3. WRITTEN RESPONSES

3 a.

3.a.i.

This program aims to make it easier for a robot driver to pick up game pieces on the ground automatically. Which is faster than doing it manually. In the FRC 2023 Season, the front of the robot blocked the view of game pieces, making manual pickup harder.

3.a.ii.

The program works by allowing users to get the pose estimation of an object using vision, which grabs the translation, and rotation vectors of said object using OpenCV, detected by a color-tracking algorithm. This data is used to map a "linear trajectory" (constrained linear function), between the object, and the camera in a bird's eye view of the world. Simulating what a drivetrain path to an object would look like.

3.a.iii.

The inputs are two keystrokes, the 'u' key signified to track "cubes", and the 'o' key, for "cones". Another input is the object, present in reality, or in an image to track. The resulting output is a pose estimate of the object, represented by a drawn cube on the object, and text describing the translation and the rotation of the object, i.e. (x,y,z) position. As well as a cartesian graph, showing a simulated linear path to the object.

3 b.

3.b.i.

```
    points = np.array(
        [(0, 0), (pose[1][0].item(), pose[1][2].item())], dtype=np.float32)
    coefficents = traj.generateLinearTrajectory(points)
    traj.draw(coefficents, points)
```

3.b.ii.

```
def generateLinearTrajectory(points):
    secondList = points
    x = [secondList[i][0] for i in range(len(secondList))]
    y = [points[i][1] for i in range(len(points))]
    a, b = np.polyfit(x, y, 1)
    theta = np.degrees(np.arctan(a))
    secondTheta = 90 - theta
    return a, b, theta, secondTheta
```

3.b.iii.

The name of the list is "points", which is used as a parameter in the "generateLinearTrajectory()" function

3.b.iv.

The data in the list represents an array of tuples, in the form of (x,y) points in cartesian space.

3.b.v.

The "points" list is vital for the program, because the np.polyfit function used for polynomial regression (get function coefficents with a degree), requires a list of x and y values. Not a single numerical value. Otherwise a 1d vector error is thrown. Making it difficult to plot data accurately with a linear function. Additionally, using a list is more appropriate for polynomial regression because it can store large amounts of data in an organized manner, which can increase the accuracy of the regression algorithm when more data is appended to the list. Defining many tuples as seperate variables for the regression algorithm is convoluted, and more time consuming to implement compared to just appending values into a list from a dynamic dataset. Additionally, It's not possible to define new numeric variables during realtime data collection. So data that could be used for regression is just thrown out. So storing the data from the pose estimator into a list, is the simpliest and less-complex approach to plot a linear function from regression.

3 c.

3.c.i.

```python
def correctRotation(measurement, tvec, cap, poseInliers, minKalmanInliers, jacobian):

    kalman_filter = cv2.KalmanFilter(9, 3, 0)
    if cap.isOpened():
        dt = cap.get(cv2.CAP_PROP_POS_MSEC)/1000
        if not cap.isOpened():
            dt = 0.01
    if (measurement.shape == (3, 3)) or (poseInliers.shape[0] >= minKalmanInliers):

        measurement, _ = cv2.Rodrigues(measurement)
        measurement = measurement.astype(np.float32)
        measurementMatrix = np.eye(3, 9, dtype=np.float32)
        transitionMatrix = np.array([[1, 0, 0, dt, 0, 0, 0, 0, 0],
                                     [0, 1, 0, 0, dt, 0, 0, 0, 0],
                                     [0, 0, 1, 0, 0, dt, 0, 0, 0],
                                     [0, 0, 0, 1, 0, 0, 0, 0, 0],
                                     [0, 0, 0, 0, 1, 0, 0, 0, 0],
                                     [0, 0, 0, 0, 0, 1, 0, 0, 0],
                                     [0, 0, 0, 0, 0, 0, 1, dt, 0],
                                     [0, 0, 0, 0, 0, 0, 0, 1, dt],
                                     [0, 0, 0, 0, 0, 0, 0, 0, 1]], dtype=np.float32)

        processNoiseCov = np.eye(9, dtype=np.float32) * 1e-6
        measurementNoiseCov = np.eye(3, dtype=np.float32) * 1e-3
        errorCovPre = np.ones((9, 9), dtype=np.float32)
        statePre = np.zeros((9, 1), dtype=np.float32)
        jacobianRot = jacobian[:9, :3]
        multiple = np.dot(jacobianRot, measurementNoiseCov)

        X = np.dot(multiple, jacobianRot.T)
        X = X.astype(np.float32)
        errorCovPost = X

        statePost = np.zeros((9, 1), dtype=np.float32)

        kalman_filter.measurementMatrix = measurementMatrix
        kalman_filter.transitionMatrix = transitionMatrix
        kalman_filter.processNoiseCov = processNoiseCov
        kalman_filter.measurementNoiseCov = measurementNoiseCov
        kalman_filter.errorCovPre = errorCovPre
        kalman_filter.errorCovPost = errorCovPost
        kalman_filter.statePre = statePre
        kalman_filter.statePost = statePost

        for _ in range(1000):
            prediction = kalman_filter.predict()
            kalman_filter.correct(measurement)
            estimate = kalman_filter.correct(measurement)
            kalman_filter.errorCovPost = errorCovPost

        final_estimate = prediction[:3, :3]
        final_estimate = final_estimate.astype(type(tvec[0][0]))

        second_final_estimate = kalman_filter.statePost[:3, :3]
        second_final_estimate = second_final_estimate.astype(type(tvec[0][0]))

        third_final_estimate = estimate[:3, :3]
        four = kalman_filter.statePre[:3, :3]

        return final_estimate, second_final_estimate, third_final_estimate, four
```

3.c.ii.

```
if contours or len(contours) > 0:
    pose = getPose(contours)

    cv2.putText(filter, str(pose[1]), (50, 100),
                cv2.FONT_HERSHEY_COMPLEX, 0.25, (0, 255, 0), 1)
    cv2.putText(filter, str([np.degrees(angle) for angle in pose[0]]), (50, 200),
                cv2.FONT_HERSHEY_COMPLEX, 0.25, (0, 255, 0), 1)

    imagePoints, jacobian = cv2.projectPoints(
        axis, pose[0], pose[1], mtx, dist)

    correctRvec = correctRotation(
        pose[0], pose[1], cap, pose[2], 2, jacobian)[2]
```

3.c.iii.

The function "correctRotation()" inputs principally the rvec value returned by the pose estimator, and attempts to correct its faulty value with a Kalman filter, to provide a more accurate pose-estimate of the object in view of the camera. This contributes to the overall functionality of the program because it shows a more visually-accurate(tly rotated) drawn cube of the object by applying the new rvec to the cv2.projectPoints() function, indicating it has a more accurate pose.

3.c.iv.

The algorithm first uses selection, an if-statement to determine the best initial pose estimate before passing it into the Kalman filter. This is determined by inliers, a value that says how confident a pose estimate is. If a high amount of inliers is greater than the imputed Kalman inliers. The Kalman filter will proceed to correct that estimate. Or the if-statement checks if a measurement is available at all, by testing if it has a shape of (3,3), instead of (0,0). Next, the function proceeds to use a Kalman to predict, and correct the state-estimate with new rotation-measurements being passed in. This process is iterated by a for loop over 1000 times, because with each iteration. The filter refines the estimate with incoming measurements from the timestamp. Making a more accurate estimate of the new rotation matrix.

3 d.

3.d.i.

First call:

the first call is used to declare a variable, "correctRvec" as: "correctRvec = correctRotation(pose[0], pose[1], cap, pose[2], 2, jacobian)[2]"

Second call:

A second case call would be if the camera never detects an object, in this case the "correctRvec" values would be 0 with a matrix-shape of 0, additionally the number of inliers to validate a good pose would change from '2' to '20': "correctRvec = correctRotation[[0], [[0]], cap, pose[2], 20, jacobian)[2]

3 d.ii.

Condition(s) tested by first call:

the first condition of the call verifies if the pose[0] (measurement parameter) has a matrix shape of (3,3). Or if the inputed inliers surpasses the ones returned by the pose estimator. Which indicates that the inputed matrix has been populated with rotation data, so it can proceed to being corrected by the Kalman filter.

Condition(s) tested by second call:

the conditions tested by the second call, is the same as the first call. However because the matrix shape of the measurement parameter is 0, since the camera didn't track anything to create a pose estimate. It fails to pass through the if-statement. And isn't corrected by the Kalman filter.

3.d.iii.

Results of the first call:

The first call accesses the 2nd index of the returned tuple in the correctRotation() function, (correctRotation()[2]) it will return the "third_final_estimate". Which is the rotation estimate post of the timestamp, synchronized with the new measurements influencing the state.

Results of the second call:

Since the second call fails to pass through the if-statement, the function will return nothing since the returns are nested in the if-statement.