AprilTag Pose Estimation & Robot Program

Qingyuan Li

See demo video. Codebase is private.

Problem Statement

Create a robot program, controlled over local-area network, using AprilTags for pose estimation with specialized kinematics to adapt the pose to motor controls.

Methodology — Vision

1. Tools used

Vision targets are standard Tag36h11 AprilTags printed on Dibond, 16mm by 16mm with a 4mm white border for detection. They are mounted on a custom designed platform. For pose estimation purposes, the center of the platform is (0, 0, 0) in world coordinates, and the corners of each target are measured to 0.5mm relative to the center.

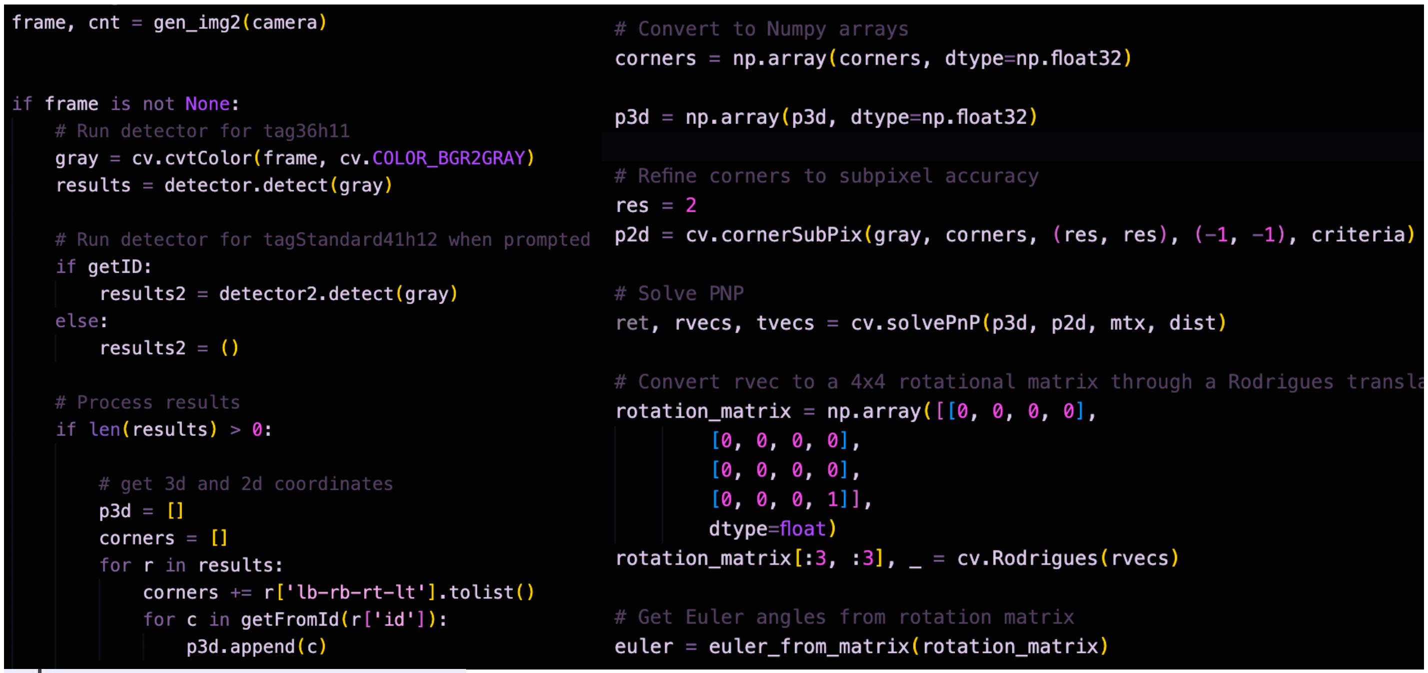
Note: the demonstration video shows an older iteration of the targets, with Apriltags printed on stickers. They are less accurate than Dibond due to distortion and misalignment, but still work decently.

Because my vision targets are small, I needed a relatively high resolution, low latency camera that was still inexpensive. I ended up using a standard 1080p Logitech webcam. All code runs on a Dell laptop with Ubuntu 20.04 LTS installed (for ease of package management).

2. Video Detection and Pose Estimation

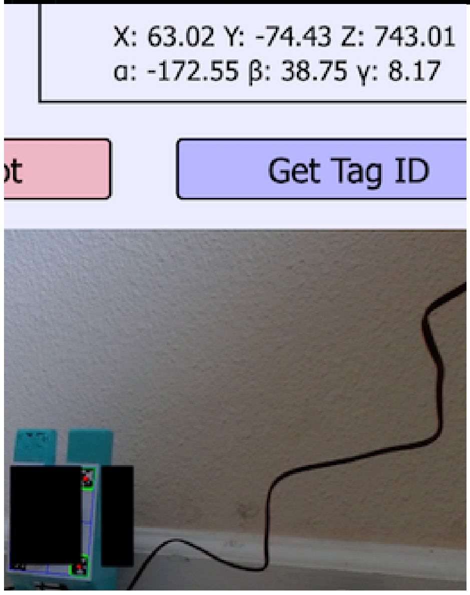
The computer vision itself was relatively simple to implement. I used the AprilRobotics apriltag library and OpenCV for general vision processing. The OpenCV VideoCapture class virtualizes the webcam. Once an image has been captured in matrix form, the apriltags library has a detector that returns the corners and ID of every AprilTag in the frame. After refining the corners to the subpixel using OpenCV's cornerSubPix algorithm, I match each corner's 2D camera coordinate with a pre-measured 3D world coordinate and use OpenCV's PNP (perspective-n-point) algorithm to solve for the location of (0, 0, 0) relative to the camera.

PNP returns rotation and translation vectors. To convert rotation vectors to a more usable form, I use the Rodrigues function get them as a 4x4 rotational matrix. Then, using ROS (Robot Operating System)'s function, I convert it to Euler angles ([α, β, γ]—[pitch, yaw, roll], relative to x, y, z axes). Now, I have the translational vectors [x, y, z] in millimeters and [α, β, γ] in radians.



```
frame, cnt = gen_img2(camera)

if frame is not None:
    # Run detector for tag36h11
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    results = detector.detect(gray)

    # Run detector for tagStandard41h12 when prompted
    if getID:
        results2 = detector2.detect(gray)
    else:
        results2 = ()

    # Process results
    if len(results) > 0:

        # get 3d and 2d coordinates
        p3d = []
        corners = []
        for r in results:
            corners += r['lb-rb-rt-lt'].tolist()
            for c in getFromId(r['id']):
                p3d.append(c)
```

```
# Convert to Numpy arrays
corners = np.array(corners, dtype=np.float32)

p3d = np.array(p3d, dtype=np.float32)

# Refine corners to subpixel accuracy
res = 2
p2d = cv.cornerSubPix(gray, corners, (res, res), (-1, -1), criteria)

# Solve PNP
ret, rvecs, tvecs = cv.solvePnP(p3d, p2d, mtx, dist)

# Convert rvec to a 4x4 rotational matrix through a Rodrigues transla
rotation_matrix = np.array([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 1]],
        dtype=float)
rotation_matrix[:3, :3], _ = cv.Rodrigues(rvecs)

# Get Euler angles from rotation matrix
euler = euler_from_matrix(rotation_matrix)
```



```
X: 63.02 Y: -74.43 Z: 743.01
α: -172.55 β: 38.75 γ: 8.17
```

Here are some results from testing. The measured distance is around 742±3 mm, while the PNP distance is 743±1mm. Angles also look reasonable, although they are much harder to measure. Given the relatively large distance compared to the tiny tags, these results are highly accurate given the available hardware.

3. Robot Program

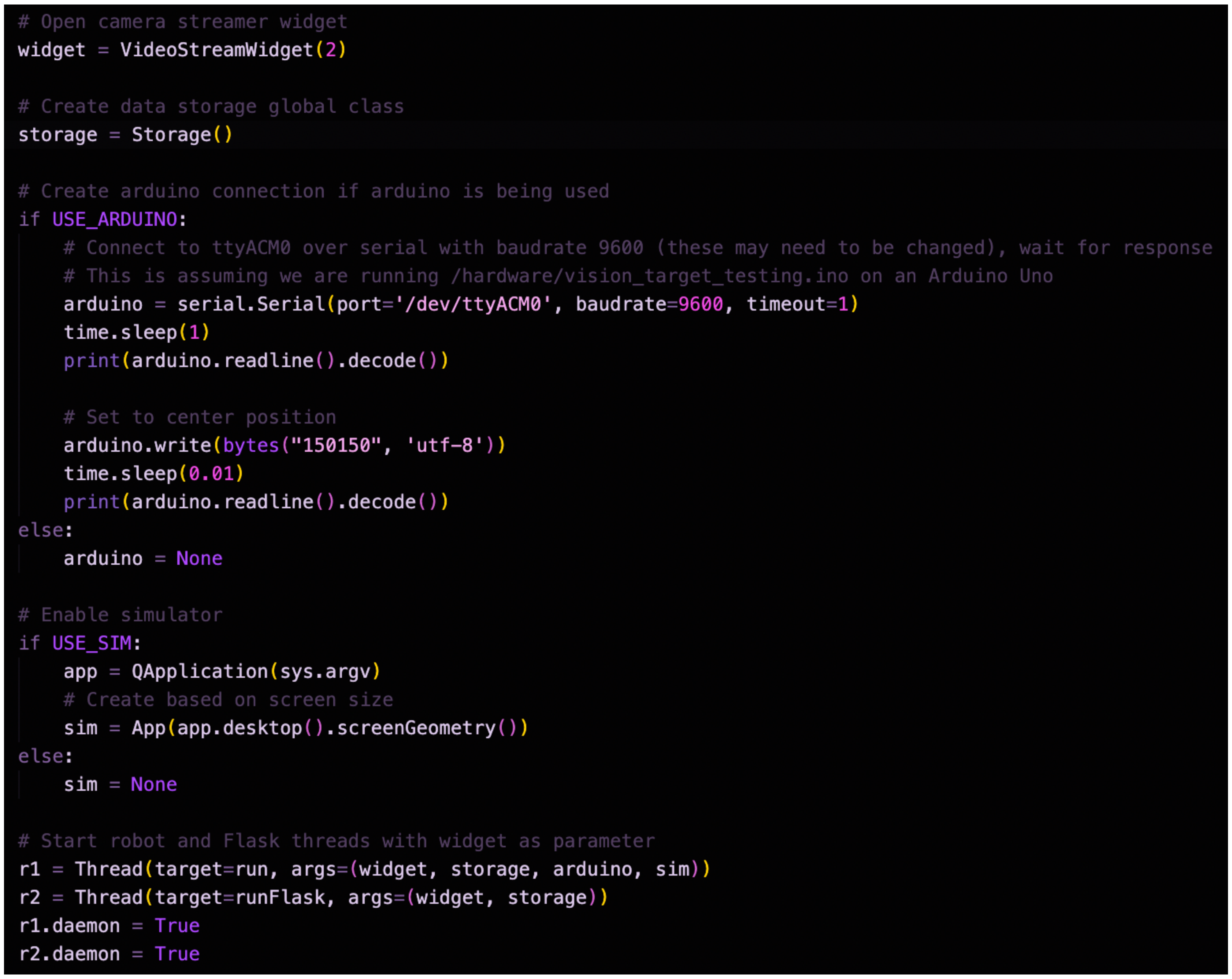
With the vision working, I needed a way to run a robot program that can be controlled over local area network (LAN) and can be combined with additional programs (such as a simulator, Arduino connection, debug screen, etc). I created two separate programs, a web application using Flask with Python, and a robot program that runs in an infinite loop at approximately 50 Hz.

Initially, I used JSON files to communicate between the programs. Unfortunately, I would often run into the issue of race conditions, when one of the programs would read the file while the other was writing to it, resulting in either incomplete information or crashing. I could not find a way to easily solve this problem; if it was really necessary, either using an external database with race condition protection or locking the other program while updating, such as with Python's *threading.Lock* with the two programs as separate threads, would likely have worked.

However, there turned out to be a couple of simpler solutions. Given the scenario of a separately running server and robot program, I successfully used sockets (a type of inter-process communication between processes that uses a specific port over LAN) to communicate; as one program would wait for the other to send data or a "confirmation" for necessarily synchronous processes. This completely solved the issue of race conditions. However, the latency created by using LAN was still an issue that could be improved upon.

I realized that the robot program and webapp, in practice, would always be run together. So instead of running them separately, I created another short script to run them asynchronously as daemon threads. This way, I could create a global "storage" object and pass it to both processes, which could then communicate by checking for any updates to the object. Although theoretically this still has a race condition, the speed of updating memory and (hopefully) some sort of builtin race condition protection should prevent the issue from arising.

The benefit to this method is that it allows me to trivially add new programs, for example a Serial port connection to the Arduino and a separate simulator app. Here is the "main" file:



```python
# Open camera streamer widget
widget = VideoStreamWidget(2)

# Create data storage global class
storage = Storage()

# Create arduino connection if arduino is being used
if USE_ARDUINO:
    # Connect to ttyACM0 over serial with baudrate 9600 (these may need to be changed), wait for response
    # This is assuming we are running /hardware/vision_target_testing.ino on an Arduino Uno
    arduino = serial.Serial(port='/dev/ttyACM0', baudrate=9600, timeout=1)
    time.sleep(1)
    print(arduino.readline().decode())

    # Set to center position
    arduino.write(bytes("150150", 'utf-8'))
    time.sleep(0.01)
    print(arduino.readline().decode())
else:
    arduino = None

# Enable simulator
if USE_SIM:
    app = QApplication(sys.argv)
    # Create based on screen size
    sim = App(app.desktop().screenGeometry())
else:
    sim = None

# Start robot and Flask threads with widget as parameter
r1 = Thread(target=run, args=(widget, storage, arduino, sim))
r2 = Thread(target=runFlask, args=(widget, storage))
r1.daemon = True
r2.daemon = True
```

4. (a) XY Intersection Demo

One milestone was to get a demonstration of using pose estimation to get the intersection of the normal vector of the AprilTags target and an arbitrarily defined plane. I used the origin coordinates relative to the camera (x, y, z) and Euler angles (α, β, γ).

To create the equation of the line, I converted Euler angles into a unit directional vector as per the definition of spherical coordinates, only with α and β flipped.

$$\vec{v} = \ < sin(\beta) \cdot cos(\alpha),\ sin(\alpha),\ cos(\beta) \cdot cos(\alpha) >$$

I could then define the line as two points:

$L_0 = (x,\ y,\ z)$ (original center)

$L_1 = (x + sin(\beta) \cdot cos(\alpha),\ y + sin(\alpha),\ z + cos(\beta) \cdot cos(\alpha))$ (center plus directional vector)

The plane is defined relative to the camera. I chose to use the representation of any three points on the plane, for ease of definition. I assumed here that the plane was just the XY plane of the camera and defined it as

$$< A,\ B,\ C > \quad = \quad < (0,\ 0,\ 0),\ (1,\ 0,\ 0),\ (0,\ 1,\ 0) >$$

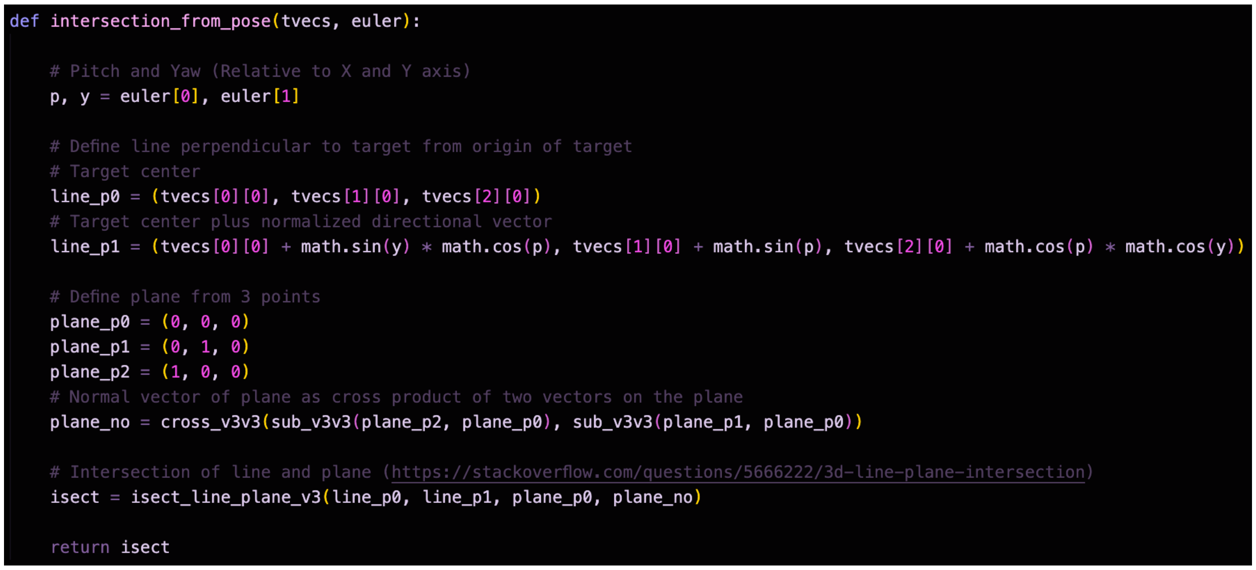
The algorithm I use for intersection of a line and plane takes in the normal vector of the plane, plus a point on the plane, as input. To convert $< A,\ B,\ C >$ to a normal vector, I can take the cross product of two of the vectors between the points (cross product geometrically represents the vector perpendicular to the intersection of the two vectors).

$$P_{Normal} = (B - A) \times (C - A)$$

Now I can run the algorithm (found on StackOverflow):

```
def isect_line_plane_v3(p0, p1, p_co, p_no, epsilon=1e-6):
    u = p1 - p0
    dot = p_no * u
    if abs(dot) > epsilon:
        w = p0 - p_co
        fac = -(plane * w) / dot
        return p0 + (u * fac)

    return None
```

My code using the algorithm:



```
def intersection_from_pose(tvecs, euler):

    # Pitch and Yaw (Relative to X and Y axis)
    p, y = euler[0], euler[1]

    # Define line perpendicular to target from origin of target
    # Target center
    line_p0 = (tvecs[0][0], tvecs[1][0], tvecs[2][0])
    # Target center plus normalized directional vector
    line_p1 = (tvecs[0][0] + math.sin(y) * math.cos(p), tvecs[1][0] + math.sin(p), tvecs[2][0] + math.cos(p) * math.cos(y))

    # Define plane from 3 points
    plane_p0 = (0, 0, 0)
    plane_p1 = (0, 1, 0)
    plane_p2 = (1, 0, 0)
    # Normal vector of plane as cross product of two vectors on the plane
    plane_no = cross_v3v3(sub_v3v3(plane_p2, plane_p0), sub_v3v3(plane_p1, plane_p0))

    # Intersection of line and plane (https://stackoverflow.com/questions/5666222/3d-line-plane-intersection)
    isect = isect_line_plane_v3(line_p0, line_p1, plane_p0, plane_no)

    return isect
```

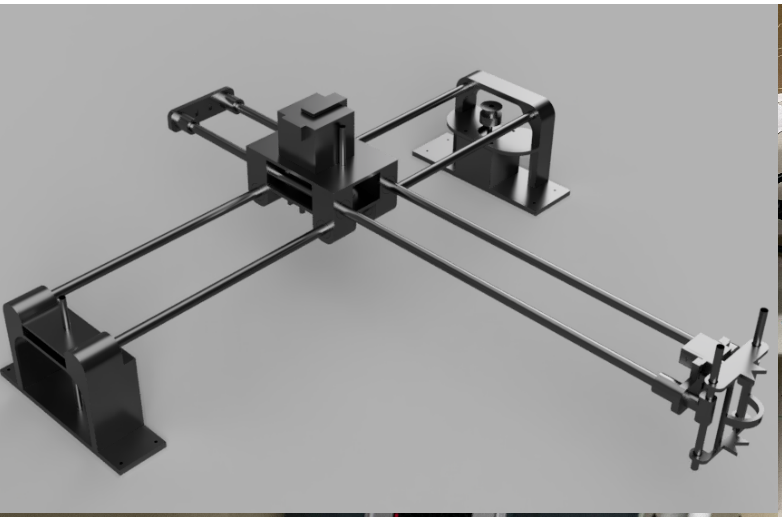
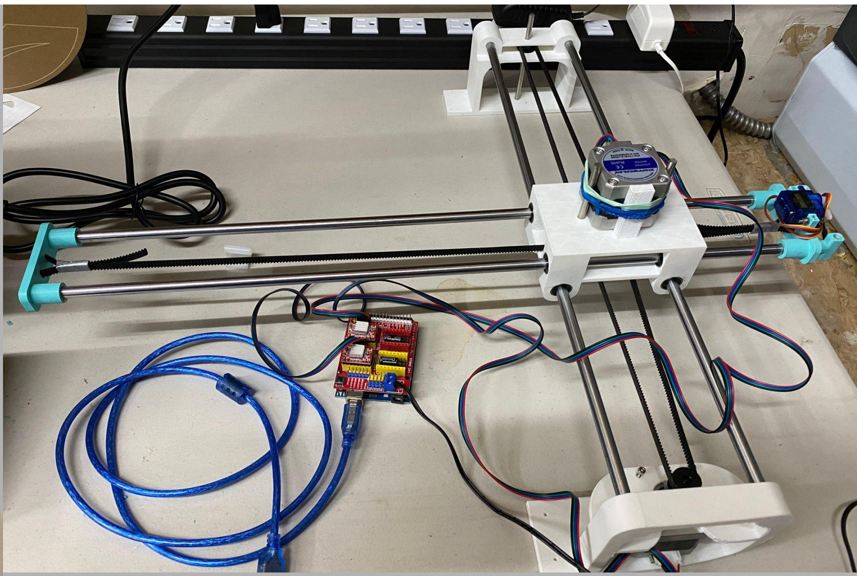
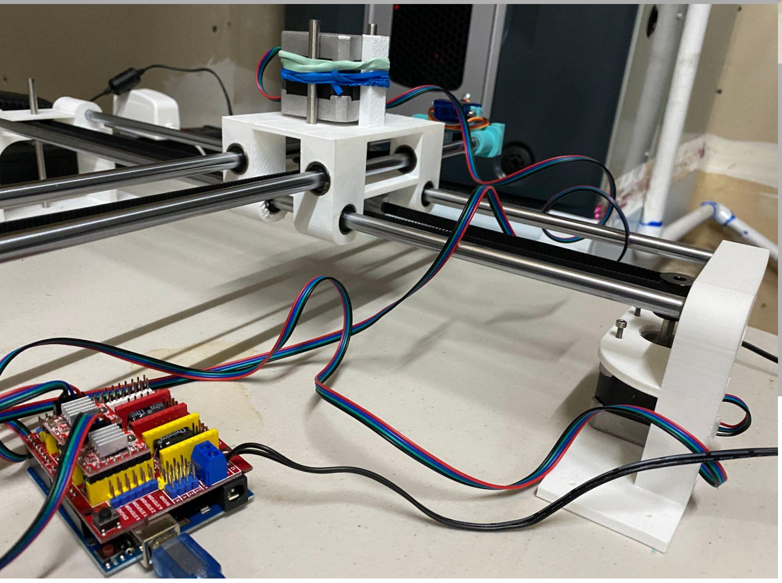
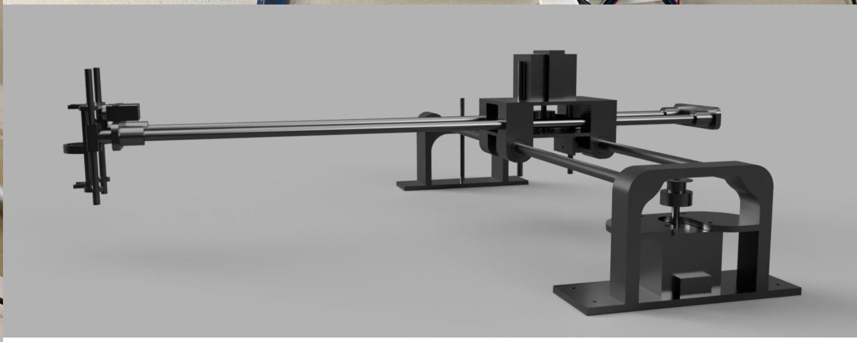
I now have the coordinate of the intersection in millimeters.

4. (b) XY Platform Build

I designed all of the XY linear rail platform in the demo video. It uses 6mm GT2 timing belts powered by NEMA-17 16x microstepping stepper motors, guided by 8mm linear rods and accompanying bearings. The mechanism is controlled by an Arduino Uno R3 with a CNC shield and Polulu A4988 motor controllers to drive the motors. All structural parts are created with CAD using Autodesk Fusion360 and then 3D printed in PLA with an Ender 3 Pro.

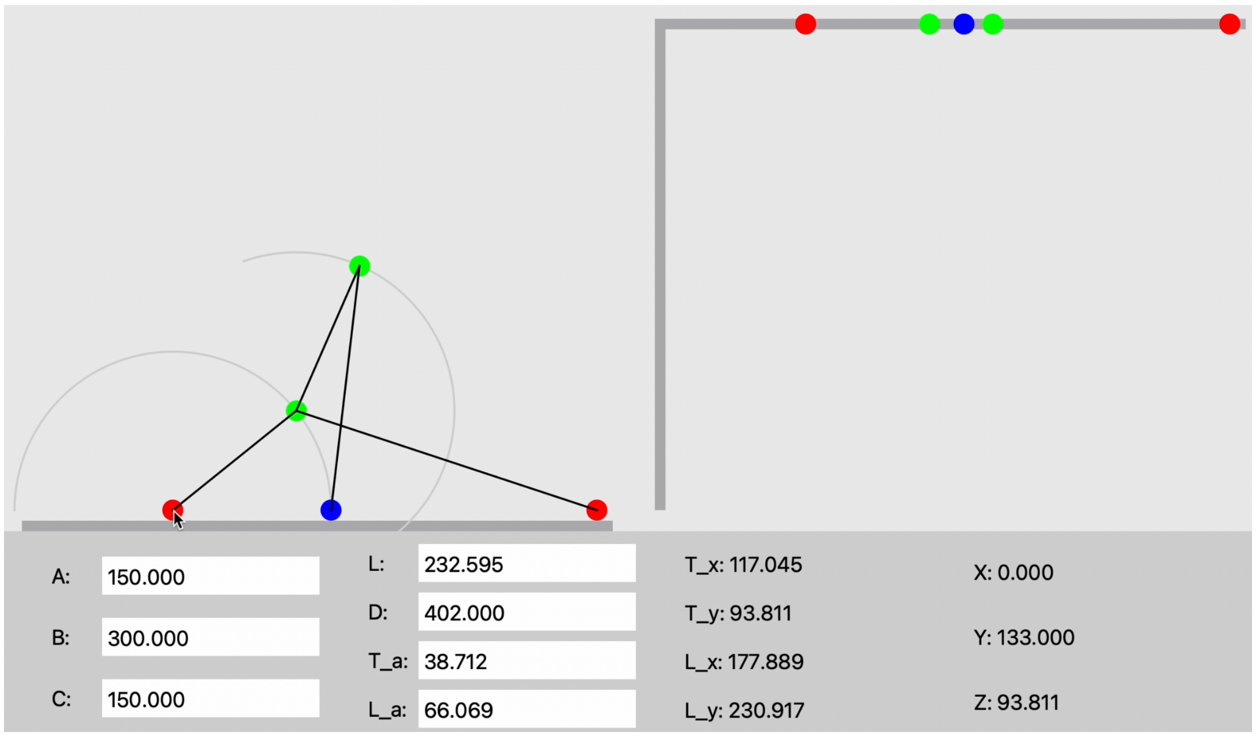
The robot program communicates with the Arduino via serial port. It sends a (x, y) coordinate in millimeters to the Arduino program every time I press a button on the webapp. The Arduino C program converts millimeters to steps and moves the mechanism accordingly.

Renders and the build are shown below.

5. Kinematics Simulator

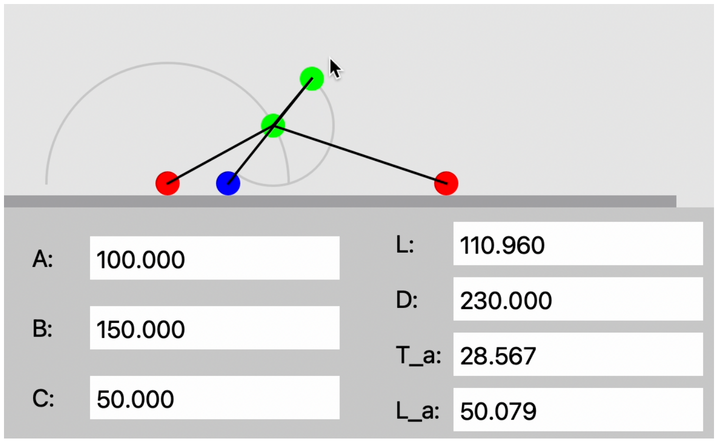
As another proof-of-concept, I created a kinematics simulator that runs based on inputs from the robot program. After trying PyGame, I settled on using PyQt5 (a Python port of the popular Qt software) for ease of creating the UI. The kinematics were just trivial trigonometry and geometry.



A:	150.000	L:	232.595	T_x: 117.045	X: 0.000
		D:	402.000	T_y: 93.811	
B:	300.000				Y: 133.000
		T_a:	38.712	L_x: 177.889	
C:	150.000	L_a:	66.069	L_y: 230.917	Z: 93.811

The red and green points can be dragged, and the display values on the bottom are edited accordingly using forward kinematics. The red points are bounded by the platform and each other, while the possible range of motion of the green dots are drawn.

The seven parameters can be changed with user input, and the rendered setup is edited accordingly using inverse kinematics. Bounding checks are used to prevent impossible configurations.



A:	100.000	L:	110.960
		D:	230.000
B:	150.000		
		T_a:	28.567
C:	50.000	L_a:	50.079