Line Drawer

Introduction

The line drawer is a project I made to expand my understanding of Python, Arduino, and how they are used together to make and operate a machine. It is a more complicated project than the ones I've done before as there are more components, both virtual and physical, as well as different versions of the project I had to make in order to test the various code components. It is still not perfect as I had to use cardboard and wooden skewers as the construction material which brought bugs into the physical version and made the project unstable.

Description

The program takes a picture, which is then turned it into a black and white line drawing version using edge detection. The program then draws the line version of the picture on a white board using an Arduino controlled robot.

Successes and Failures of the Project

There are two major components to the program, the virtual component, and the physical component. The virtual component to me is a complete success, there are very minor bugs, and it works accurately enough that the physical components should be able to draw the image. I made several virtual versions of what the physical version should be doing and the program works for those versions. The physical components are where the failures come into play. Now the case isn't that the physical version doesn't work at all, it does work. However, there are issues with the construction, and friction caused due to the marker moving across the board that causes the physical version to derail from the virtual ones.

Video 1. Drawing the circle image
Youtube Link

Video 2. Drawing the square image
Youtube Link

Video 3. Drawing a complex image
Youtube Link

You can see that the algorithm is not 100% correct, however the images drawn by the two simulations are accurate to the original image. The physical version is not working correctly and I will go into details on why that might be and my hypothesis further in the document.

Simulation 1:

Made with vPython, this simulation program gives a general idea of how the motors would work to draw the pictures. The motors in the simulation move according to how they should be moving in their physical form, including the motor shaft and its rotation. All the models in the program are built using individual rectangles and cylinders by giving coordinates and sizes of each part. The program was made before the construction of the physical components began, because of this it uses its own coordinate system which is why it is not 100% like the physical version. This does however give a great visualization of how the physical version would work. As you can see below, the program draws our test circle as well as some more complex pictures.

Video. Simulation 1 drawing a triangle
Youtube Link

Simulation 2:

After I made the physical version, I could see some bugs and glitches so I decided to make this second easier simulation. I made this simulation to see and confirm if the coordinates for a "pixel" generated by my main program were actually correct and would make the image. What this simulation does is simply creates a gif where every frame adds to the picture, essentially it draws the line picture and saves it as a gif so I can see the exact path the physical version should be making and make confirm that it should be working correctly. After testing, I was able to confirm that the coordinates being given are correct. This meant that there was some reason my physical version wasn't working even though all the information I give it is correct. As you can see, this simulation also creates our test circle correctly.

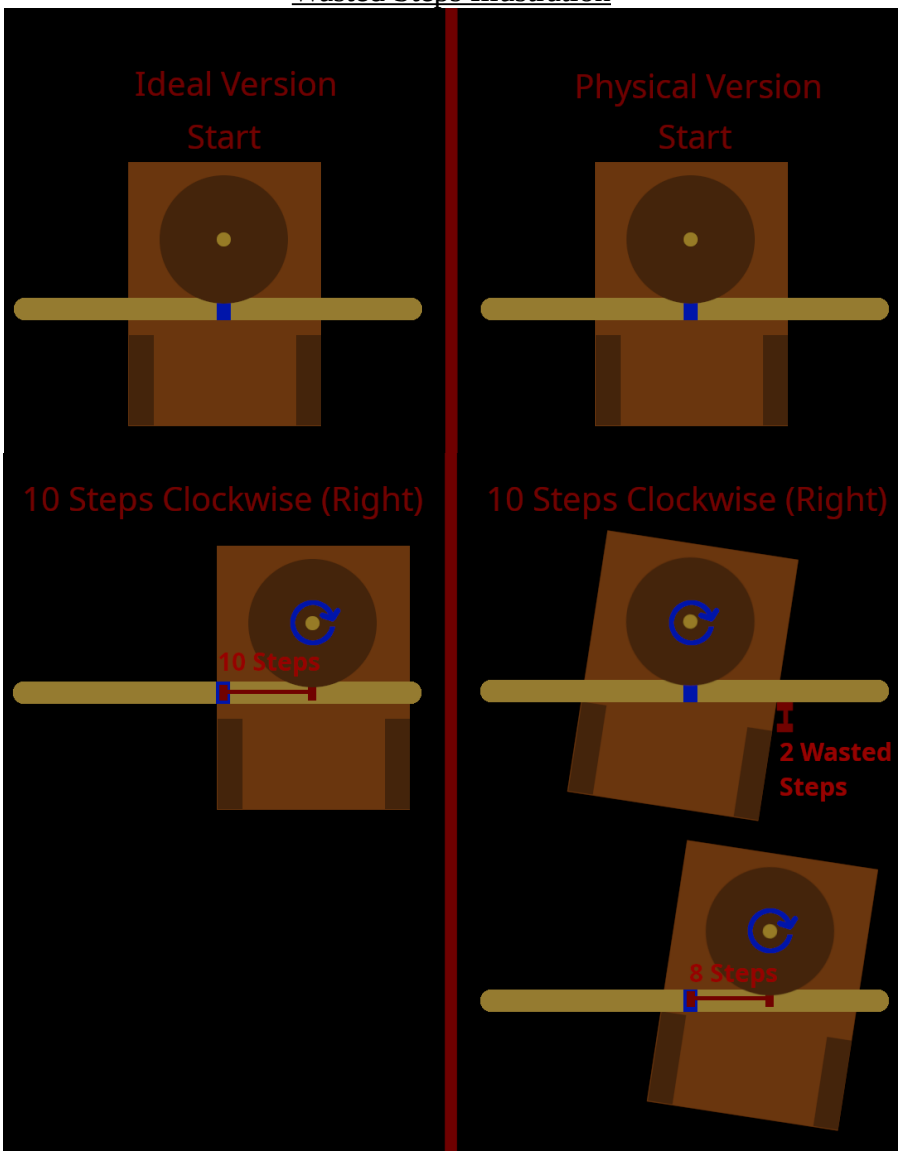
Video. Simulation 2 drawing a triangle
Youtube Link

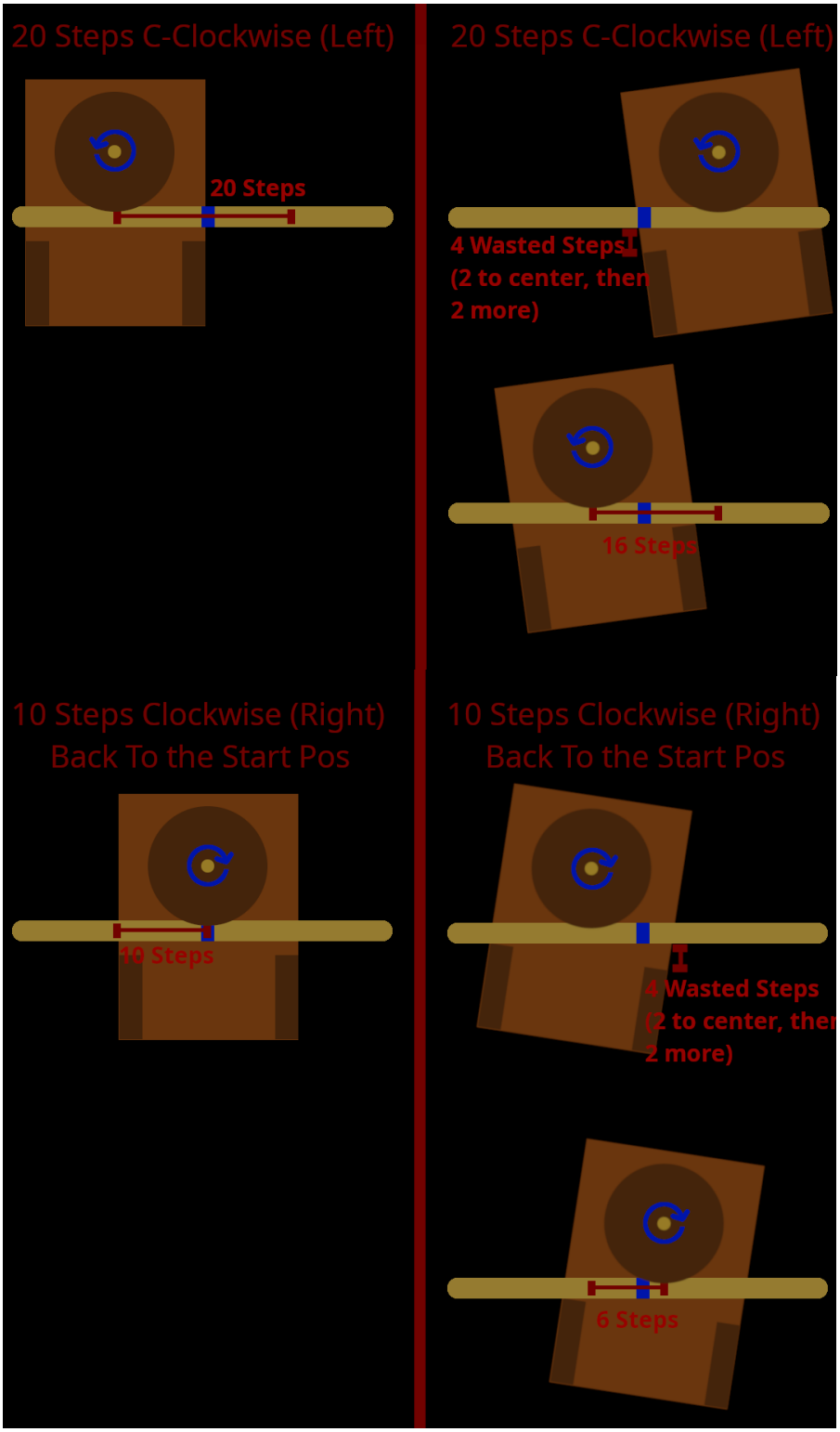
Possible Problems:

The biggest problem to me seems to be the construction of the physical project. It is constructed with cardboard, each piece measured and roughly cut by hand, skewers, tape, and superglue. I had to remake

the “carts” that carry the marker many times due to the weight of the motor titling the carts and applying incorrect torque. In my first design of the cart, I did not include the sides. Imagine a wheel sandwiched between 2 square pieces of cardboard. When I ran that, the wheel did not move but the rest of the construction did. Imagine if the body of a car starts moving instead of the wheel. My hypothesis was that the motor provided quite a lot of torque and the friction between the motor and cardboard being higher than the wheel and cardboard made the wheel stay in place and the rest of the assembly move. Because of this, my second design included sides that act as catches which catch onto the skewers and keep the body of the cart in place allowing the wheels to move. This added a new problem however, there were still some gaps between the side and the skewer and this means that the body rotates until the side touches the skewer after which the wheel starts to rotate. This introduces uncertainty in the movements making the image being drawn look worse. Below is a rough illustration of what I mean. Note the 2 step waste is a random number I picked for the demonstration. The motor I use has 4096 steps per revolution, so the waste steps are much higher.

Wasted Steps Illustration





Another reason the images are not being drawn correctly is also due to construction. While the motor provides quite a lot of torque the marker, when pressed onto the board provides too much friction. So, when we change direction the top end of the marker moves (connected to the cart) while the other end

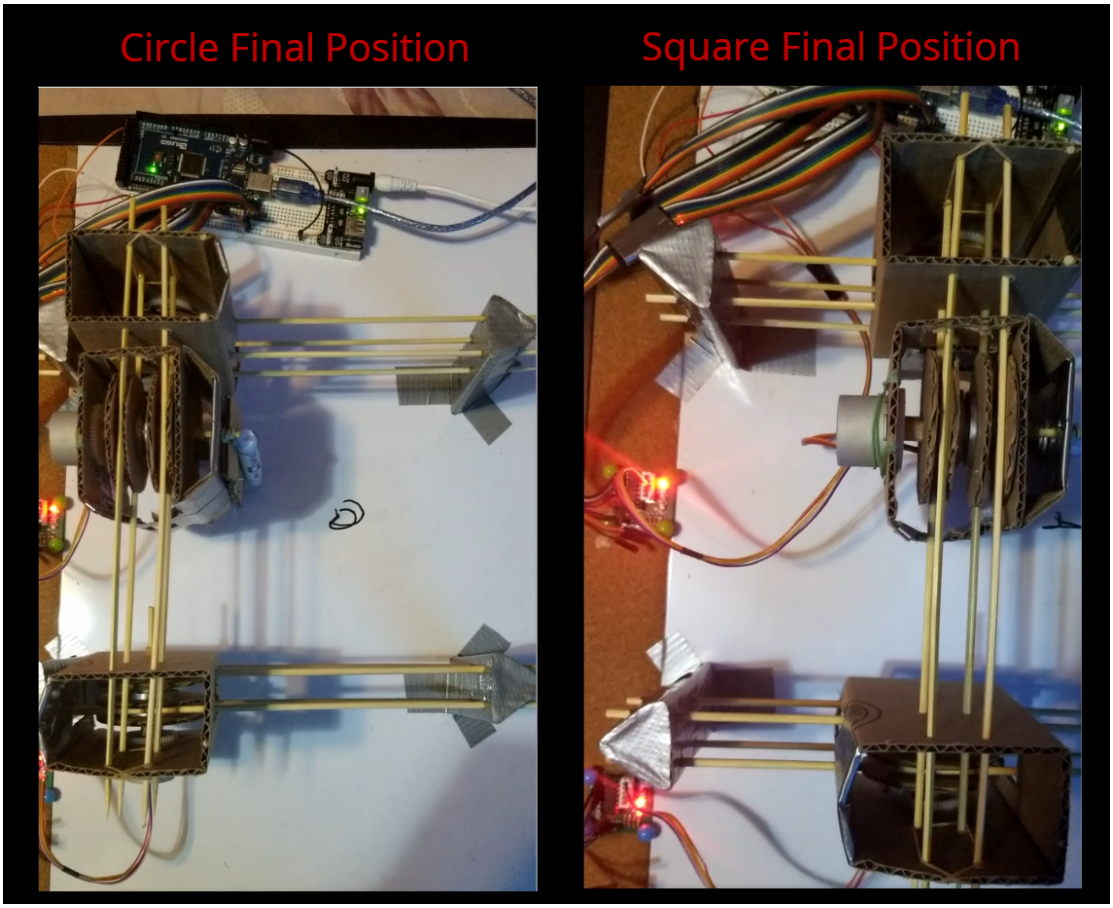
(the one that draws) stays fixed to the board and then start to drag after the cart has moved far enough. Below is a video showing this defect.

Video Demonstration
Youtube Link

As you can imagine, a drawing where there are multiple direction changes, all these uncertainties will add up to make all these complex images not be drawn correctly. Now the algorithm I've made to extract the lines isn't perfect but from the comparison videos you can see that the drawer should draw the image correctly, because of this I believe the physical construction is the biggest factor in the project not working correctly.

Another thing which illustrates that the "wasted steps" are the problem. The project is programmed to return to coordinates 0,0 (top left corner) after finishing drawing. This is shown at the end of the circle video, the marker returns to the top left corner (almost). You can notice that the vertical component of the circle is not drawn correctly, the program does not return completely to the top because of the movements causing these errors. This is much more noticeable in the square and complex (the drawing was aborted) videos. Notice how big the gap between the top left corner and the ending position of the marker is. This is caused due to the program moving to 0,0 virtually however all the uncertainties due to construction mess up the coordinates. The program things its moved to the top left corner however all the mistakes in the construction add up to make it so it does not move there completely.

Final positions



So what can I do to fix this? The easiest method is to 3D print the physical version. Unfortunately I do not have access to a 3D printer so I cannot test this solution but I am confident that simply constructing the project with more accurate measurements and sturdy materials should improve it greatly.

Detailed Explanation of the Program

The program is made using python to process the images and C/Arduino to run the hardware so I will break this section into their individual parts.

Arduino:

Lets start with the Arduino since the program for that is simpler. Im using the Arduino mega 2560 as it has sufficient amount of pins to run all the components. The Arduino uses programs written in C to control its components. The physical version uses stepper motors to move, stepper motors are motors where you can rotate the shaft using steps. This means that I can precisely control where the marker should go using x amount of steps. Now the Arduino has a set of basic libraries which the user can use to control individual components, So I had to use a stepper motor library to control them however the base stepper motor library can only run 1 stepper motor at a time, I have to use 3. For this reason I have to use an external library called accelStepper since it allows me to run 3 motors without having to write my own custom firmware. The physical version also uses a servo to control the marker moving up and down, the servo is a sort of motor like a stepper motor however it cannot do more than 1 revolution, there are "steps" which you can still control the motor using. In addition to this, the circuit uses an external power supply component to power the motors as the Arduino cannot provide the power by itself.

The code for the program is available on my github. Here is a link to the Arduino file. ~~Add Link~~

The starting of the program is just initializing and defining my variables. I include libraries I need, I define the pins I'm using for the motors. The speed, steps per revolution and acceleration are defined by the specific motors. They will be different for others depending on what stepper motor you use. The pins can also be different, it depends on what pins on the Arduino you use.

The setup uses methods from the accelStepper library to set the max speed, max acceleration, and normal speed of the motors. The library uses acceleration to run the motors faster than the normally can, the speed starts slow and then builds up however my program uses constant speed as using acceleration was slowing the movement a lot since the drawings go back and forth. I also attach the servo pins and its starting position using methods from its library.

The loop is where the Arduino gets instructions from python. I use serial communication to get commands to move the marker. I can give individual commands using the serial monitor or automate it in python to draw the image. The command is just "x-steps, y-steps, drawVal" without the quotes. The x and y steps are kind of like coordinates where the step value on the board doesn't change, so, (0, 0) is the start position, (x, y) is x steps horizontally and y steps vertically. The steps can be positive or negative. Positive x means move to the right, negative x means move to the left, positive y means move to the down, negative y means move up (note y is reversed since our origin(0,0) is at the top left corner, whereas normal Cartesian coordinates have the origin at bottom left). The draw val is 1 or 0, 1 being draw (the marker gets moved down using the servo) and 0 being don't draw (the marker gets move up using the servo). So python gives an instruction, for example "1000, 567, 0". The serial monitor gets these values and we enter the while loop, here all those values get assigned to variables that will be used later. Then the Arduino checks if we've reached the given steps, and if we have not then we run

the motors by one step. Once we've reached the number of steps, we stop moving and save the values to use in future calculations. Now we wait for the next command and loop.

Python:
The python component is quite a bit bigger as it does multiple tasks. The program loads and processes any image and finds the edges of the image. It then generates multiple arrays that are the lines in the picture, the arrays are full of coordinates that tells the program where to send the marker in the future. The program then sends those line arrays to the Arduino which then draws the lines.

The code for the program is available on my github as well. Here is a link to the Python file. ~~Add Link.~~ This file contains the simulation versions as well but we will ignore those in this write up as those are not part of the main program, they are more for testing.

So, the program takes help from some libraries, these being opencv2, numpy, and pillow for image and array processing. Vpython is used for the graphic simulation version. Then we have the time and serial libraries which help time out and send instructions to the Arduino.

The program is made of many classes with their own methods which helps in organizing the program. Below I will outline and explain each class. As I use them in the main methods of the program and at the end I will explain the main method itself.

Arduino:
The first thing we do in the main function is connecting to the Arduino. For this we have the Arduino class. This class' function is to connect to and communicate back and forth with the Arduino. We need some base parameters in order to do this. The comPort and baudRate are needed depending on how we set up the Arduino in the Arduino IDE. These two are then used to connect to the Arduino using the serial library and now we can talk to the Arduino.

The class has a couple of functions. The readFromArduino function just reads anything that the Arduino has outputted to the serial port and returns it. If nothing is there then it returns None. There is another read function which loops until the Arduino has outputted some data (Basically waits for some data) and then it returns that data. This class also has write functions. The write to Arduino function simply outputs to the serial port where the Arduino can read the data and process it. The writeWaitForAnswer is a combination function where the user can write to the Arduino and then return any data back.

ImageProcessor:
Next we load the image using the ImageProcessor class. This is a static class which is full of helpful functions to help with our images and is primarily used by our next class "Drawer" to help in generating coordinates. The processor has functions to load and show images with the help of opencv. The resizing function is custom since one of my goals was to be able to fit any image onto the white board/canvas. This helps me fit a different sized image with different aspect ratio (think of trying to fit a picture you took on a phone in portrait mode to a television) to our canvas. The calculation is based on the image and board aspect ratios. Basically, we have to make one side of the image fit to one side of the board and then we scale the other side depending how we fit the side in order to maintain the aspect ration (this makes it so the image isn't squished or elongated).

The detect edges method is used to turn our original image into a line image, this can then be processed to generate coordinates for lines. Although I've made versions with sobel edge detection, the one I'm

currently using is canny edge detection since that one gives the best results. Canny edge detection is an algorithm to highlight the edges. It is broken down in multiple steps, firstly you turn your image into grayscale (black and white). Then you blur the image, this smooths the image and gets rid of some noise. Finally you can scan through the entire image using a “kernel” and thresholds. Now in the past I’ve made my own versions of the algorithms for classes however I decided to use the opencv library and their methods for optimization and speed since mine took too long to give the results (I was a 3rd year student and these libraries are made and optimized by far more experienced professionals).

HIGHLIGHTED

The next two functions, generateLineSequence and traceLine, is where a big part of the processing magic happens. Now were two ways to “draw” a picture onto the whiteboard. You can draw like a printer, where the marker would go left to right, top to bottom, and just mark where a “line” should be. While that way would be fine and work correctly, I did not want my machine to work like that, I want my machine to actually “draw” the image using curves. So, in order to draw dynamically, I had to write a custom algorithm to extract lines from the image and these two functions are part of it. I will now explain in detail how this algorithm works.

GenerateLineSequence: This algorithm takes the edge image we got using our detectEdges function as a parameter. I first add a 1px border around the image as a cushion, this helps prevents errors we get while scanning in the future. I then make an array “checkedPixels” which is the same size as our image and set all its values to false, this array will then be used to check if a pixel has already been scanned. Now our image is essentially a 2D array, where each individual element is the value of the pixel, in our case this value is either 0 (black) or 255 (white) since our edge image is black and white. So I now loop through the image as you would loop through a 2D array, with 2 for loops, I start at the top left corner of the image, go to the right and repeat until we’ve reached the bottom (kind of how you read a page in a book, or how you’re reading this article). So going from left to right, top to bottom, I find a pixel that has not been scanned (here I use the checkedPixels array) and whose value is not zero (not black). We have just found a pixel that is part of a line. Now I send this pixel into another function, the traceLine function (there is a traceLineR function which uses recursion but that cannot be used to big images), to trace our full line.

- TraceLine: Now that we’ve found a white pixel, we can assume that this is part of a line. So we can trace this line until it ends or loops and store those coordinates. I do this using a stack and a while loop. I also use our checkedPixels array and a similar array “inStack” to see if a pixel is in out stack. I add the first pixel to the stack and enter our loop. In the loop I pop the pixel, add its coordinate to our line, and set that it is scanned, then I check around it for other pixels. Now any pixel in a line must have a pixel before and a pixel after (except for the start and end pixels). Now these previous and next pixels can be in 8 directions, imagine a tic-tac-toe game board with the 3 by 3 squares, you have the square in the middle which is our current pixel, and these next and previous pixels are around in it those other 8 squares. So I scan around the pixel like that. Once I find a pixel that is white, I check if it was already added to the stack in a previous iteration. If it was in the stack, the I pop it out from the stack and add it to our line, now there are a couple of factors that determine where it needs to be added (add it before our current pixel, or after) and I determine that using some math which I won’t go into here. Another possibility is that the white pixel we found was not in our stack and for that case we simply add that pixel to the stack to be scanned later. Now we loop, there are elements in the stack that were added and we do the same operations on them. We exit our loop when either the stack is empty (we’ve reached the end of the line) or we return to our start pixel (the line is a loop). This is how a whole line is scanned and we return this array to our previous function.

We now come back to the GenerateLineSequence function where we add this line to an array, and we continue scanning until we find the next non scanned white pixel. This is a pixel in another line so we got back to the traceLine function and the process repeats until we've gone through the whole image. Finally, I sort the array of lines based on decreasing line size which gives a nice effect of the machine drawing the longest lines first and the lines get shorter as the image is drawn.
Now as I would like to get a bit more technical and do an algorithm analysis to show how my algorithm while seems slow is not actually slow. From the amount of nested loops I've used I thought that the algorithm would be extremely slow, however that is not the case at all.
To analyze the time complexity of this algorithm, we check how many loops and operations the algorithm does for an image. Lets say we have an image of size WxH where W is the width of the image in pixels and H is the height of the image in pixels. The generateLineSequence function has 2 nested loops that iterate over the width and height of the image. Those loops then call traceLine for every white pixel we find. TraceLine has a while loop that will have maximum iterations of the longest line in the image and for every iteration in the while loop we have a for loop that iterates 8 times. All of these are nested beginning from the generateLineSequence function, now this seems like a very time consuming algorithm. If we do the analysis the normal way where we multiply the number of iterations of each loop when theyre nested we get a time complexity of O(W x H x pixelsInMaxLine x 8) (using big O notation) lets say in the worst case the image is just one big line, so our pixelsInMaxLine becomes WxH and this makes the complexity O(W x W x H x H x 8) which simplifies to $O(8 \times W^2 \times H^2)$ which is a bad algorithm in my view. This is essentially a $O(n^4)$ algorithm which is bad. However when we do a deeper analysis on the algorithm we can see that it is actually not close to $O(n^4)$. Because I implemented checks to see if I've already scanned a pixel, this reduces the complexity. So, anytime I go into the traceLine and find a line, all those pixels that are in that line will not be scanned further. So the while loop iterations never happen for those in the future. This means that essentially, I am only going through all the pixels in the picture once and the time complexity of that becomes O(W x H x 8). If our image is one big line, then the first traceLine call goes through all the pixels in that call and the rest of the for loops just iterate normally without any other nested calls making the time complexity O(W x H x 8 + W x H) which is just O(9 x W x H). So the algorithm is actually in the time complexity of $O(n^2)$ which is A LOT better than $O(n^4)$. To simply show the differences, lets take a standard HD image of size 1920x1080, this makes our incorrect time to be $O(8 \times 1920^2 \times 1080^2)$ which means there would be 34.4 trillion iterations. In reality, by using our correct time complexity, we get O(8 x 1920 x 1080) which is 16.6 million iterations, which seems big to us but modern computers can compute this very quickly.

Finally, we have the getDrawInstructions method which is mostly a functions that combines all our other functions into one. This allows us to just call getDrawInstructions and the function does the rest. The function loads an image, resizes it, detects the edges, and then finally generates the line sequence and returns the coordinates.

Drawer:
The Drawer class is a middle man between the Arduino and Image Processor. This class converts all the coordinates given by the ImageProcessor into values the Arduino can use and sends that information to the Arduino. You see, an image is made up of pixels, however to get that image to be drawn onto the board we need some other units. In this case we need steps that the stepper motors can travel to and this is how the physical version will draw the image. It is a bit more complex since the stepper motors are essentially wheels while we need to travel in a linear (or straight) motion. I will explain that portion of the math later on as we go through the code. In our context, the virtual coordinates are pixels, while physical coordinates are steps.

Note: The class has 2 functions, drawTestGif and drawTestGif2, built into it, these are the simulation functions I talked about for simulation 2 and because they are sims I won't be explaining them in depth.

__init__: Since the Drawer is an object, we have a constructor for it. We construct the drawer by sending an Arduino object, our canvas or whiteboard dimensions, the number of steps per revolution for our stepper motor, motor wheel radius, and optionally a pixels per centimeter value as parameters. The Arduino is the object created using the Arduino class. The canvas dims is a tuple value that has the dimensions of our canvas in centimetres (13 cm x 8 cm for me). Motor steps per rev is the number of steps that do one full rotation of the stepper motor (4096 steps for me). Motor wheel radius is the radius of the wheel or circle we have connected to the stepper motor (1.5 cm for me). Finally, we have an optional parameter cm per pixel. This tells the computer how many pixels we want in 1 cm. More pixels per centimetre means less steps per pixel. You can imagine how putting 100 pixels in 1 cm could be compared to 10 pixels. 100 pixels would give better quality, but the drawing process will be slower since the motors will have to do more micro/mini steps. Basically, the image we will be drawing gets scaled and fitted to the box created by our canvasInPixels x pixelsPerCm. For example, if we have an original image of size 1080x720 pixels, our canvas is 15 cm by 10 cm, and we want 10 pixels per cm. The original image gets scaled to fit in the resolution of 150x100 pixels (15 cm * 10 pixels/cm and 10 cm * 10 pixels/cm) using the resizing function I talked about in the ImageProcessor section, so it keeps its aspect ratio.

DrawImg: This is one of the two main functions the user will use, this function takes an image that we got from using loadImage in the ImageProcessor. The function then calls the getInstructions function to get the list of coordinates and then it calls draw.

GetInstructions: This is a helper function, it basically calls the getDrawInstructions function in the ImageProcessor to get the instructions, and it optionally passes the resizing params if we defines a pixelsPerCM value. At the end we call recalcParams to recalculate some values (I will explain this function later). Finally, we return the instructions we got.

DrawIns: This is the second main functions. If the user decides to make their own instructions array or change the values in a generated one then the user can call this function as you just need the instructions for this. Essentially this would skip the getInstructions step in the previous function. Since, the instructions are predefined, we don't know the boundaries of this image so we call findMax which finds the maximum x and y values. This helps us get a rough idea of what the image size is. We then call recalcParams to fix some values and then we call the draw function.

RecalcParams: Now this function involves similar math to our resizing function in ImageProcessor. Basically, this function exists to set horizontal or vertical offsets. Suppose our image is narrower in width than the width of the white board. So if we drew normally, there would be a lot of white space to the right. This function defines an offset value that makes it so that extra white space is split between the left and right side meaning that it makes it so our image is in the centre of the whiteboard. The same thing is done if we have a shorter image than our whiteboards height. The image gets centred vertically. We also define our pixelsPerCM value here if one was not given.

Draw: This function uses the instructions given to draw the image. Now the instructions are a 2D array, the elements of the instructions array are lines which are 1D arrays, and the elements of each line is the pixel coordinates that make up their respective lines. So, for each line in the instructions, the functions

goes to their start point using moveTo, then it draws the lines using drawLine. At the end, we move to 0,0 or the top left corner of our whiteboard.

MoveTo: This is the function we can use to move our marker to a given coordinate without drawing anything. The first thing we need to do is convert our pixel coordinate to step coordinates that the stepper motors will use, this is done with the pixelsToSteps function which is explained later. Once we have the number of steps we need to travel to, we simply send the instruction to the Arduino using the writeToArduino function. Now if you recall, when we were making our Arduino program, the command we use to move the marker is of the form “x-steps, y-steps, drawVal” without the quotes. So we send that string, where our x and y steps are given from the conversion and the drawVal is 0 since we don’t want to draw anything. If everything is working correctly, our marker should be moving and we listen to the commands the Arduino is sending back with the respective functions until we get a “COMPLETED” or “MOVING” command back. Our marker should be at a line starting position (or 0,0) now.

DrawLine: draw line is basically moveTo with some minor adjustment. Most of the process is the exact same except for the fact that we now have a for loop to loop though all the coordinates in the line, and our drawVal value in the command becomes 1 since now were drawing. For each coordinate, we convert the pixel coord to a step coord, send the command to the Arduino using a string, and read until we get “COMPLETED” or “MOVING”. At the end of this function we should have a full line drawn. We then go back to the draw function and potentially moveTo the next line start and draw that line, or simply go to 0,0 indicating the drawing is finished.

PixelsToSteps: This is the function where we need to convert our pixel coordinates into steps that the stepper motors can use to draw. This is also where most of the math happens and in order to explain this I will make it a separate section. I will try to explain the math in 3 ways, the general concept to explain the process without any math, the math with variables, and then the math with the values I used in my project as well as an example.

Now firstly, let me answer some basic questions.

1) Why do we need to convert pixels to steps?
   Well, this is due to the motors used in the physical machine, Our virtual image is made with pixels while our motors need a number of steps to rotate. Although if we didn’t have stepper motors, we would still need a way to equate the distance on the board to the distance of a pixel in the image.
2) Why can’t we simply make the steps and pixels equivalent so one step on the board is 1 pixel?
   To start, we would need to first check how many steps it would take to map out our board which is not constant as it changes based on the wheel size. Also since our image is made and processed with pixels, any coordinate for where a pixel in the line should be is in pixels as well. However, our physical white board has physical dimensions and although we can make it so the available drawing space on our board is made of “pixels” by setting a value like pixels per centimetre, this would make it too crowded. In my case, my drawing space is around 10 x 10 cm, if we try to fit an average HD image to this which is 1080 x 720 pixels then we get around 108 pixels per cm. We are drawing with a marker which is already thick so trying to include 108 “dots” in 1 cm doesn’t make sense.

The General Concept:

So, one thing to note is that our picture is flat on the screen, to travel from one pixel on the image to another pixel we just move in a straight line. This is also true when we draw on a physical

white board, we move our hand in a straight manner to connect two dots on our board. However, in our case we are using motors and motors spin, so we have to use the rotation of our motor in order to move our marker from one point to the other. It is kind of like a car, you rotate the wheels but move in a straight line but in order to go straight exactly a certain distance we need to rotate our wheels an exact amount, this is where a big part of the math is involved. All circles have a circumference which is the length of the perimeter or the length of the outer edge of the circle. We use this to know how much we can travel with one full rotation of the circle. Another thing we use is the arc length, which tells us how long the distance between two points on the outer edge of the circle is. And this is what we use to travel linearly or in a straight line. Now that was all intro to how what we need to do this conversion. So now I can tell you the conversion steps beginning with a coordinate.

1) We first take the coordinate and add any offsets to the values (This is to center the image onto our board).
2) Determine how many full rotations of the wheels you would need, and how many partial rotations of the wheels you would need.
3) Find out how many steps you need for the full rotations, and how many steps you need for the partial rotations.
4) Add the two rotations together.
5) Return the total number of steps.
   Note: the process needs to be done to the x and y coordinate to get the x and y steps.
   We can now travel the total steps in both direction to go to our point on the board.

Math with Variables:

Now that we know the general process, we can discuss exactly how we convert this rotational motion into the linear motion. One major thing to note is that the accelStepper library for our Arduino turns the steps into absolute position based values. This means that our steps turn into a coordinate system. So if we move 100 steps, we get to position 100, then if we want to go 200 steps from our starting position we would move 100 steps in a relative position but in this case the motor does not move since we are already at that position. Instead of moving 100 steps more we have to move to 200 steps and this is how we get to position at 200 steps. We have the option to turn this off and make it in to relative coordinates but I prefer the absolute version since we don't have to subtract coords our steps from prev values. Now to convert the pixel to steps we first need to get and define some starting values and variables which will affect our calculations:

- stepsPerRev = Steps per revolution: Number of steps it takes to do one full revolution of the motor. The more steps per revolution leads to a finer control of the marker, on the other hand, less steps per revolution leads to rougher control. This value is dependent on the motor make and model you use so refer to the spec sheet for the motor.
- WheelRad = Wheel radius: The radius of the circle/wheel you have attached to the motor. The size of the wheel determines how much you move for 1 step, a larger wheels makes the marker move more per step (since arclength for 1 step is bigger) but it means you'll lose finer movements and a smaller wheel makes the marker move less per step which means that you get more finer movements but it takes more steps to travel further.
- Offset = Any offset we need to add to the pixel in order to center the image on the board. This is calculated during our recalcParams function and is a value in centimetres.
- PixlesPerCM = Pixels per centimetre: Defined by the user or calculated by dividing number of pixels in one direction by the board size in centimetres in the same direction.

We will be using more variables later which I will define as we get to them. We can now begin converting our pixel coordinate to steps.

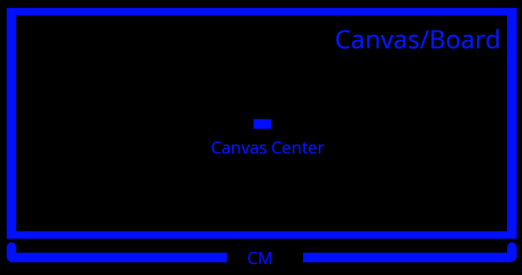
Let p = our pixel coordinate that we need to convert to steps. It is a positive integer value.

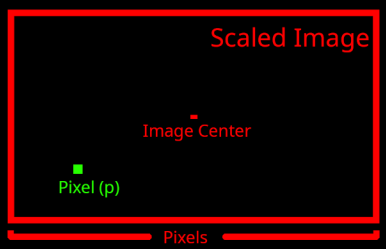
p = pixel coord (pixels)

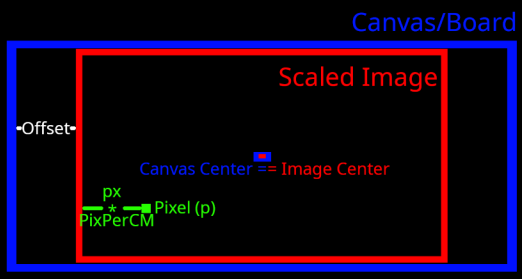
o = offset (cm)

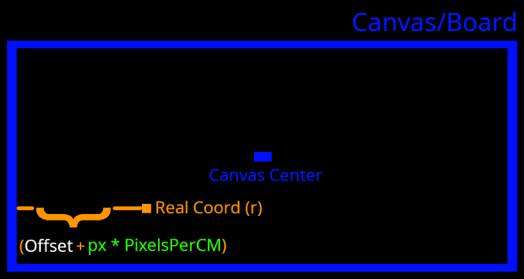
pixelsPerCM = pixels per centimetre (pixels/cm)

r = real coord (cm) = (p pixels) * (pixelsPerCM pixels/cm) + (o cm)
= ((p * pixelsPerCM) cm) + (o cm)
= (p * pixelsPerCM + o) cm





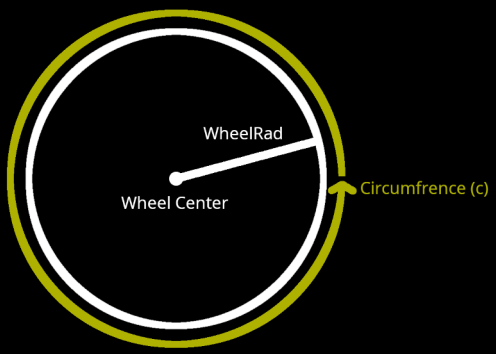




wheelRad = motor wheel radius (cm)

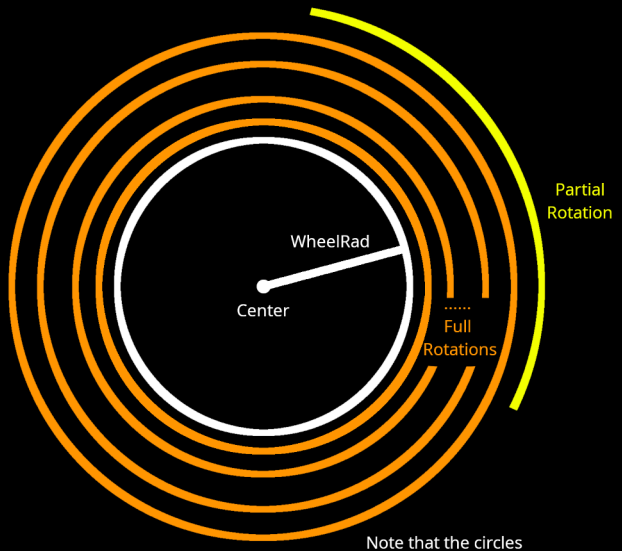
π = Pi = 3.14159

c = wheel circumfrence (cm) = 2 * π * (wheelRad cm)
= (2π*wheelRad) cm

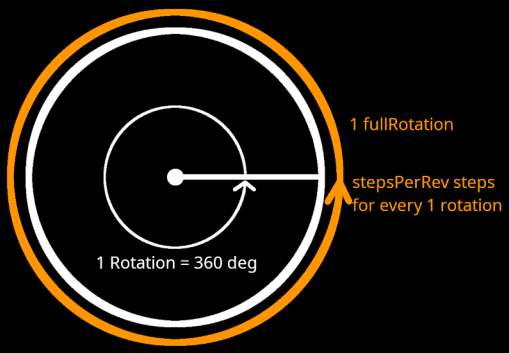


$$\text{fullRotations} = \lceil (\text{r cm}) / (\text{c cm}) \rceil$$
$$= \lceil \text{r} / \text{c} \rceil$$

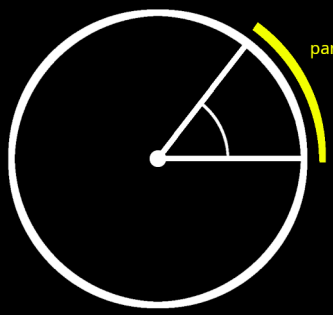
$$\text{partialRotations} = (\text{r cm}) \bmod (\text{c cm})$$
$$= \text{r} \bmod \text{c}$$



$$\text{FullRotationSteps} = \text{fullRotations} * (\text{stepsPerRev steps})$$



$$\text{PartialRotationSteps} = (\text{partialRotations} / (\text{c cm})) * (\text{stepsPerRev steps})$$



$$TotalSteps = \lceil FullRotationSteps + PartialRotationSteps \rceil$$

Now this is the process for only one dimension, we do this again for our other dimension to get an x,y coordinate. In the program they're combined so the two dimensions and steps for them are calculated at the same time.

Main:

The main function is where we initialize all our starting parameters and run the program. Here we tell the program where the image we want to draw is located on the computer, the dimensions of our whiteboard in cm, how many steps to make the motor do one revolution, the radius of our wheel. We can make our Arduino and drawer objects with the constants we defined. Finally we can load in our image and then tell our drawer object to draw it on our whiteboard.

Future Plans:

For now, the project will be on hold since I do not have a 3D printer to help me reconstruct the physical component. Once I have access to that, the obvious next step is to print and test my hypothesis. Once the physical component is working a lot better I have some more plans.

- Fix the line detection algorithm: It is working for now however when a line splints into multiple paths the algorithm has trouble. I recently noticed that the algorithm is also having troubles with larger versions of the simple shaped so I will have to debug that.
- Wall mounted version: I want to make a more compact version that can be mounted on the wall vertically and the project would draw the images on that.
- Switches for boundary detection: Currently I have to input the boundaries of my drawing surface, Id like to add some switches to the program can figure out the bounds on its own.
- Camera: I'd like to add a camera feature such that you can show the camera a picture and then the project draws it.

With this and a more polished algorithm I believe that the project could be a very interesting showpiece I can show around.