CMPUT 312 – 6-DOF Painting Robot

Justin Valentine (jvalenti), Yash Bhandari (yashaswi)

1) Goal

Our goal for this project was to develop a robotic painting system. Specifically, we wanted to use an off the shelf 6-DOF robotic arm to physically render Scalable Vector Graphics (SVG) images on a slanted canvas.

2) The Robot & Controller Computer (Justin)

There are not many options for inexpensive robotic arms outside of china, so we realized pretty quickly that if we wanted to achieve our goal we would need to risk it and order a robot from AliExpress. We really liked the design of the 7bot, as the parallel linkage in the elbow allowed for significant weight reduction in the upper arm. Another bonus is that the 7bot had been used by Pindar Van Arman in his Cloudpainter project. With this in mind we ended up ordering the Thanksbuyer ARM-21N3 a 7bot clone from AliExpress. Some nice features about the construction of this robot kit are that the movable joints are connected by bearings which reduces friction, and that the main components are made from laser cut aluminum, which makes them lightweight and rigid.

On the left is how the robot arrived, and on the right is the robot assembled.

The Robot Servo Specs:

- Models: 4 x TD-8120MG 20kg, and 2 x MG90s
- Operating Voltage: 4.8V
- No-load Current: 210mA

- Stall Current: 2100mA ± 10%

To control the robot we are using a Raspberry Pi 4 Model B along with a Adafruit 16-Channel PWM servo-hat. Initially we thought that we could just use software generated PWM signals (something the pi can do natively) however after some preliminary testing we found that the signals were not accurate enough for our use case. We researched alternative approaches to generating PWM signals and the Adafruit 16-Channel PWM servo-hat ended up meeting our needs.

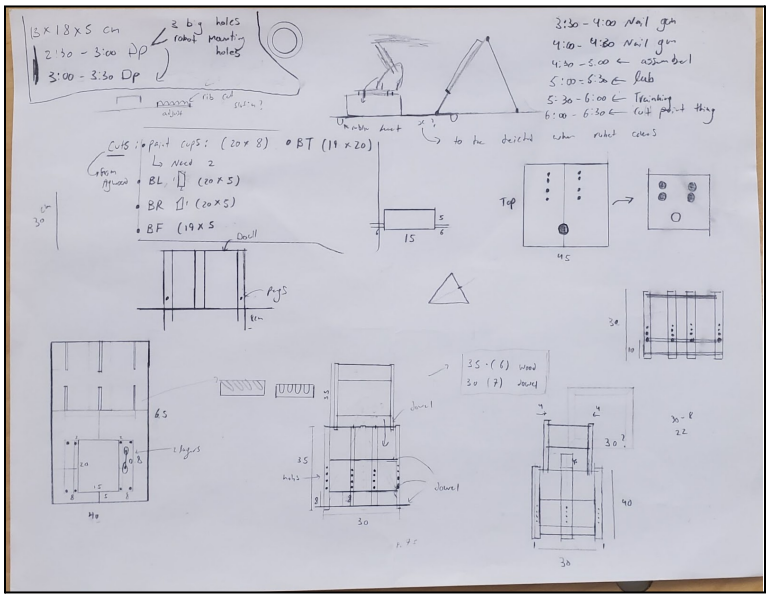
The Controller Computer Specs:

- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- 8GB LPDDR4-3200 SDRAM
- 16-Channel 12-bit PWM

To power the robot and the controller computer we are using two separate power supplies. The robot is powered by a 5V 3A power supply, and the Pi is powered from a laptop over USB-C.

3) Designing & Assembling (Justin)

Like all great designs ours started off with some quick napkin math. We had some rough ideas about how we wanted the robot to be positioned with respect to the canvas and how it would interact with the paint cups but nothing concrete. A big challenge that we faced was that we had to design the whole thing before the robot arrived. With this in mind we aimed to make the design flexible so that we could adapt it after the robot arrived.

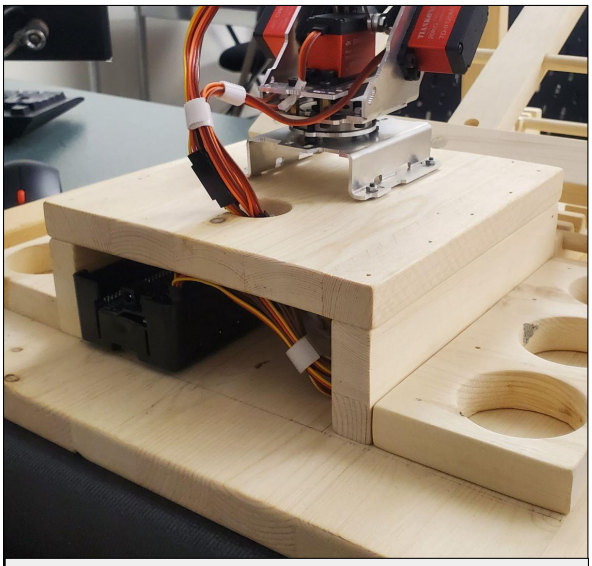


Once we had our design it was time to put my middle school woodshop skills to the test and actually build the thing. I used a table saw and a miter saw to process the wood into smaller

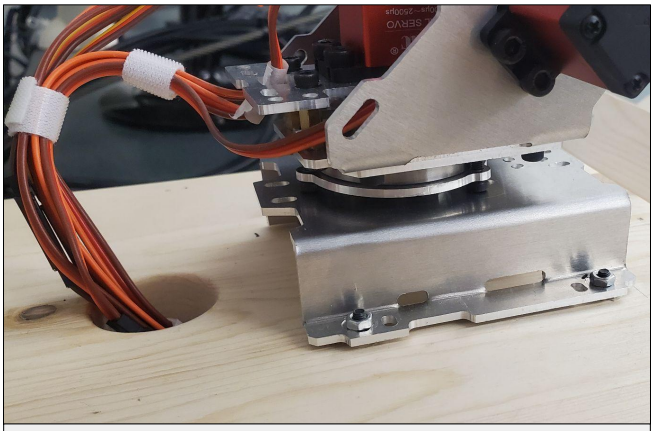
chunks and then used a drill press and a bandsaw to do the finer detailed work. Then to assemble it I used a nail gun and wood glue.

Bellow are some of components of the design:

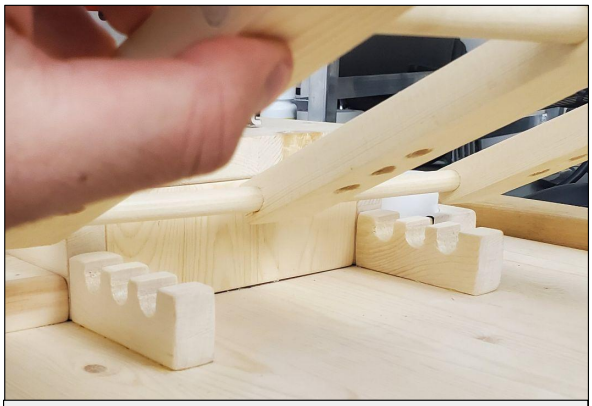
Paint cup holders, with removable paint cups (3 on left side, 3 on right side)



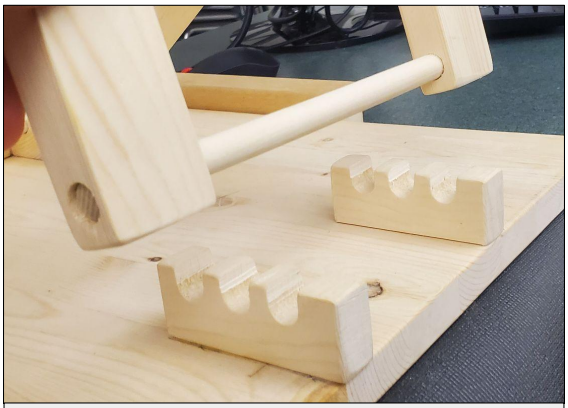
Controller computer housing



Countersunk robot mounting holes, and cable hole

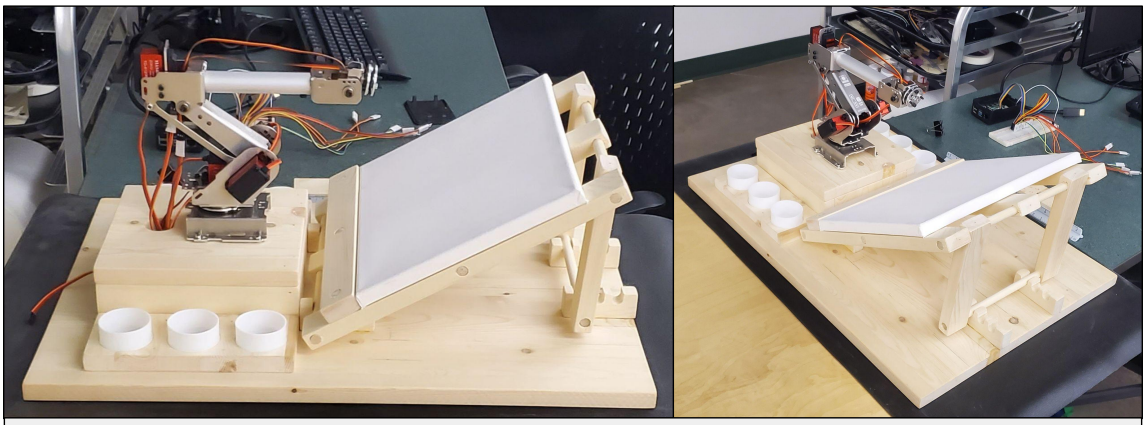


Adjustable front legs of canvas stand



Adjustable rear legs of canvas mount

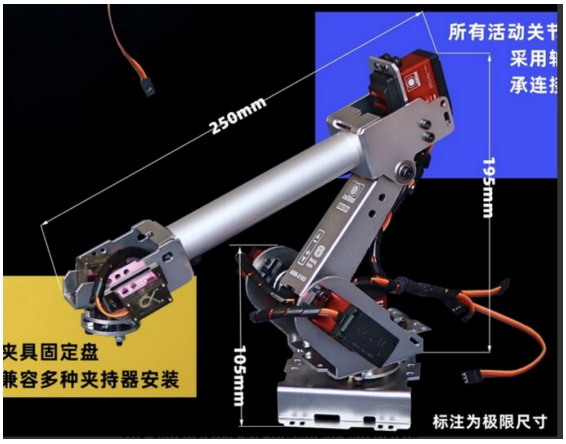
Height adjustment on canvas mount



Fully assembled robot & robot stand

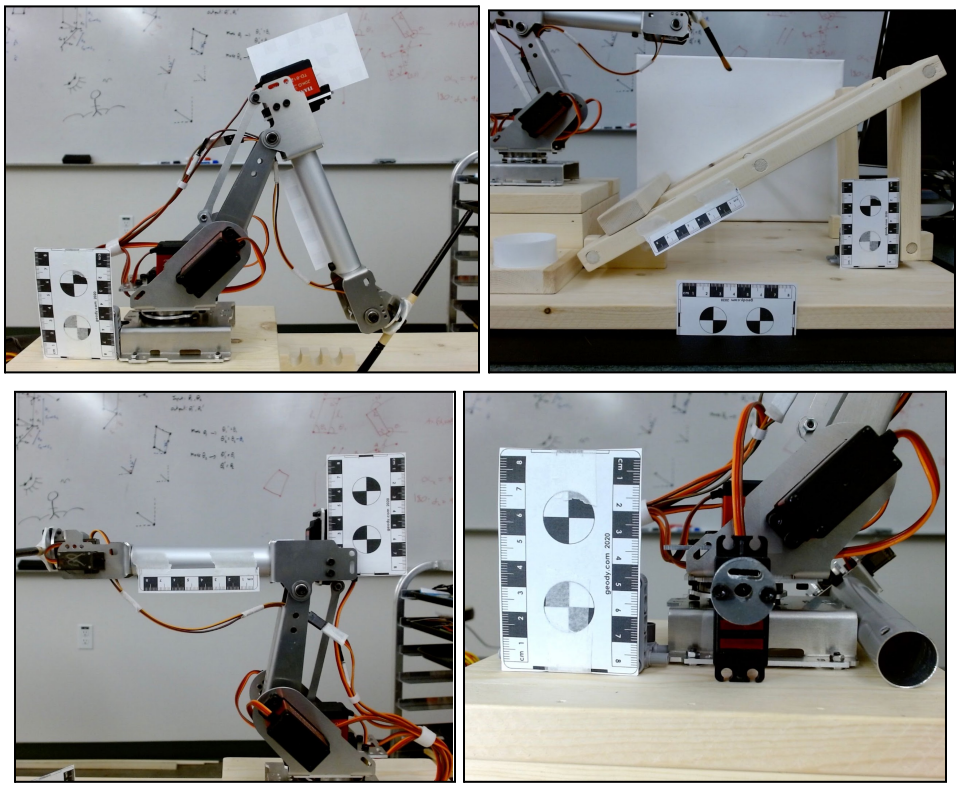
4) Measuring (Justin)

A major problem in getting the forward kinematics working was getting accurate joint measurements for the robot. We contacted the seller to see if they could send us anything that could help and this is what they sent:



Which was not particularly useful, however we appreciated their effort.

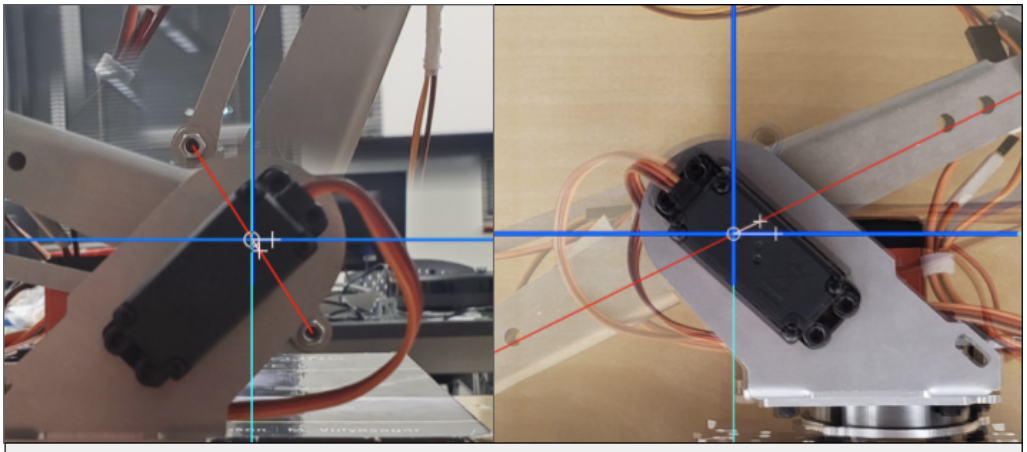
Next we tried using images to measure the robot. I printed off a bunch of cards for scale and taped them to the robot like so:



Using these images and the ruler tool in Photoshop we could set a scale using the cards in the scene and then measure parts of the image with respect to that scale. However this did not work great due to the fact that the cards were not at the same distance from the camera as the things we were measuring, and the camera was not perfectly perpendicular to the thing that you were measuring. Another challenge with this approach is that a lot of the points we wanted to measure were obstructed from the camera's view.

In the end we just used a set of calipers to measure the joints of the robot, which worked well enough.

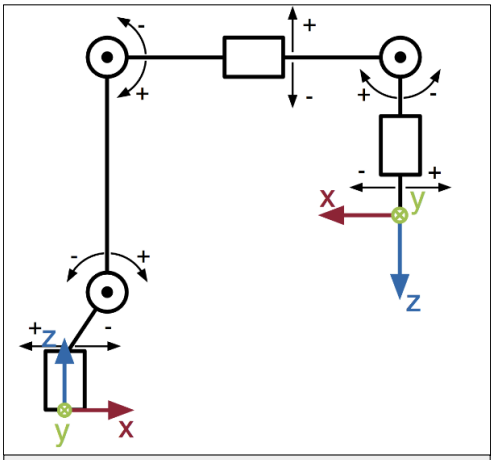
Next we needed a way to measure the servo axes with respect to their respective coordinate axes. In other words we needed to see how much the servo's 0 degree position differed from where it should be with respect to the axes in its coordinate frame. For 4 out of the 6 axes we were able to eyeball this measurement, however this could not be done well for the shoulder, and elbow axes. For those we used Photoshop to measure the angle.



In blue are the respective coordinate axes, and in red is the servos axes

5) Forward Kinematics (Justin)

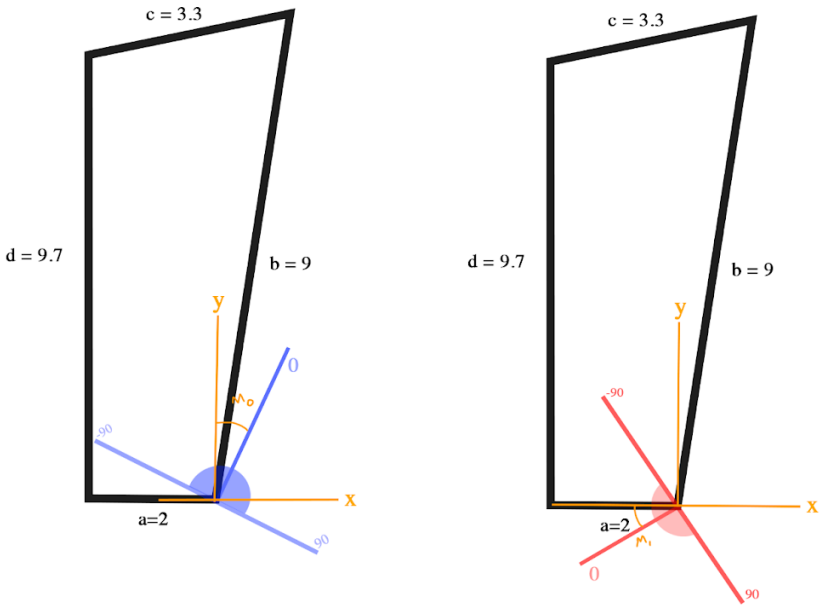
For the forward kinematics we wanted to model the arm as a serial linkage so that we could use simple homogeneous matrix transforms for the forward kinematics.



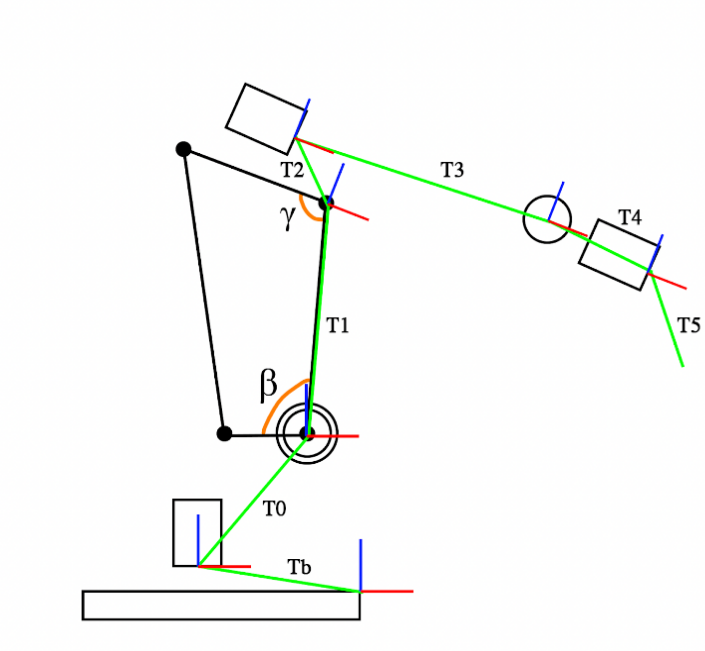
Serial linkage model

In doing so we introduced two levels of angle abstraction. First are the physical angles which are the angles we actually send to the servos. Second are the logical angles – these are the angles of the joints in the serial linkage model.

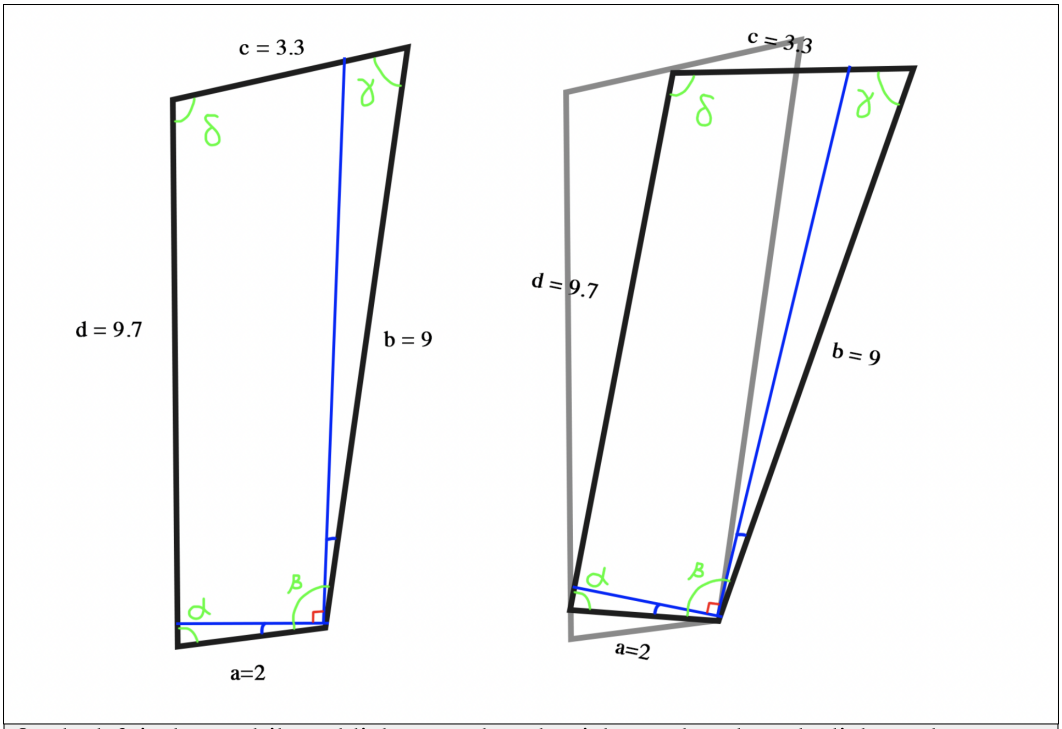
For this to work, we need to align the servo axes with their respective coordinate axes. This is achieved by ForwardKinematics.PhysicalToLogicalAxis, which works by adding a constant and flipping the direction of rotation (this was unique for each servo).



In order to model the robot arm as a serial linkage, we needed to find the appropriate angle transformations from physical to logical angles. As you can see below, the robot has 6 translational transformations and 6 rotational transformations.



The robot has mostly serial linkages except for the shoulder and the elbow, which form a dual driven quadrilateral linkage:



On the left is the quadrilateral linkage, and on the right we show how the linkage changes when we perform a shoulder movement

For the shoulder to behave like it does in the serial linkage model, we need the angle beta to remain constant as we move the shoulder joint. To do this we just add the change in angle of the physical angle of the shoulder to the physical angle of the elbow servo.

The elbow angle in the serial model is given by gamma but we can only set the angle beta with the elbow servo. So we need a way to control gamma by controlling beta. We can write gamma in terms of beta as follows:

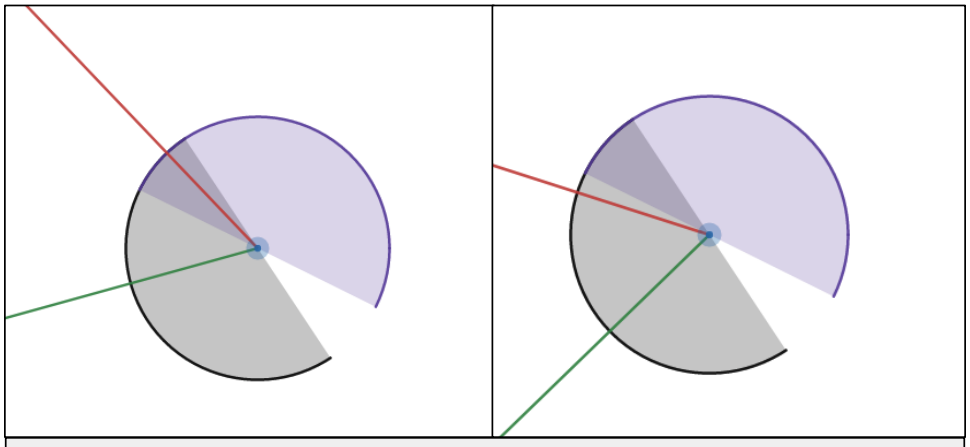
$$\gamma = \cos^{-1}\left(\frac{b^2 + \sqrt{a^2 + b^2 - 2ab\cos(\beta)}^2 - a^2}{2b\sqrt{a^2 + b^2 - 2ab\cos(\beta)}}\right) + \cos^{-1}\left(\frac{a^2 + b^2 + c^2 - d^2 - 2ab\cos(\beta)}{2c\sqrt{a^2 + b^2 - 2ab\cos(\beta)}}\right)$$

This allows us to solve for gamma given an angle beta. We also need to solve for beta given an angle gamma, but there is no analytic inverse to this function. Luckily the function is monotonic over a suitable domain, so we can use binary search to search for beta given an angle gamma up to arbitrary precision.

The process of transforming physical angles into logical angles is outlined in the function ForwardKinematics.physicalToLogicalAngles in file kinematics.py.

Lastly we needed to consider the physical limitations of the joints themselves. All joints except for the elbow have physical limits equal to some predetermined constant, however the upper

and lower bound on the physical angle of the elbow joint is a function of the shoulder joint. Below is an illustration of that relationship:



The black region is the range of the elbow joint, the purple region is the range of the shoulder joint. The red line is the upper limit of the elbow joint, and the green line is the lower bound of the elbow joint.

This is calculated in the function jointLimitsPhysical in the ForwardKinematics class.

6) Inverse kinematics (Yash)

For inverse kinematics, we used Newton's method. We estimate our Jacobian by using our forward kinematic model to compute the change in end effector position when each of 6 joints was moved slightly. If Newton's method fails to converge, we randomize the starting pose and try again. If that fails, we try again from a well-behaved starting pose. The actual robot arm does not move during this process.

Initially we let each joint vary from 0 to 180 degrees, as this is technically possible for each joint in isolation. However, we quickly learned that joint 1 (shoulder) and joint 2 (elbow) physically constrain each other and many states in our search space were impossible to reach. Our solution was to allow joint 1 to vary freely and just bound joint 2 accordingly.

We also ran into the issue of inverse kinematic solutions that required the robot arm to essentially reach through the canvas and paint a point from the back. We solved this by

As we were working through these issues and constraining our search space to physically feasible positions, we ran automated tests to ensure that our inverse kinematics procedure was still able to converge on a broad sample of points in the workspace. If any points failed to converge, we knew that our constraints had to be loosened.

Code: inverse_kin.py

7) SVG parsing and rendering (Yash)

Scalable Vector Graphics (SVG) is a language for describing vector graphics. It's a common output format that can be produced by software like Adobe Illustrator or Inkscape and can be rendered by most browsers.

SVG is XML based, so we were able to use the built-in python XML parser to parse the raw text into a tree. Our SVG parser then traverses the tree and parses the data for each shape enumerated in the tree. We use regular expressions liberally during this step to handle the common SVG compression tactics (e.g. removing delimiters and preceding digits where they are not strictly necessary). Code: svg.py

SVG shapes range from simple lines and ellipses to complex paths composed of multiple bezier curves. We generate a sample of the points located on the boundary of these shapes with pixel coordinates. These points can then be plotted in 2D or converted to 3D coordinates that can be plotted or painted directly on a canvas. Code: svg_renderers.py

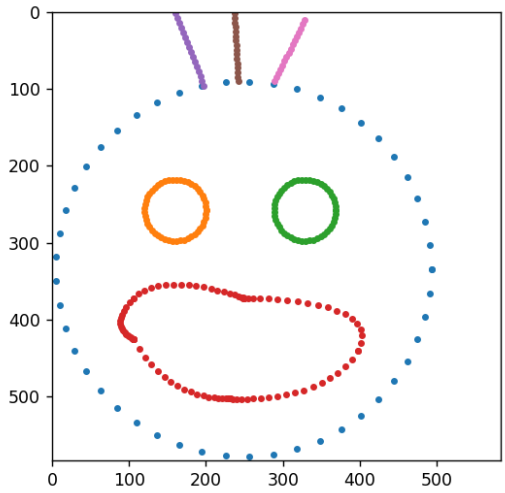
8) Converting 2D image coordinates into 3D coordinates (Yash)

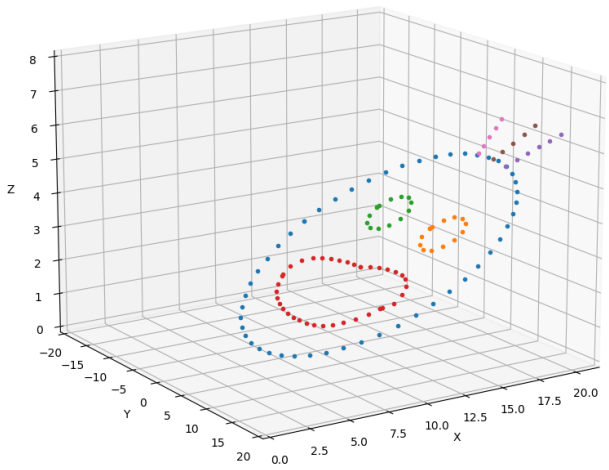
We employ the following process to map 2D images coordinates (specified in pixels) into 3D coordinates (cm) on the canvas:

$$\begin{bmatrix} x_{3D} \\ y_{3D} \\ z_{3D} \end{bmatrix} = \begin{bmatrix} H_{canvas}cos(\theta) & 0 & 0 \\ 0 & W_{canvas} & 0 \\ 0 & 0 & H_{canvas}sin(\theta) \end{bmatrix} \begin{bmatrix} 0 & cos(\theta) \\ 1 & 0 \\ 0 & sin(\theta) \end{bmatrix} \begin{bmatrix} x_{image}/W_{image} \\ y_{image}/H_{image} \end{bmatrix} + \begin{bmatrix} x_{offset} \\ y_{offset} \\ z_{offset} \end{bmatrix}$$

1. Scale the x/y coordinates to [0,1] by dividing by the image width and height
2. Map the x value of the image coordinate to the y coordinate in 3d. This is the horizontal axis along the width of the canvas.
3. Map the y value of the image coordinate into the 3d x and z coordinates.
4. Scale up the point coordinates from [0,1] to the dimensions of the canvas
5. Add the offset from the origin in robot coordinates to the bottom left corner of the canvas

We try to keep an optimal distance between adjacent points to achieve a balance between speed and fidelity. We do this by generating a very high number of points (left) and then pruning out any that are too close, resulting in an evenly spaced point cloud (right).





9) Painting an SVG

The end-to-end process for painting an SVG looks like this:

1. Parse an SVG file
2. Convert the first SVG shape into points in a 2D image
3. Map the 2D points into 3D points on the canvas
4. Dip the paint brush into paint
5. Move to every point generated for the shape in sequence
6. Repeat steps 2-5 for each shape in the parsed SVG file

Drawing a face: video - svg
Drawing a duck: video - svg