2023 Technical Binder

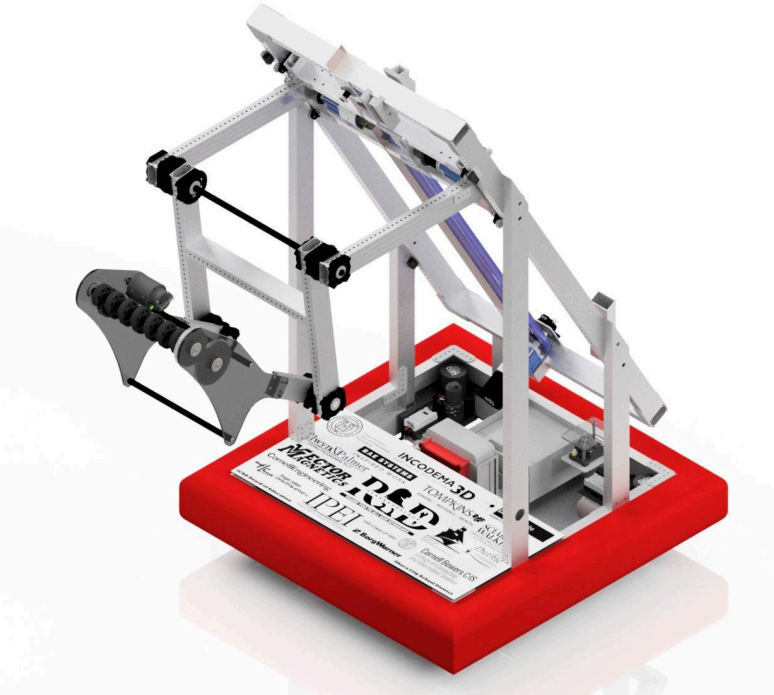


Table of Contents

Team

About

Code Red's first season was the 2001 *Diabolical Dynamics* season. This year marks our 22nd season. Code Red's mission is to inspire and empower STEM professionals. We do this by being **fully student-led**, and hosting and participating in community events.

Leadership

President	Emerson S.
Build Team Managers	Wali A. & Annalise T.
Administrator	Shaine W.
PR Officer	Jennifer Z.
Outreach Coordinator	Alex E.
Treasurer	Celene S.

2023 Code Red Robotics Officer Team

2023 Code Red Robotics Team Photo

Game Analysis

As per every year, Code Red's design decisions are fueled by initial game analysis and strategy prioritization. Charged Up provided uniquely fewer tasks than recent previous FRC games. The pick and place objective for both cube and cone game pieces made the two incredibly similar, and the charge station required no subsystems beyond the drivetrain. Due to the relative simplicity of this year's game, it was imperative for Code Red to create a reliable and consistent machine in the subsystems that did need to be designed.

Strategy and Ranking

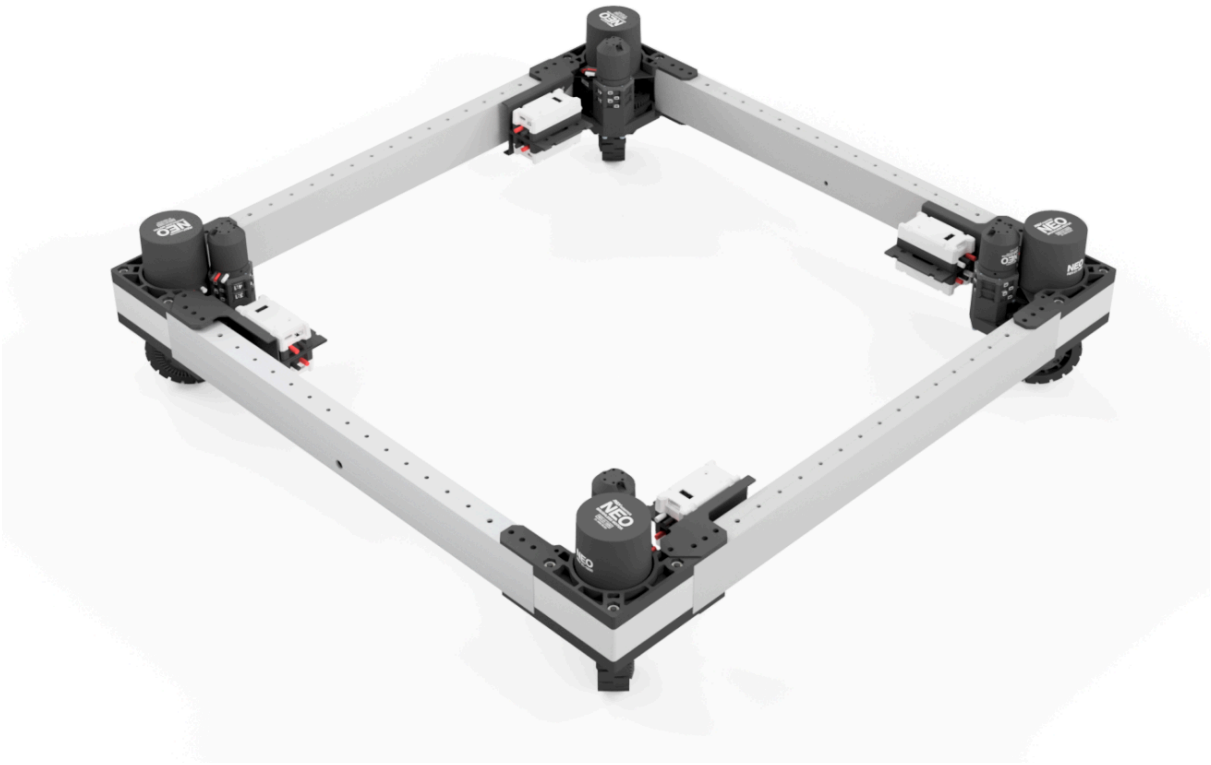
We determined that the most critical aspect of scoring points in order to win matches was to connect links. As such, we knew we wanted to create a robot that could handle both cones and cubes at all scoring levels. Charged Up's endgame challenge of balancing on the charging station was an area that we felt all teams could excel at with enough driver practice or programming efforts, so it was essential that we set ourselves apart with fast cycle times and high scoring teleoperated periods.

The large extensions both upwards and outside the frame perimeter of the robot meant that this game would be no stranger to tipped robots and dropped game pieces. We prioritized creating a robot that could fold into a compact robot with a low center of gravity to maneuver around the field quickly without having to worry about inertia taking over control.

Subsystems

Drivetrain

Head: Ziqi W. | Mentor: James Eddlestone



2023 signifies Team 639's first use of a COTS swerve drive. This decision was made to maximize maneuverability in the congested areas of the field such as the loading stations and community, as well as to be able to outmaneuver defensive robots.

Rev MaxSwerve Modules:

- NEO motor with a 4.71:1 gear reduction for a 15.76 ft/sec free speed
- NEO 550 for steering with a ~46.42:1 gear reduction
    - Custom 3D-Printed covers protect gears from unwanted external debris

Chassis:

- 26" x 26" Frame Perimeter to minimize space taken up in the community and on the charge station
- Solid 1/16" Aluminum belly pan for electronics, pneumatics and battery box

Elevator

Head: Ziqi W. | Mentor: James Eddlestone

Our elevator configuration provides our team with the extension to reach all three node levels as well as the flexibility to score and obtain game pieces from the ground, single sub-station, and double sub-station.

Tilted Elevator:

- Fixed Elevator at a 60˚ tilt.
- Carries Elevator Carriage from top to bottom
- Top of elevator resides very close to both maximum starting height and the edge of frame perimeter to get as close to scoring nodes as possible
- As the elevator carriage travels up the tilted elevator, it gets closer to the scoring nodes in both height and horizontal distance

Elevator Carriage:

- Powered by a NEO motor with a 25:1 gearbox reduction
- Travels up and down tilted elevator with #25-H chain featuring an inline chain tensioner
- 2 x 1 x 0.1” Aluminum 6061-T6 Tube sliding on COTS elevator bearings
- Holds both Falcon 500s for powering the double jointed arm

Arms

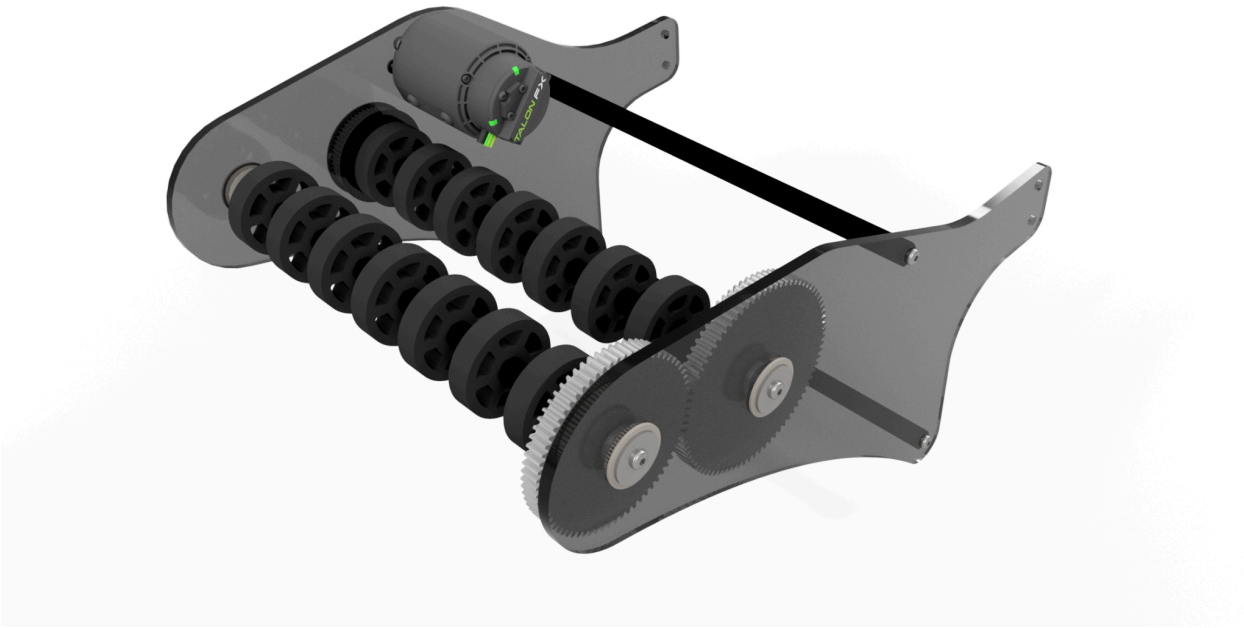
Head: Wali A. | Mentor: Doug Armstead

Double Jointed Arm:

- First joint powered by a Falcon 500 with a 100:1 gearbox reduction allowing rotation about end of static member using #25-H chain and sprocket(16:40)
- Second joint with end-effector powered by a Falcon 500 with a 20:1 gearbox reduction allowing rotation about end of “elbow” using #25-H chain and sprockets(24:38)
- 2 x 1 x 0.04” Aluminum 6061-T6 tube for all the arm members to minimize weight compounding and allow for fast movement
- Members connected using ½” Aluminum 7075 Round Hex Shaft and sliding bearing blocks with 1.125” round hex bearings for ease of assembly and disassembly
- Sliding bearing blocks feature WCP Cams for quick adjustment of #25-H chain tension in between matches

End Effector

Head: Wali A. | Mentor: Adam Newhouse



Cone Intake:

- Our cone intake utilizes two rollers of 2” compliance wheels spinning in opposite directions using a 68:80 tooth spur gear mesh
- The rollers are powered by a Falcon 500 with a 3.5:1 belt reduction
- Grabs cones by the top and actively rolls them up
- Intakes cones from the ground that are standing up as well as cones on the double substation tray

Cube Intake:

- Our cube intake uses the lower of the two cone rollers to grab cubes and store them in our end-effector
- Hex shafts provide compression for obtaining and maintaining control of cubes through a cycle

Programming

Head: Nathan B.
Tilden S.

Mentor: Hadas Kress-Gazit
Mentor: Zach Stillman

Control

In our robot code, we follow the Command Based Framework. For a quick overview, all functions of the robot are handled in subsystem classes. More complex functions are abstracted into commands, which are run by the command scheduler. The command scheduler keeps track of which commands need to be run and for how long. Here's an example: The Elevator subsystem is where the elevator, elbow, and wrist motors are initialized. It contains methods like setElevatorPosition and setArmPosition that take in a double in meters or radians and use the motors' set method. In the setArmPosition command, the desired elevator, elbow, and arm positions are passed into the constructor. We set up a motion profile based on those values and pass in the setpoints to setElevatorPosition and setArmPosition. With the command based structure, each motor or piston is controlled by only one subsystem, and only one command is allowed to run at a time per subsystem, meaning that no confusion arises from motors being set to different values from different places.

The entire elevator subsystem consists of the elevator and two arm joints, which are operated by one motor each. We also have two magnetic limit switches at the top and bottom of the elevator to ensure the carriage doesn't harm the frame. The subsystem is controlled by 3 PID controllers, one for each motor. For the two arm joints, we run them on the controller themselves as there are no external encoders. This has the benefit of updating much faster and being more accurate than using an external encoder.

To tune the PID constants, we followed a similar pattern each time. For each motor we made sure to put the position in encoder ticks and voltage being applied. For the joints we zeroed the encoders when it was parallel to the ground and rotated it by hand until it was perpendicular. The encoder value displayed when perpendicular would allow us to calculate the encoder ticks per degree and radian. For the elevator it was a similar process, but instead of rotation we measured 10cm and used that encoder value. After we had the conversion factor we needed to find the feedforward term, or the voltage needed to overcome gravity. For the elevator, this is a constant, and we measured it by keeping the carriage stationary by binding a joystick to the elevator and keeping track of the approximate voltage applied. We also did this for the two joints, but with these the feedforward term would vary depending on the angle. The feedforward term can be calculated as the maximum feedforward times the cosine of the angle, so we measured the necessary feedforward when each joint is parallel to the ground and incorporated that into the PID controller. For tuning the constants we found we only needed proportional gain, and to find that number we started at 0.00001 and worked our way up until there was visible change. Then, all we had to do was make small adjustments until it was fast and accurate.

In addition to PID controllers, we use trapezoidal motion profiling to more smoothly interpolate between the current point and the desired position. When looking at each motor, the graph of the velocity will look like a trapezoid, with a constant acceleration to the max velocity, then a constant deceleration to 0. In order to have accurate encoders, we needed a way to ensure the elevator started each

match in the same position. Initially, we had a calibration routine that would use limit switch data to find the bottom, but we found a stable starting position that wouldn't require that.

This year was the first year that our team has used a swerve drive system. Swerve drive consists of 4 swerve modules, each with a motor that controls the speed of the wheel and a motor that turns the wheel. This is vastly different from west coast drive, which is what we have used in years prior. West coast drive has two motors that each control a side of the robot. We used the example code that came with the swerve modules we bought. The SwerveModule class is a wrapper class for each individual module, and the DriveTrain subsystem uses 4 instances of SwerveModule to control the drive base.

Also in DriveTrain is a Kalman filter. A Kalman filter takes in data from the odometry, like rotation of the gyroscope and encoder data from the modules, and visions data. From three cameras we get their best guess of the location and rotation of the robot based on april tags. This process is described in more detail in the Visions section. The filter is called a SwerveDrivePoseEstimator in WPILib and it takes in two vectors of standard deviations. These represent how certain the filter should be about the measurements it is being given. A higher number means less certain, a lower number means more certain. Every run of the scheduler, which is about 20 times a second, the filter is updated with odometry and vision data if the cameras can see any tags. This is extremely useful for getting accurate estimates of the robot's position on the field.

```
// Kalman filter for tracking robot pose
SwerveDrivePoseEstimator poseEstimator = new SwerveDrivePoseEstimator(
    Constants.Drive.driveKinematics, // kinematics
    Rotation2d.fromDegrees(gyro.getAngle()), // initial angle
    new SwerveModulePosition[] { // initial module positions
        frontLeft.getPosition(),
        frontRight.getPosition(),
        backLeft.getPosition(),
        backRight.getPosition()
    },
    new Pose2d(x: 0, y: 0, Rotation2d.fromRadians(radians: 0)), // initial pose
    VecBuilder.fill(n1: 0.1, n2: 0.1, n3: 0.1), // odometry standard deviation for x, y, theta
    VecBuilder.fill(n1: 0.5, n2: 0.5, n3: 0.5) // visions standard deviation for x, y, theta
);
```

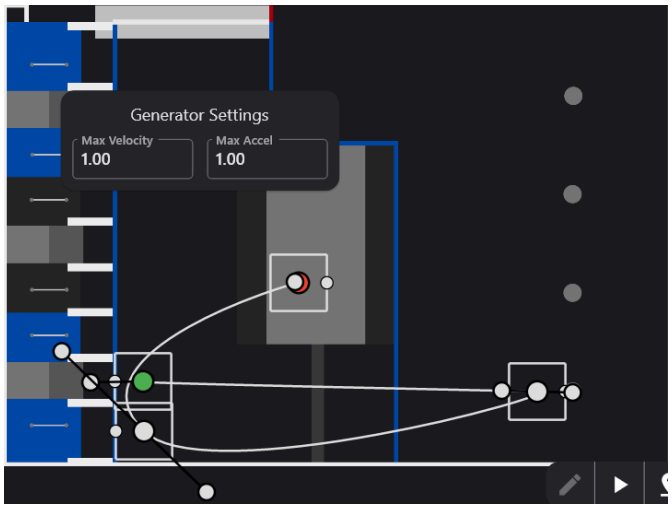
To actually drive the robot around, we used the code provided to us by the company we bought the modules from. It incorporates slew rate limiting, which provides finer control of the robot by limiting the rate of change of the current being sent to the motors. It functions by first converting the current angular and linear velocity of the robot into magnitude and direction, as we are using swerve drive and we have to take into account the x and y speeds. It then calculates the slew rate, or rate of change, of the direction and magnitude. Then, the slew rate limiters adjust those values to a reasonable value and they are converted back into x, y and theta. ChassisSpeeds.fromRelativeSpeeds() is a method that takes in x, y and theta values and converts them to SwerveModuleStates which are fed into the modules as setpoints.

Spindexer™

After a cone or a cube comes through our acquisition, we need a way to orient it so the arm can effectively pick it up. Our solution was the spindexer™, a spinning disc to orient the cone. We have a camera mounted on the frame directly above the spindexer™ which can detect whether there is a cone or a cube, and if it is a cone, what angle it is relative to the robot. The method by which we do this is more thoroughly explained in the Visions section. The angle is sent over serial to the roborio, so using the supplied JeVoisInterface class the angle is put onto NetworkTables. This is used in our code to supply the setpoint for a PID loop controlling the spindexer™ motor.

Autonomous Modes

This year was the year that we switched from PathWeaver, a tool provided by WPILib, to another tool called PathPlanner. We found that PathPlanner was much more intuitive and easier to create smooth and accurate paths. It also includes a holonomic mode, which allows us to be more efficient with our paths.



Autonomous Operation during Teleop Play

PathPlanner comes with a library called PathPlannerLib that enables us to do on-the-fly trajectory generation. We simply pass in the relative pose that we want to reach and it generates a profile that our robot can easily follow. Our DriveTrain subsystem includes a method that takes in a trajectory and generates a command group that represents the path. Inside that command group is a PPSwerveControllerCommand which is provided by PathPlannerLib. We pass in a trajectory, along with more data from the drive subsystem, and it creates a command that controls the modules to follow that path.

```java
public Command followTrajectoryCommand(PathPlannerTrajectory traj, boolean isFirstPath) {
    return new SequentialCommandGroup(
        new InstantCommand(() -> {
            // Reset odometry for the first path you run during auto
            if(isFirstPath){
                resetOdometry(traj.getInitialHolonomicPose());
            }
        }),
        new PPSwerveControllerCommand(
            traj,
            this::getPose, // Pose supplier
            driveKinematics, // SwerveDriveKinematics
            new PIDController(kp: 0, ki: 0, kd: 0), // X PID controller
            new PIDController(kp: 0, ki: 0, kd: 0), // Y PID controller, probably the same as X controller
            new PIDController(kp: 0, ki: 0, kd: 0), // Rotation PID controller
            this::setModuleStates, // Module states consumer
            useAllianceColor: true, // mirrors path based on alliance
            this // Requires this drive subsystem
        )
    );
}
```

To generate on-the-fly trajectories, we need a target position. We get this from various sources depending on the goal. We get data from visions describing the pose of the nearest cube and cone, and we know the locations of important positions around the field such as the double substation and scoring positions. Given a target pose, the angle between the current and target poses is calculated with the x and y components of the difference in position.

To create a trajectory, we need speed and acceleration constraints, and a starting point and ending point. The constraints are stored as constants. A PathPoint object takes in the current translation2d, the heading or rotation of the wheels, and holonomic rotation, the rotation of the actual robot. The starting point is passed the current position calculated by the previously described Kalman filter, the calculated angle between the current and target as the heading, and the current rotation from the Kalman filter as the holonomic rotation. The end point is given the target position, the same heading as the start, and the target rotation.

```java
@Override
public void initialize() {
    Pose2d targetPose = new Pose2d(new Translation2d(x: 14.66, y: 3.85), Rotation2d.fromDegrees(degrees: 0)); // top node on red
    Translation2d robotPosition = driveTrain.getPose().getTranslation(); // current position
    Translation2d translationDifference = targetPose.getTranslation().minus(robotPosition); // difference between target and current
    // calculates wheel angle needed to target from x and y components
    Rotation2d translationRotation = new Rotation2d(translationDifference.getX(), translationDifference.getY());
    Command driveCommand = driveTrain.followTrajectoryCommand(PathPlanner.generatePath(
        new PathConstraints(maxSpeed, maxAcceleration),
        new PathPoint(robotPosition, translationRotation, driveTrain.getPose().getRotation()), // starting pose
        new PathPoint(targetPose.getTranslation(), translationRotation, targetPose.getRotation())), // ending pose
        isFirstPath: false);
    driveCommand.schedule();
}
```

Vision

Head: Gene W. Rocky S.	Mentor: Bradford Smith Mentor: Hadas Kress-Gazit Mentor: Zach Stillman

Game Piece Detection

To acquire game pieces, we first need to detect them, which we have done using a RealSense D435 RGB-D camera. The main reason we are using this is because it can detect the distance to objects, which we can use to find their coordinates in 3d space more easily.

The setup of the library for this camera is tricky as the library we're using, *pyrealsense2*, only supports python 3.7 to 3.8. If you don't have the correct python version, you will have to set a **virtual environment** for it .
To create a virtual environment:

- https://www.python.org/downloads/ download python 3.7
- Open command prompt(for windows)
- Type following commands in terminal:
  pip install virtualenv
  virtualenv My_env --python=python3.7
  My_env/Scripts/activate.bat
  pip install pyrealsense2
  pip install opencv-python
- Exit the terminal and you'll see a folder in users called "My_env".
  #to check python version of the virtual environment, open the folder in terminal use : *python - -version*

When using the camera, we got both a color image and a depth map of each frame, which we could use to detect game pieces on the color image, and determine their location relative to the robot using the depth map.

First, we look at the color frame and figure out which objects in it are likely to be cones and cubes. We pass the image through a series of filters. The first filter is a color filter, which we have two separate ones, one for the yellow cones and one for purple cubes.

After this, we use OpenCV to find all the contours on each filtered image, get their convexHulls, and find the centers of mass of these hulls. For each one, the code then checks whether it is roughly the size of the game piece .

- First, it uses the minAreaRect function to find a rectangle that can fit to the hull, and checks whether the aspect ratio of this rectangle is greater than a minimum value.
- Then, using the depth map, which will be discussed in the next paragraph, it computes the actual size of the pixel at the center of the hull, and uses that to approximate the perimeter of the hull based on the length in pixels. This can then be used to filter out anything whose perimeter is too big or small to be a cone/cube.

Once the code has a set of objects that it thinks are game pieces, it has to find the coordinates of each one relative to the robot and then provide data for the robot. This is where the depth map comes in, but to get it to work well, a few things need to be done.

- First, the RealSense has different resolutions for the color image and the depth map, so a *pyrealsense2* function needs to be used to align them. We also need another function to fill holes in the depth map.
- Reminder : because this happens before the alignment, the color image will also have holes if aligned to the depth map, making it important to specifically align the depth map to the color image.

Using the combined color and depth maps, we can find the 3d coordinates of the game piece relative to the camera. However, a few transformations need to happen to convert to robot coordinates.

- First of all, since the camera is rotated 90° for a larger vertical field of view, the coordinates also need to be rotated.
- Also, the camera is angled downward, so its reference frame is different from the robot's, so we had to use trig to transform the coordinates. Since the camera is at a fixed angle to 30°, we had enough information to do this.

We now have the coordinates of all cones and cubes seen by the camera, but need to send them to Programming. To do this, we used the networktables library.

- First, we connect to a network, and start a "Detector" table.
- Then, for each frame, we can push two lists of floats, one with the key "Cone," and one with the key "Cube."
- The lists contain the coordinates of the closest game piece of their respective types, using the format :
  [ObjectSeen (1.0 or 0.0 to indicate whether the camera sees anything), ObjectX (left-right), ObjectY (up-down), ObjectZ (forward/backward)].

Game Piece Orientation

- First, we have to identify game pieces based on color using JeVois camera
- If cargo is identified as a cone, draw contours around the game piece on the color frame and record the contour points. Then find the center of the contour points
- Approximate the contour points to a polygon and find the vertex point that has the smallest angle, which is the tip of the cone.
- After that, draw an arrow from the center of the cone to the tip of the cone, and record the angle of the offset of the cone. Finally, send the information through the serial port to networktables.

AprilTag Detection

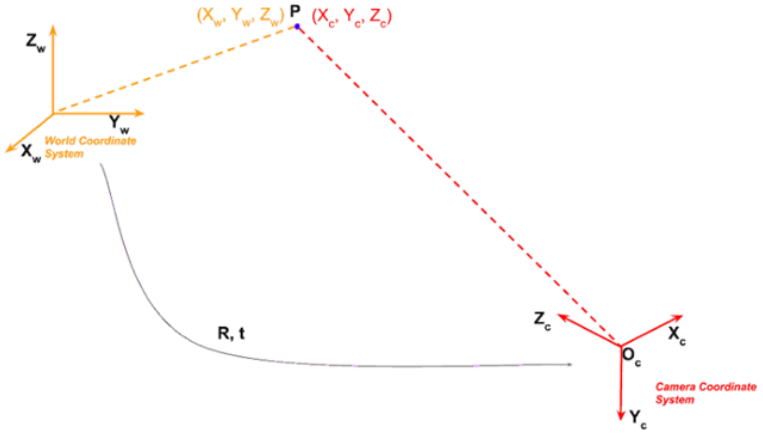
The 2023 Game introduced a new fiducial as a field element: the Apriltag. The field has 8 Apriltags, each with a unique ID. This means that we can now use a vision system to complement our odometry and get closer to accurate global positioning.

The First step was to build the AprilTag library on our coprocessor. Once it was built, we started programming using the python bindings. The most important part of the library is the Detector class. The Apriltag detector can be instantiated with several parameters including maxHamming distance and blur that may affect the quality of your detections. In our code, we made the maxHaming parameter equal to 0 and the blur parameter equal to .1. With these parameters we have no phantom tags, making our detections extremely robust.

Multi-Camera Global Positioning

Camera Calibration

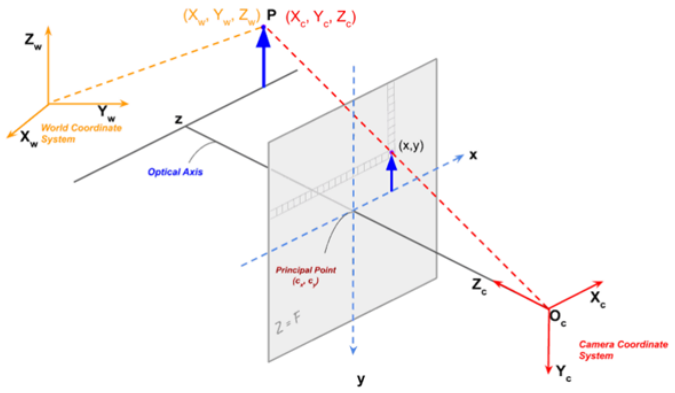
Camera calibration is the key to solving for pose. If we know exactly how 3d world points are projected onto the image frame, we can solve for our position. Consider a point in 3D space, the origin of which is placed arbitrarily. We have a camera at point Oc with a rotation and translation represented by R and T respectively. In our case, the x-axis points to the right, the y-axis points down, and the z-axis points out the front of the camera.



Using simple transformation matrices, we can get the point in the world coordinate system in the camera's coordinate system:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_c = [R|T] * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_W$$

With the point in the camera's coordinate system we can now project it onto the image plane.



The distance the image plane is from the camera origin is the focal length. With our knowledge of similar triangles, we can solve for coordinates x,y with

$$x = f_x \frac{X_c}{Z_c}$$

$$y = f_y \frac{Y_c}{Z_c}$$

The different focal lengths for the x and y axes are due to the fact that the camera's horizontal and vertical focal lengths may not be the same resulting in non-square pixels. Now, there are also other factors that need to be considered. What if there is a skew between the plane and the sensor? What if the optical center does not lie directly in front of the sensor? We can implement all these changes in the following matrix. To get the image points, we simply divide the resulting x and y by z.

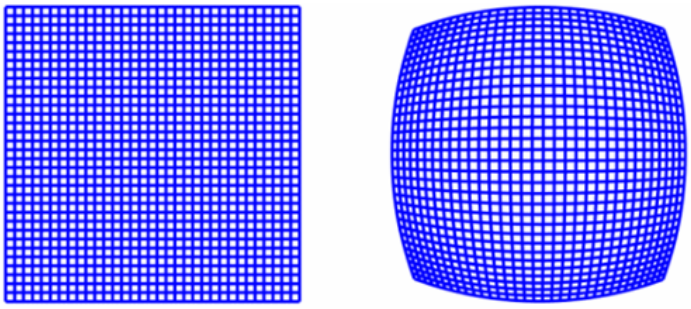
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix}$$

$$X_{img} = x/z$$

$$Y_{img} = y/z$$

This is the intrinsics camera matrix. It represents all the camera parameters we need in order to project an image onto a camera plane. However, another factor impacts our projections: distortion.

OpenCV describes two types of distortion: radial and tangential. Radial distortion results in a fisheye effect.



Radial distortion can be represented by

Tangential distortion can be represented by

$$x_{distorted} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$
$$y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2xy]$$

Basically, along with the intrinsic camera matrix, we need to find the distortion coefficients, k1, k2, p1, p2, and k3. OpenCV's calibration tool does this for us. We take pictures of a chessboard pattern so that it can generate the constants.

Solving for Pose

With the 3D-2D correspondences from apriltag corners, the intrinsic camera matrix and the distortion coefficients, we are ready to solve for pose. Using our equation from earlier, pose can be represented by a rotation matrix augmented by a translation vector. It is the matrix used to convert the coordinates of a point in the world coordinate system into the coordinates in the camera's coordinate system.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} * [R|T] * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_W$$

$$X_{img} = x/z$$

$$Y_{img} = y/z$$

Passing in the 3D-2D correspondences of apriltag corners, the camera matrix, and distortion coefficients into solvePnP, we get a rotation vector and a translation vector. With this rotation vector, we then get the rotation matrix from OpenCV's Rodrigues function.

The translation vector represents the translation from the origin of the camera's coordinate system to the origin of the world coordinate system. We want the inverse of that: the translation from the origin of the world coordinate system to that of the camera. We can invert this translation by negating the translation vector. Essentially, we are shifting our coordinate system the same amount from the origin but in the opposite direction. The same concept applies to rotation. Even after negating the values in the translation vector, the translation it represents is still with respect to a rotated coordinate system. We need to invert the rotation. We could do this by simply

negating the angle, but that is the unknown we are solving for. Therefore, we invert the rotation by transposing the rotation matrix. After that, we multiply the transposed rotation matrix by the negated translation vector, giving us the coordinates of our camera:

```
mmat, rvec, tvec = cv.solvePnP(
    objectpoints,
    cornerpoints,
    cmtx,
    dist,
    )

tvec = (np.array(tvec))
rvec = (np.array(rvec))

rotationmatrix, _ = cv.Rodrigues(rvec)

final_coords = np.dot(-rotationmatrix.T, tvec)
```

To find the angle, we simply apply the rotation to an arbitrary point, and project it onto a certain axis, and solve for the angle. For example, we wanted to solve for yaw (rotation around the z axis) so we applied the rotation to the point (1, 0, 0), projected it to the x-y plane, and solved for the angle.

```
pointX, pointY = [pointCoords[0][0], pointCoords[1][0]]



# Signs of x and y coordinates on unit circle
sx = 1 if pointX <= 0 else -1
sy = 1 if pointY <= 0 else -1
# # Modify theta based on coordinate quadrant to compensate for arctan only going from -90 to 90
ztheta = math.degrees(math.atan(pointCoords[1][0]/pointCoords[0][0])) + (180*sy)*(sx - 1)/(-2)
ztheta -= 90
if ztheta < 0: ztheta += 360
```

Pose Translations
We want the position and angle of the robot. Not the position and angle of the camera.

To translate the position, we measure the coordinates of the center of rotation of the robot in the camera's coordinate system. Since we know the angle and position of the camera in the world, we can get the coordinates of the robot in the world coordinate system.

To translate the angle, we simply add the angle of the robot relative to the camera from the angle of the camera to the world.

A transformation matrix will do.

$$\begin{bmatrix} x_r \\ y_r \\ 1 \end{bmatrix}_G = \begin{bmatrix} \cos\theta_c & -\sin\theta_c & x_c \\ \sin\theta_c & \cos\theta_c & y_c \\ 0 & 0 & 1 \end{bmatrix}_G \begin{bmatrix} x_r \\ y_r \\ 1 \end{bmatrix}_C$$

Sending values
We send five values over to network tables:

ntags	Number of tags the camera sees
rx	X position of robot
ry	Y position of robot
theta	Rotation of robot
time	Time the frame was taken

Deployment

To get the processes to begin on startup, we created a new systemd process on our vision PC.

- cd /etc/systemd/system
- Touch vision.service
- Vim vision.service
- Paste the following:

```
[Unit]
Description=Vision process

[Service]
User=crr
Restart=always
WorkingDirectory=/home/crr/2023Visions
ExecStart=/bin/python3 /home/crr/2023Visions/positioning/apriltagtesting.py -h

[Install]
WantedBy=multi-user.target
```

-
- Systemctl daemon-reload
- Systemctl status vision.service
    - Should be active!