate with an adafruit driver.

I2C is the communication protocol to communicate with the driver.

Evdev is a library for reading inputs from a gamepad and pretty much any device.

adafruit_servokit is for setting the position of the servos in the driver.

In lines 5 and 6 I create the objects for drivers. There are two. Then I import standard libraries, time, and math. If you know some python which I assume you know, then I don't need to explain. And the last two (os, sys) I use just for restarting the program if anything goes wrong, like an emergency exit.

```
print(gamepad)
leftg = 310
rightg = 311
nalefttg = 308
nrightg = 309

#up = 307
down = 305
left = 304
right = 306

start = 313
select = 312

start_red = 316

clicjoystickleft = 314
clicjoystickright= 315

```

Next, I assign servos to the board while setting the pulse width range, which is assigning positions (the lowest and highest). If it is not correct, then, for example, 90 degrees in the program will not be 90 degrees in real life, or servos wouldn't move at full 180 degrees. Most servos have different pulse width ranges. The last line is assigning the connected device as a gamepad.

I print in the command line which device is connected because while I was testing, I connected the mouse and keyboard at the same time as the gamepad, so I needed to know what I am connected to. Worth noticing that evdev can be used to get an event on clicks and analog values not just from a gamepad but from many other devices.

Then I assigned integers to variables. They are addresses of buttons, so for example, button down has address 305.

```
coxa = 74
femur = 100
tibia = 162

x = 160
y = 0.1
z = 210

xx = x
yy = y
yyy = y
xxx = x
xx2 = x
yy2 = y
yyy2 = y
xxx2 = x

n = 1
```

Next, I assigned lengths of parts of the legs (coxa, femur, tibia). All legs are identical. All values are given in millimeters.

Then I set initial values for the x, y, and z coordinates. Variables xx, xx2, yy, yyy, and so on are for moving pairs of three legs because while moving, the robot has 3 legs on the ground and 3 in the air, and the legs must move in different directions. That's why there are 4 variables for every 3 legs. In the chapter mathematics, I explained why there are 4 variables.

n is the value of the length of movement from an initial point to a new point, it can be adjusted with a gamepad to make the robot move faster, but the bigger the value, the less smooth the movement is because it misses more and more steps which makes the robot move nicely.

```python
x_max = 215
up = 2

gammalist1 = []
alist1 = []
blist1 = []

gammalist2 = []
alist2 = []
blist2 = []

outalist = []
outblist = []
outgammalist = []

xh = 0
yh = 0

def zz():
    return z
def move():
    z = 210
```

x_max is the maximal length of movement in each direction. The value is the sum of normal x and the length of how much I want it to move. It's like a length of steps. So, the robot in each step will move x_max – x = 55mm.

Up says which three legs will be on the ground while moving. It normally is one or two, which indicates which three legs are on the ground. It can also be three, which means that all 6 legs are on the ground. I used it just for testing and demonstrations. Then I define lists to store data from the computation.

xh and yh are variables for analog signals from the joystick.

The zz function gets z from outside the function, there must be a better way of doing it, but when creating the program, I couldn't find a better one.

Calculations

Here is the calculate function. The chapter "Mathematics" explains how it works.

Each if represents a leg.

```python
def calculate():
    xc = x - coxa
    if xc < 0:
        gamma = math.atan((xc / -1) / y) * 57.296

        L1 = xc / -1
        L = math.sqrt(z ** 2 + L1 ** 2)

        a1 = math.asin(z / L) * 57.296
        a2 = math.acos((L ** 2 + femur ** 2 - tibia ** 2) / (2 * femur * L)) * 57.296
        a = a1 + a2 - 90
    else:
        gamma = math.atan(xc / y) * 57.296

        L1 = xc
        L = math.sqrt(z ** 2 + L1 ** 2)

        a1 = math.acos(z / L) * 57.296
        a2 = math.acos((L ** 2 + femur ** 2 - tibia ** 2) / (2 * femur * L)) * 57.296
        a = a1 + a2
    b = math.acos((tibia ** 2 + femur ** 2 - L ** 2) / (2 * tibia * femur)) * 57.296 - 90
    if gamma < 0:  # map() function
        gamma = (gamma - (-90)) * 90 / -90 + 90
        gamma = ((gamma - 90) / -1) + 90
    return gamma,a,b
```

```python
if up == 1 or up == 3:
    x = xx
    y = 0.1
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
else:
    x = xx
    y = 0.1
    z1 = zz()
    z = 191
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
    z = z1
if up == 2 or up == 3:
    x = xx
    y = 0.1
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
else:
    x = xx
    y = 0.1
    z1 = zz()
    z = 191
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
    z = z1
if up == 2 or up == 3:
    x = xxx
    y = yy
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
else:
    x = xxx
    y = yy
    z1 = zz()
    z = 191
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
    z = z1
if up == 1 or up == 3:
    x = xxx
    y = yyy
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
else:
    x = xxx
    y = yyy
    z1 = zz()
    z = 191
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
    z = z1
if up == 2 or up == 3:
    x = xxx
    y = yyy
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
else:
    x = xxx
    y = yyy
    z1 = zz()
    z = 191
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
    z = z1
if up == 1 or up == 3:
    x = xxx
    y = yy
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
else:
    x = xxx
    y = yy
    z1 = zz()
    z = 191
    gamma, a, b = calculate()
    gammalist1.append(gamma)
    alist1.append(a)
    blist1.append(b)
    z = z1
return alist1, blist1, gammalist1
```

The first two legs are the ones in the middle, on each side. They should have the y's value of 0 because y does not need to change in order to walk straight or back. It is not 0 but close to zero because in the calculation I divide by y, so instead, I set it to be a number close to 0, in this case, 0.1.

There comes a very similar definition, just the values of moving the legs are different, that one is responsible for moving to the front and back, and the second is for moving to the right and left

The other function has the same calculation definition.

```python
def move2():
    z = 210
    def calculate():
        xc = x - coxa
        if xc < 0:
            gamma = math.atan((xc / -1) / y) * 57.296

            L1 = xc / -1
            L = math.sqrt(z ** 2 + L1 ** 2)

            a1 = math.asin(z / L) * 57.296
            a2 = math.acos((L ** 2 + femur ** 2 - tibia ** 2) / (2 * femur * L)) * 57.296
            a = a1 + a2 - 90
        else:
            gamma = math.atan(xc / y) * 57.296

            L1 = xc
            L = math.sqrt(z ** 2 + L1 ** 2)

            a1 = math.acos(z / L) * 57.296
            a2 = math.acos((L ** 2 + femur ** 2 - tibia ** 2) / (2 * femur * L)) * 57.296
            a = a1 + a2
        b = math.acos((tibia ** 2 + femur ** 2 - L ** 2) / (2 * tibia * femur)) * 57.296 - 90
        if gamma < 0:  # map() function
            gamma = (gamma - (-90)) * 90 / -90 + 90
            gamma = ((gamma - 90) / -1) + 90
        return gamma,a,b
```

```
234     if up == 1 or up == 3:
235         y = yy2
236         x = math.sqrt(math.pow(y,2) + 25600)
237         gamma, a, b = calculate()
238         gammalist2.append(gamma)
239         alist2.append(a)
240         blist2.append(b)
241     else:
242         y = yy2
243         x = math.sqrt(math.pow(y,2) + 25600)
244         z1 = zz()
245         z = 191
246         gamma, a, b = calculate()
247         gammalist2.append(gamma)
248         alist2.append(a)
249         blist2.append(b)
250         z = z1
251     if up == 2 or up == 3:
252         y = yyy2
253         x = math.sqrt(math.pow(y,2) + 25600)
254         gamma, a, b = calculate()
255         gammalist2.append(gamma)
256         alist2.append(a)
257         blist2.append(b)
258     else:
259         y = yyy2
260         x = math.sqrt(math.pow(y,2) + 25600)
261         z1 = zz()
262         z = 191
263         gamma, a, b = calculate()
264         gammalist2.append(gamma)
265         alist2.append(a)
266         blist2.append(b)
267         z = z1
```

```
268     if up == 2 or up == 3:
269         x = xxx2
270         y = yyy2
271         gamma, a, b = calculate()
272         gammalist2.append(gamma)
273         alist2.append(a)
274         blist2.append(b)
275     else:
276         x = xxx2
277         y = yyy2
278         z1 = zz()
279         z = 191
280         gamma, a, b = calculate()
281         gammalist2.append(gamma)
282         alist2.append(a)
283         blist2.append(b)
284         z = z1
285     if up == 1 or up == 3:
286         x = xxx2
287         y = yy2
288         gamma, a, b = calculate()
289         gammalist2.append(gamma)
290         alist2.append(a)
291         blist2.append(b)
292     else:
293         x = xxx2
294         y = yy2
295         z1 = zz()
296         z = 191
297         gamma, a, b = calculate()
298         gammalist2.append(gamma)
299         alist2.append(a)
300         blist2.append(b)
301         z = z1
```

```
302     if up == 2 or up == 3:
303         x = xx2
304         y = yyy2
305         gamma, a, b = calculate()
306         gammalist2.append(gamma)
307         alist2.append(a)
308         blist2.append(b)
309     else:
310         x = xx2
311         y = yyy2
312         z1 = zz()
313         z = 191
314         gamma, a, b = calculate()
315         gammalist2.append(gamma)
316         alist2.append(a)
317         blist2.append(b)
318         z = z1
319     if up == 1 or up == 3:
320         x = xx2
321         y = yy2
322         gamma, a, b = calculate()
323         gammalist2.append(gamma)
324         alist2.append(a)
325         blist2.append(b)
326     else:
327         x = xx2
328         y = yy2
329         z1 = zz()
330         z = 191
331         gamma, a, b = calculate()
332         gammalist2.append(gamma)
333         alist2.append(a)
334         blist2.append(b)
335         z = z1
336 return alist2, blist2, gammalist2
```

Then comes the function for moving the robot in opposite directions. The biggest difference here is the change in values of x and y before calculating the angles.

One can see that in line 236 and some others instead of setting the value of x as xx or xx2 I used Pythagoras to get x

$x = \sqrt{y^2 + 25600}$ 25600 is 160 squared. 160 is the nominal x. The reason for doing so is explained in the chapter “Moving a leg in x and y coordinates” in “Mathematics”.

Controlling the robot

I will briefly explain the code, none of the mathematics because I have already done that.

Here I create the loop in which the sequence is repeating.

Gamepad.read.one() is a function that gives the newest event from a gamepad. It can be an analog value from a joystick or a digital value from pressing buttons. If there aren't any, then the function returns None.

If event != None – there is an event and

event.type == ecode.Ev_ADS, which is a code of analog events from joysticks.

```
while True:
    event = gamepad.read_one()
    if event != None and event.type == ecodes.EV_ABS:
        if event.code == 0:
            xh = (event.value / 255 - 0.5) * 2
            if xh > 0 and xh < 0.1:
                xh = 0
            elif xh < 0 and xh > -0.1:
                xh = 0
        elif event.code == 1:
            yh = (event.value / 255 - 0.5) * 2
            if yh > 0 and yh < 0.1:
                yh = 0
            elif yh < 0 and yh > -0.1:
                yh = 0
```

event.code is the address of coordinates. For example, code 0 is x value of joystick 1.

Then happens the calculation that I already explained.

```
    if xx == 160 and xxx == 160 and xx2 == 160 and xxx2 == 160:
        yyh = yh
        xxh = xh
    if event != None and event.type == ecodes.EV_KEY and event.value == 1:
        if event.code == leftg:
            n = n - 1
            if n < 0:
                n = 1
        elif event.code == rightg:
            n = n + 1
        elif event.code == start_red:
            os.execv(sys.executable, ['python'] + sys.argv)
        elif event.code == start:
            kit2.servo[4].angle = 90
            kit2.servo[5].angle = 0
            kit2.servo[6].angle = 0
            kit2.servo[7].angle = 90
            kit2.servo[8].angle = 0
            kit2.servo[9].angle = 0
            kit2.servo[10].angle = 90
            kit2.servo[11].angle = 0
            kit2.servo[12].angle = 0
            kit.servo[3].angle = 90
            kit.servo[4].angle = 0
            kit.servo[5].angle = 0
            kit.servo[0].angle = 90
            kit.servo[1].angle = 0
            kit.servo[2].angle = 0
            kit2.servo[13].angle = 90
            kit2.servo[14].angle = 0
            kit2.servo[15].angle = 0
```

The first if is for assigning the updated values for movement when x is at its nominal position, I did that because if the values were assigned at other points in the sequence then the midpoint in which xx, xxx, xx2, and xxx2 meet would change, and that's a big problem because then the robot does different lengths of steps.

The second checks if an event is a clicking, button.

The third if and the elif is simply for changing the constant of n that determines the length of the smallest movement of the leg, so by changing it , I can change the robot's speed at the cost of less smoothly walking.

The second elif is for resetting the program.

The third elif is for setting the values of servos in such positions that it is easier to carry the robot.

```
if 0 > yyh:
    if up == 1: #straight
        xx = xx - n
        yy = yy - n
        yyy = yyy + n
        xxx = xxx + n
    elif up == 2:
        xx = xx + n
        yy = yy + n
        yyy = yyy - n
        xxx = xxx - n
    alist1, blist1, gammalist1 = move()
elif 0 < yyh:
    if up == 1: #back
        xx = xx + n
        yy = yy + n
        yyy = yyy - n
        xxx = xxx - n
    elif up == 2:
        xx = xx - n
        yy = yy - n
        yyy = yyy + n
        xxx = xxx + n
    alist1, blist1, gammalist1 = move()
if 0 > xxh:
    if up == 1: #left
        xx2 = xx2 - n
        yy2 = yy2 - n
        yyy2 = yyy2 + n
        xxx2 = xxx2 + n
    elif up == 2:
        xx2 = xx2 + n
        yy2 = yy2 + n
        yyy2 = yyy2 - n
        xxx2 = xxx2 - n
    alist2, blist2, gammalist2 = move2()
elif 0 < xxh:
    if up == 1: #right
        xx2 = xx2 + n
        yy2 = yy2 + n
        yyy2 = yyy2 - n
        xxx2 = xxx2 - n
    elif up == 2:
        xx2 = xx2 - n
        yy2 = yy2 - n
        yyy2 = yyy2 + n
        xxx2 = xxx2 + n
    alist2, blist2, gammalist2 = move2()
```

This is the part of the code that activates the definitions that move the robot, it is just a matter of if yyh and xxh are not zero and then if they are positive or negative. It is explained in detail in the chapter “Making all the legs move” in the chapter “Mathematics”.

Then it returns lists with calculated angles of the servos.

```
if gammalist1 != [] or gammalist2 != []:
    number = 0
    yyyh = yh
    xxxh = xh
    if xxxh < 0:
        xxxh = xxxh/-1
    if yyyh < 0:
        yyyh = yyyh/-1
    if xxxh == yyyh:
        h = 0.5
    elif yyyh == 0:
        h = xxxh
    elif xxxh == 0:
        h = yyyh
    else:
        if xxxh < yyyh:
            h = yyyh / (yyyh + xxxh)
        elif xxxh > yyyh:
            h = xxxh / (yyyh + xxxh) / -1 + 1
```

If both lists are not empty, then the part of math that is responsible for moving the robot in all possible directions and not just straight/back/left/right, how it works is explained it detail in the chapter “Moving the robot in all possible directions” in the chapter “Mathematics”.

```
if gammalist1 != [] and gammalist2 != []:
    while number != 6:
        outalist.append((alist1[number]-alist2[number])*h+alist2[number])
        outblist.append((blist1[number]-blist2[number])*h+blist2[number])
        outgammalist.append((gammalist1[number]-gammalist2[number])*h+gammalist2[number])
        number = number + 1
    kit2.servo[4].angle = outgammalist[0]
    kit2.servo[5].angle = outalist[0]
    kit2.servo[6].angle = outblist[0]
    kit2.servo[7].angle = outgammalist[1]
    kit2.servo[8].angle = outalist[1]
    kit2.servo[9].angle = outblist[1]
    kit2.servo[10].angle = outgammalist[2]
    kit2.servo[11].angle = outalist[2]
    kit2.servo[12].angle = outblist[2]
    kit.servo[3].angle = outgammalist[3]
    kit.servo[4].angle = outalist[3]
    kit.servo[5].angle = outblist[3]
    kit.servo[0].angle = outgammalist[4]
    kit.servo[1].angle = outalist[4]
    kit.servo[2].angle = outblist[4]
    kit2.servo[13].angle = outgammalist[5]
    kit2.servo[14].angle = outalist[5]
    kit2.servo[15].angle = outblist[5]
    alist1.clear()
    blist1.clear()
    gammalist1.clear()
    alist2.clear()
    blist2.clear()
    gammalist2.clear()
```

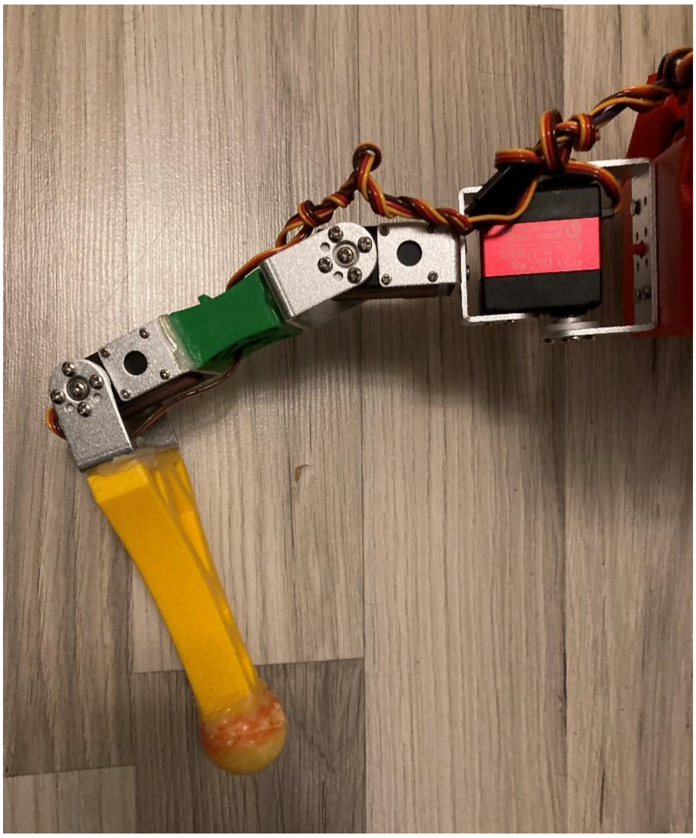
After the calculations, new values are sent to the servos, and all the lists are cleared.

```
        elif gammalist1 != []:
            kit2.servo[4].angle = gammalist1[0]
            kit2.servo[5].angle = alist1[0]
            kit2.servo[6].angle = blist1[0]
            kit2.servo[7].angle = gammalist1[1]
            kit2.servo[8].angle = alist1[1]
            kit2.servo[9].angle = blist1[1]
            kit2.servo[10].angle = gammalist1[2]
            kit2.servo[11].angle = alist1[2]
            kit2.servo[12].angle = blist1[2]
            kit.servo[3].angle = gammalist1[3]
            kit.servo[4].angle = alist1[3]
            kit.servo[5].angle = blist1[3]
            kit.servo[0].angle = gammalist1[4]
            kit.servo[1].angle = alist1[4]
            kit.servo[2].angle = blist1[4]
            kit2.servo[13].angle = gammalist1[5]
            kit2.servo[14].angle = alist1[5]
            kit2.servo[15].angle = blist1[5]
            alist1.clear()
            blist1.clear()
            gammalist1.clear()
        elif gammalist2 != []:
            kit2.servo[4].angle = gammalist2[0]
            kit2.servo[5].angle = alist2[0]
            kit2.servo[6].angle = blist2[0]
            kit2.servo[7].angle = gammalist2[1]
            kit2.servo[8].angle = alist2[1]
            kit2.servo[9].angle = blist2[1]
            kit2.servo[10].angle = gammalist2[2]
            kit2.servo[11].angle = alist2[2]
            kit2.servo[12].angle = blist2[2]
            kit.servo[3].angle = gammalist2[3]
            kit.servo[4].angle = alist2[3]
            kit.servo[5].angle = blist2[3]
            kit.servo[0].angle = gammalist2[4]
            kit.servo[1].angle = alist2[4]
            kit.servo[2].angle = blist2[4]
            kit2.servo[13].angle = gammalist2[5]
            kit2.servo[14].angle = alist2[5]
            kit2.servo[15].angle = blist2[5]
            alist2.clear()
            blist2.clear()
            gammalist2.clear()
        outalist.clear()
        outblist.clear()
        outgammalist.clear()
    if xx >= x_max or xxx >= x_max or xx2 >= x_max or xxx2 >= x_max:
        if up == 1:
            up = 2
        elif up == 2:
            up = 1
```

The last thing is assigning the values if just one list is empty after assigning lists are cleared.

The last if is for changing which three legs are on the ground. When the step is completed it is activated and the legs change their position in z axes.

Improvements in design and thoughts



General design, choosing components, and designing components for a 3d printer were done by my brother. Still, while making this robot walk, I found that the design needs some improvements to make the walking smooth.

The important thing to have in such a robot are spheres at the end of the legs, the point of having them is

when it is a sphere, then you can set the endpoint in the center of it, and all possible with the ground will have the same distance from the endpoint which makes it go smoother that if you have squares at the end the endpoint will always have a different value. It is harder for the robot to walk because it must fight the unsmooth surface of the foot.

When it is a sphere, you can set the endpoint in the middle of it and all possible touching with the ground will have the same distance from the endpoint, making it go smoother.

In the beginning, everything was connected together with glue, which is not the best connection. So I needed to make holes in the plastic and connect everything with screws instead because sometimes something fell apart. If that happened during a demonstration to someone, I would be screwed.

Conclusion

The robot was a fun project, and I learned much about robotics. At this point, I feel pretty confident about programming and designing any robot as long as it doesn't involve using advanced AI. In the future, I want to learn AI, and then any robotic project will not be scary to me.

In the future, i will probably design some types of robots.

I learned fundamentals about inverse kinematic on this website:
https://oscarliang.com/inverse-kinematics-implementation-hexapod-robots/