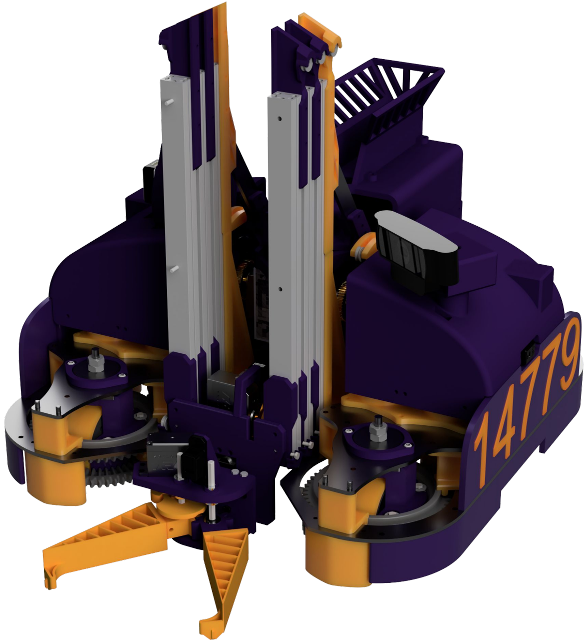
Swerve Drivetrains

Presented By Spontaneous Construction

FTC 14779

Table of Contents



Questions? Post them in the Q&A and we'll answer them at the end of the presentation

Recording and presentation will both be available on our website later today

Who We Are

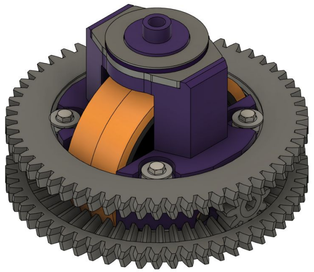
- 5th year team
- Won 1st Inspire at Minnesota State Championship
  - Come visit us at worlds!
- Kiwi drive at Chicago Robotics Invitational 2022
- Using a Differential Swerve drive in 2023

What is a Swerve Drive?

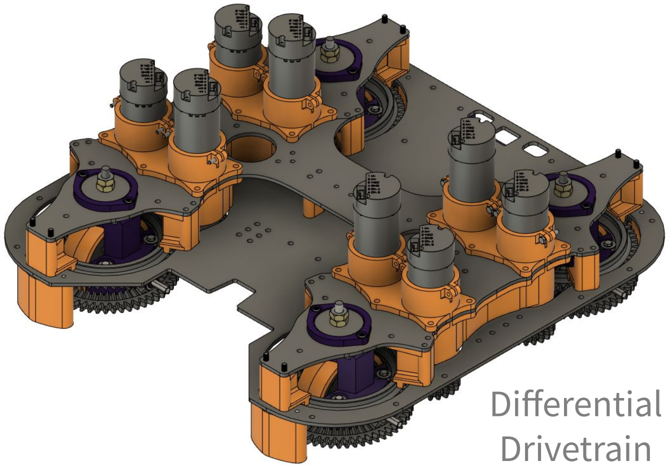
- It is a drivetrain with multiple pods that are able to rotate independently
- A pod is the central rotating part that contains the wheel and the gears
- Drivetrain movement is a combination of all pod movements

Two Types:

- Coaxial Swerve
- Differential Swerve



Differential Pod Example



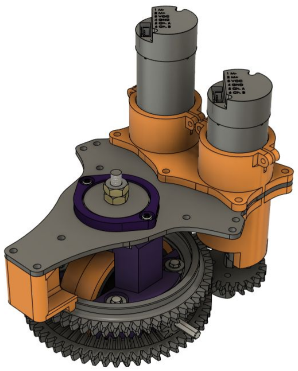
Differential Drivetrain

What is What

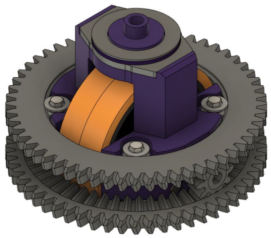
Module: A self contained unit containing the pod and the motors required to turn it.

Pod: The central part of the Module including the gears and wheel

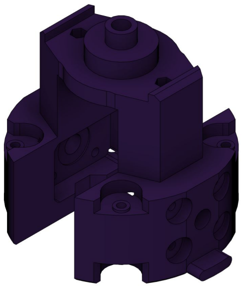
Structure: The purple pla part that holds the gears and wheel



Module



Pod



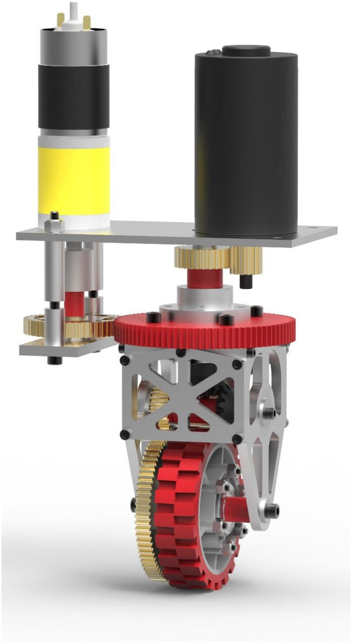
Structure

Why Use Swerve

- Because Swerve is so uncommon in FTC, using it will give you a boost for Innovation

Swerve:

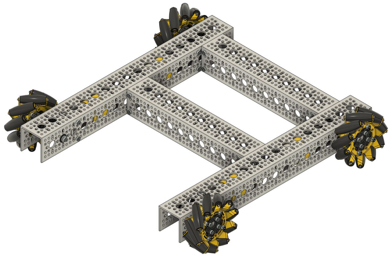
- Stronger
- More complex
- More innovative
- Better defense
- Reliable



Coaxial Swerve

Mecanum:

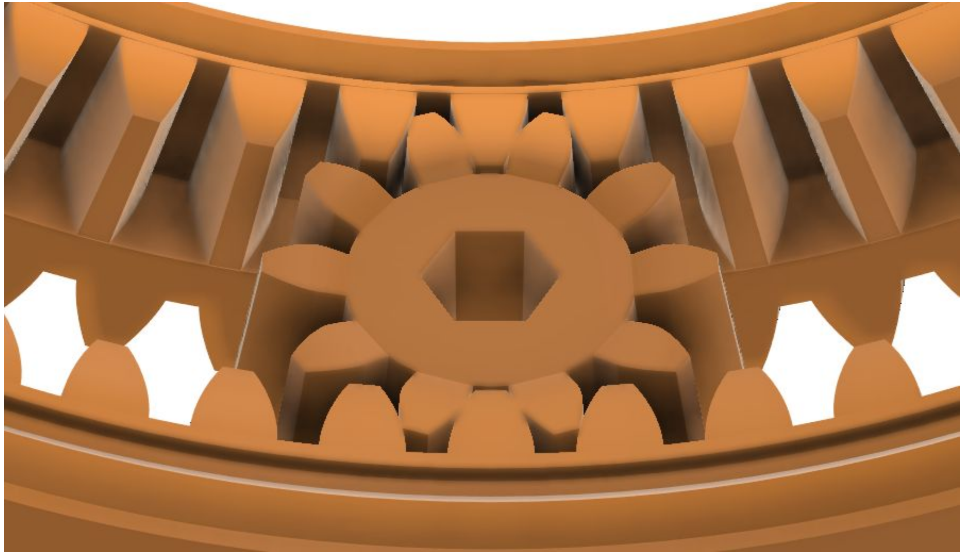
- Responds faster to inputs
- Easier to design
- More versatile
- Most common design



Mecanum

Why Not Use Swerve

- Takes a lot of time to get right
- Costs more to produce and refine
- Will run into unique challenges
- Requires advanced CAD skills
- Not all games are suited for swerve
- Not for inexperienced teams



Types of Swerve Drives

Types

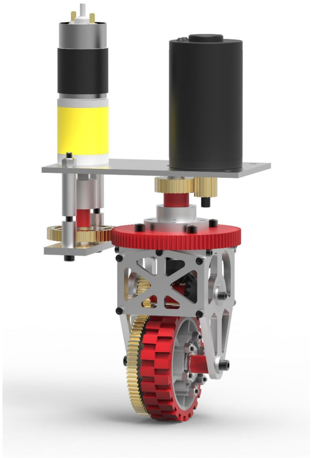
Coaxial:

- Has one motor for rotating pod and one motor for rotating the wheel

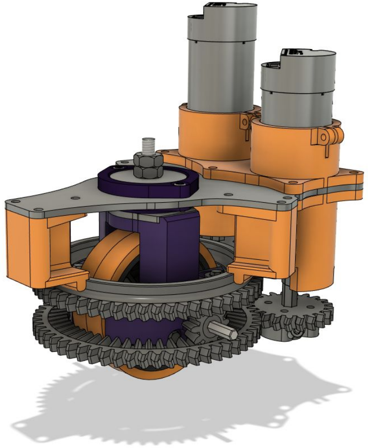
Differential:

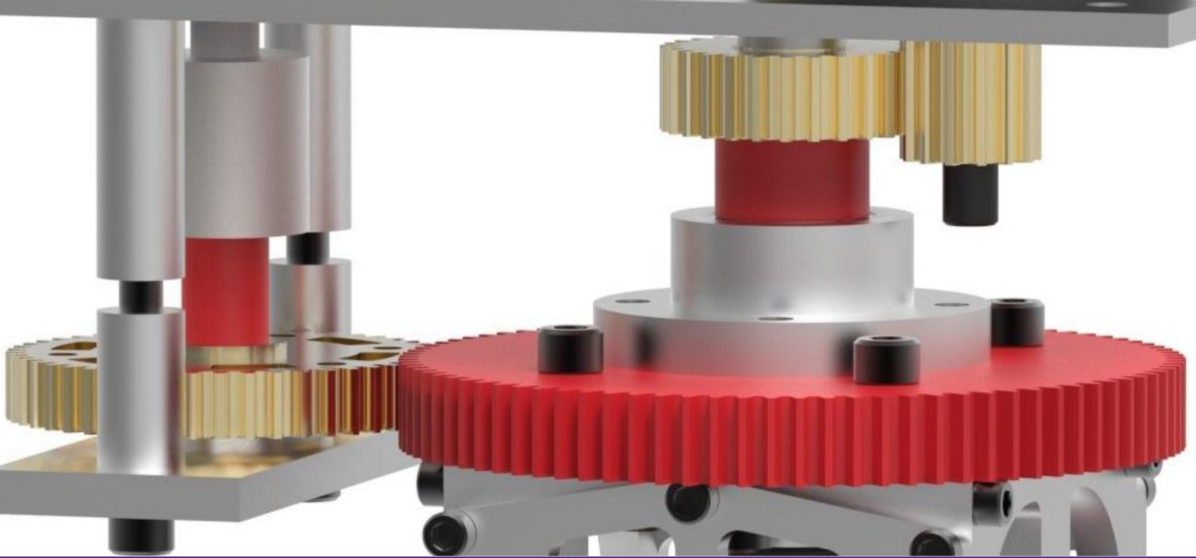
- Has two motors working together to rotate the pod and wheel

Coaxial



Differential



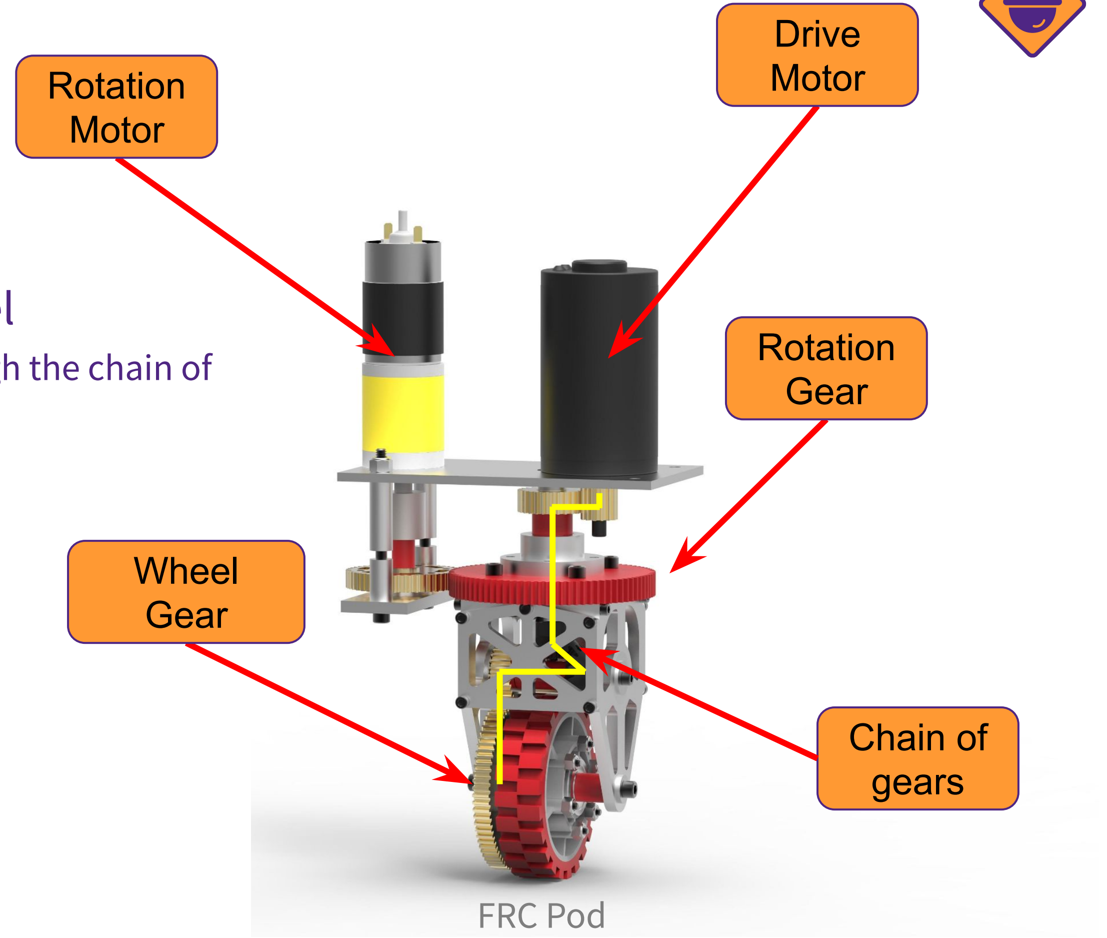


Coaxial Swerve

Coaxial Diagram

How it works:

- One DC motor rotates the wheel
  - Rotation of the motor goes through the chain of gears
- Another motor rotates the pod
  - Rotation can be done by a servo

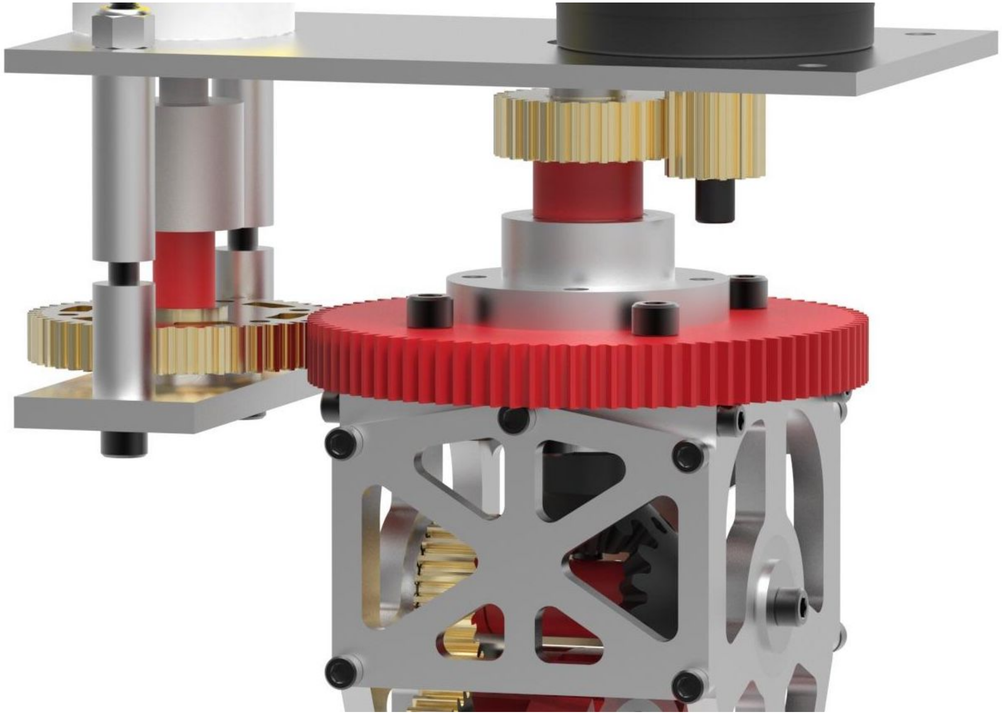


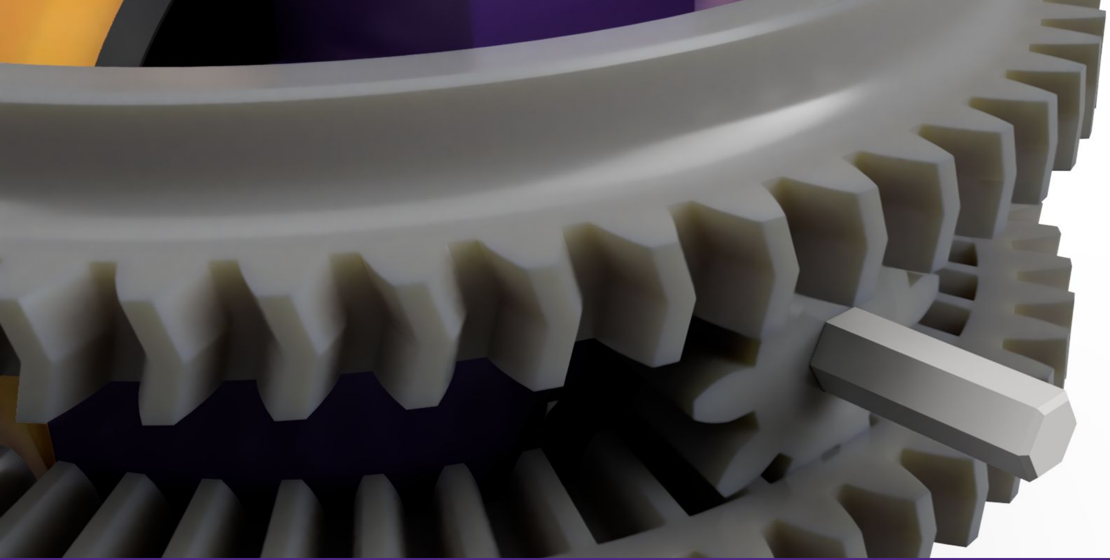
FRC Pod

Singlar Pod Operation

Attaching the Pod to the Drivetrain

- Considerations
  - Gear size
  - Wheel Diameter
  - Motor size/placement
- Custom Baseplate
  - Hole Placement
  - Material
    - Stiffness

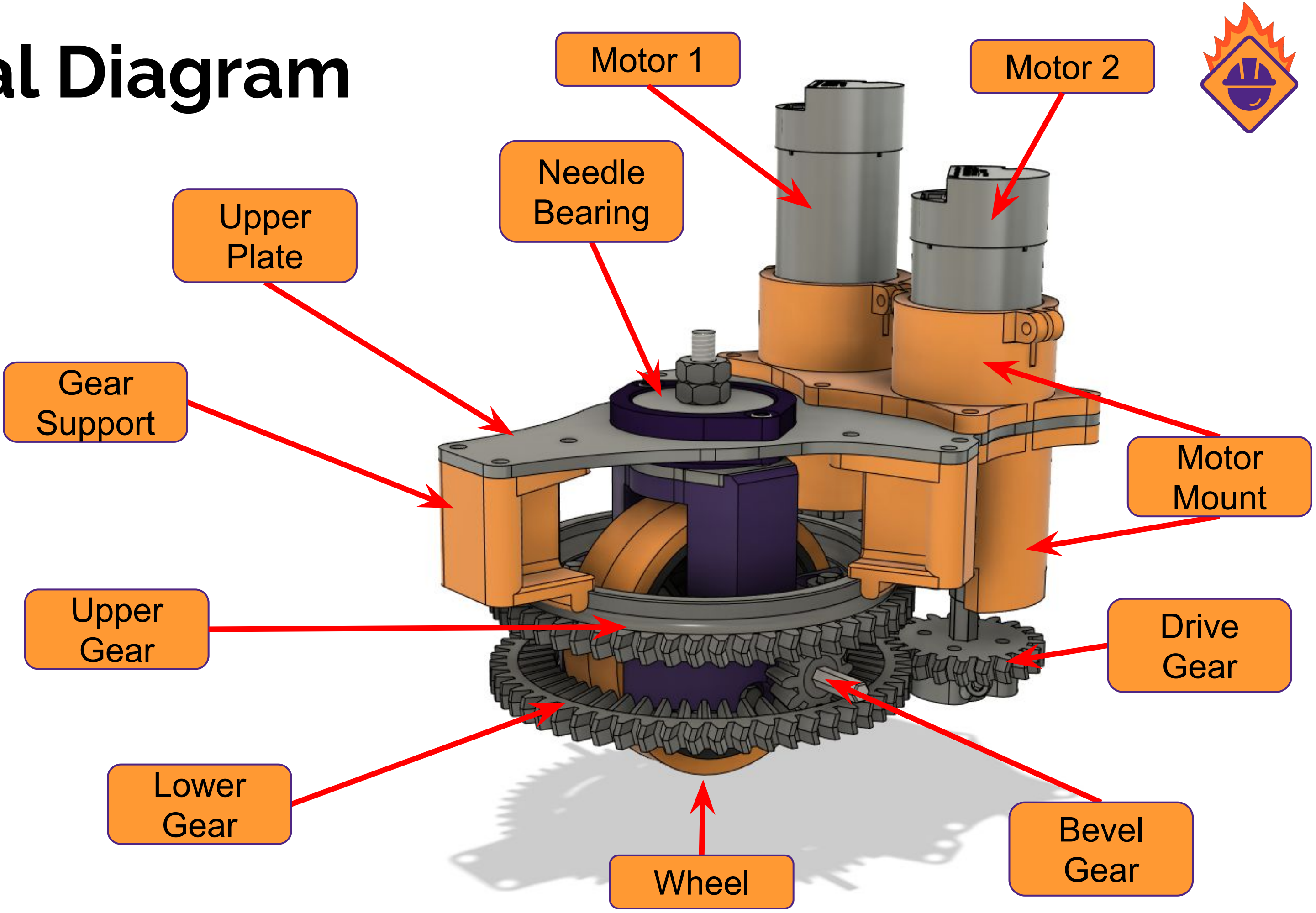




Differential Swerve

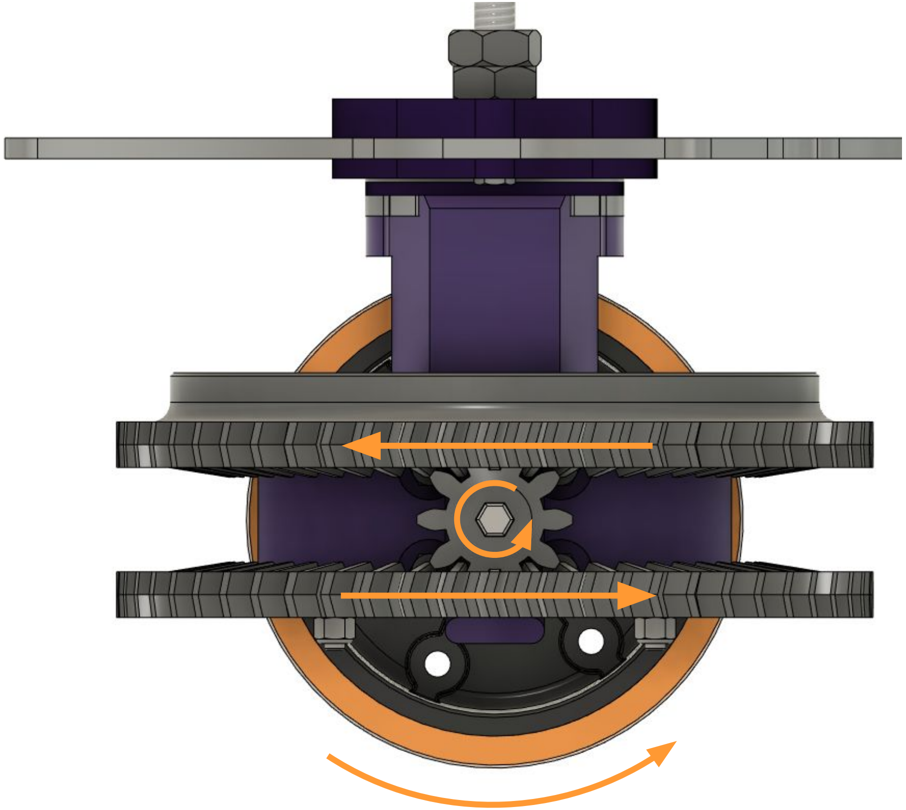
Differential Diagram

Singular Module



How it Works

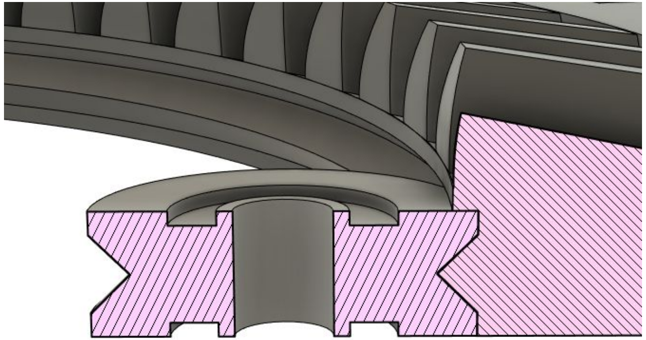
- Each pod has two sets of gears, an upper set and a lower set
- Movement is a combination of motor outputs
  - Turning - Motors move in same direction
  - Rotating Wheel - Motors move in opposite directions
- Variation between motor speeds allows rotation while moving



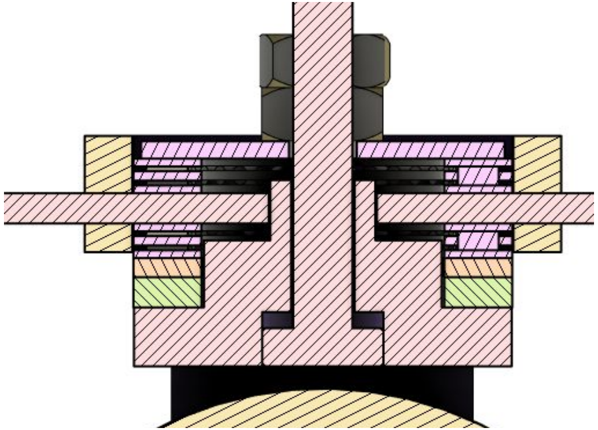
Single Module Operation

What's in our Swerve Pod

- Two bearing blocks
- Metal superstructure
- Four gears
- Bearings connect to groove in gears
- Central axle

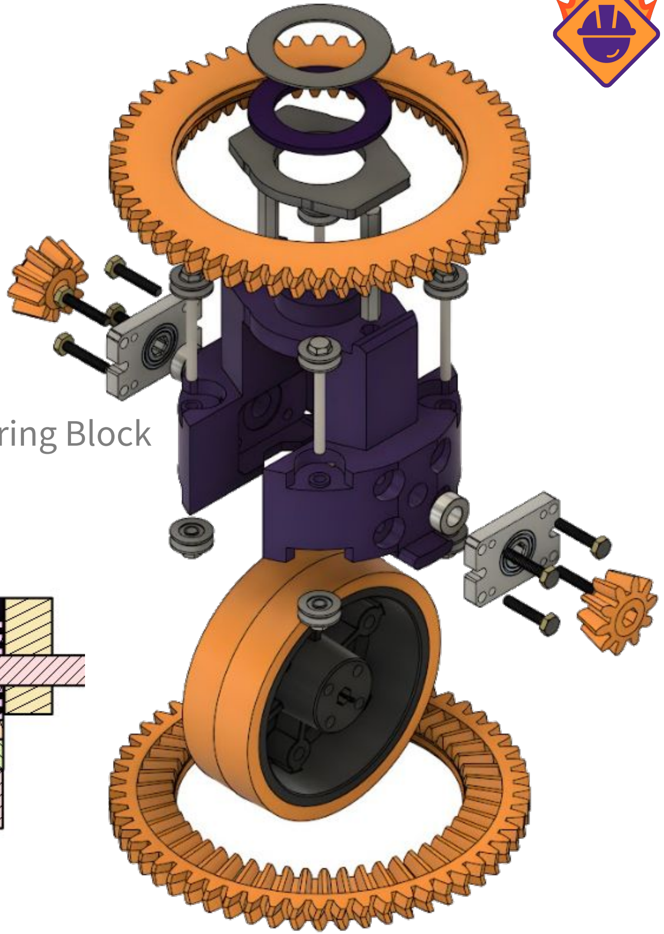


Bearing Groove



Mounting

Bearing Block



Comparison

Why use Coaxial

Benefits

- Easier to understand
- Simpler to program
- Can use servos for rotation
- Leaves encoder ports open

Considerations

- Less powerful
- Needs to use extra belts/chains/gears
- Usually takes up more vertical space

Why use Differential

Benefits

- 2x Strength
- Reliable
- Smaller vertical profile*
- Don't have to use belts/chains*

Considerations

- Build complexity
- Programming complexity
- Uses all 8 DC motors and encoder ports
- Uses more power
- Drains battery quickly

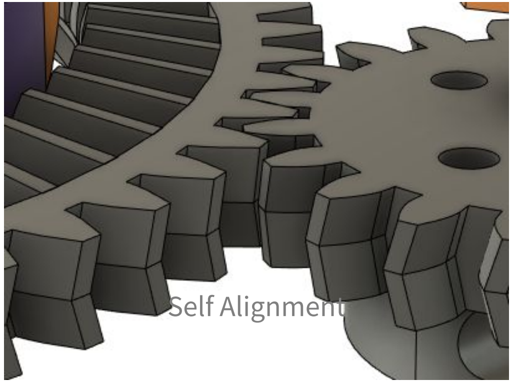
* Depends on design

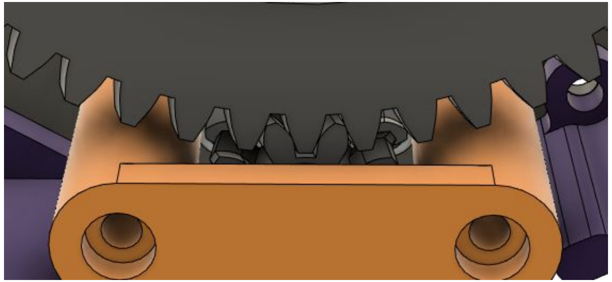
What we Learned

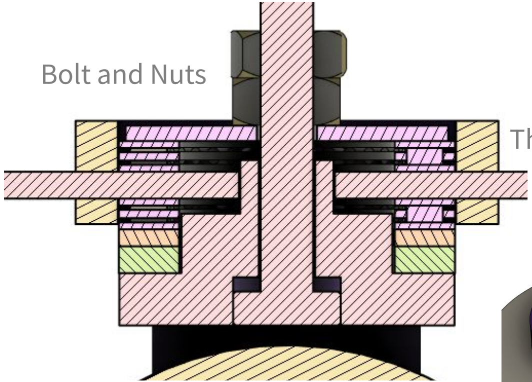
Attaching the Pod to the Drivetrain

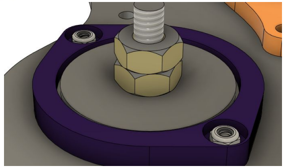
A challenge is preventing the pod from wobble or skips.

- Custom baseplate
- Use thrust bearings to take up play and wobble
    - Needle/Ball thrust bearings
- Use a central bolt
- Herringbone gears helps with self-alignment
- Rails and support gears to hold the pod




Support Gears


Bolt and Nuts
Thrust Bearing



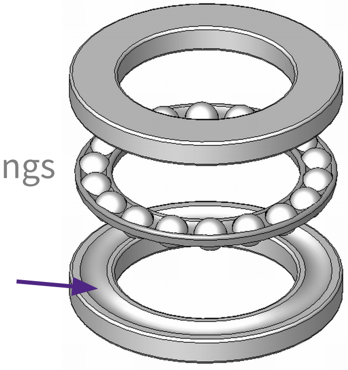
Thrust Ball Bearings vs Needle Bearings

Thrust Ball Bearing

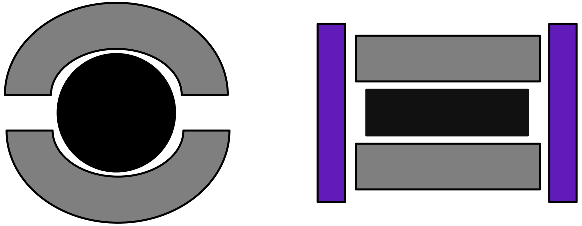
- Ball bearings between two grooved rings
- Self aligning
- More expense for larger sizes

Needle Bearing

- Needle rollers that are sandwiched between two flat washers
- Needs extra guides to align
- Cheaper for larger sizes



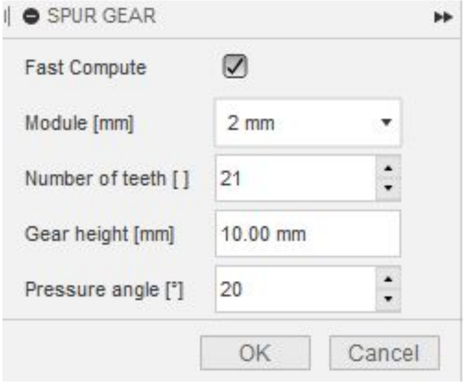
Cross Sections

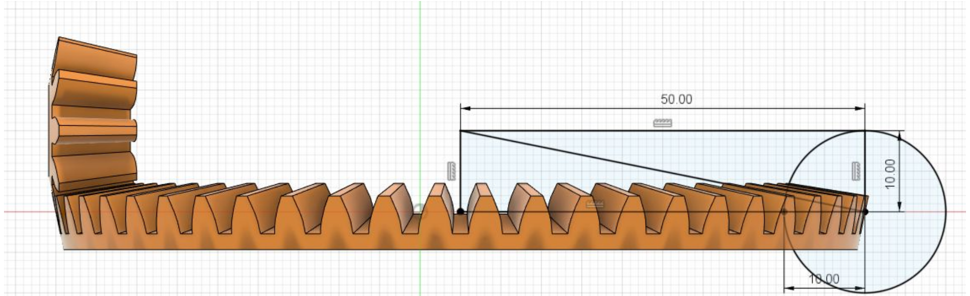


How We Designed Our Pod

We used Fusion 360 for our CAD work

- Worked out the space we had and wheel sizes
- Started from working prototypes and built up from there
- Lots of testing
- Contacted local engineers to help solve specific problems
- Document what went well and what to improve



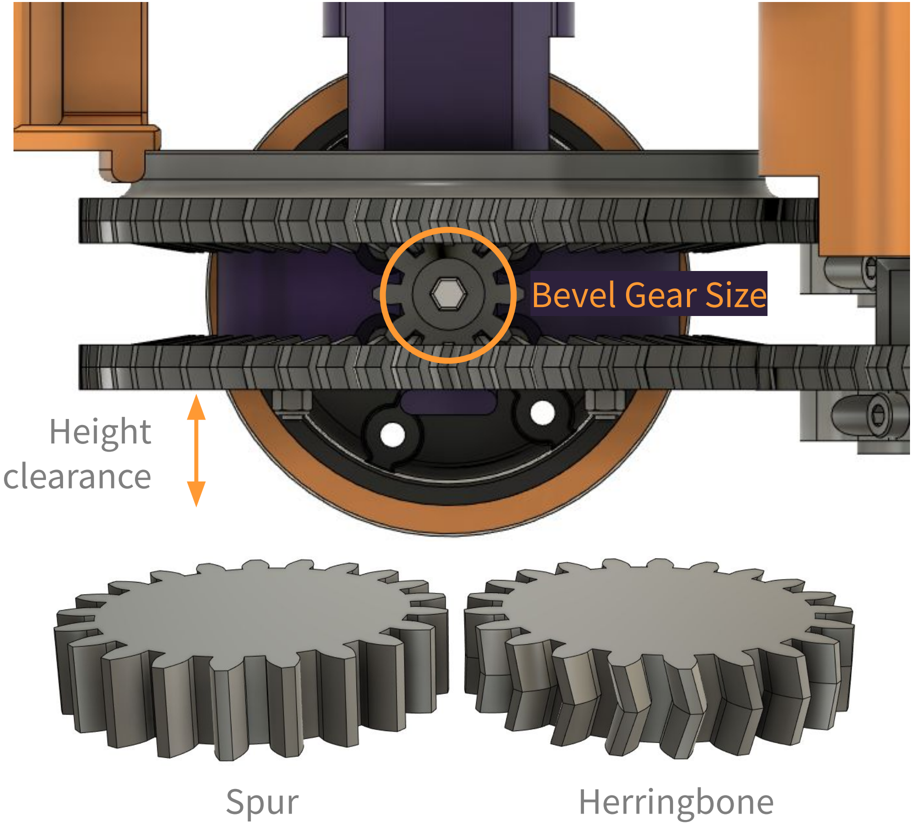


Making the bevel gears

AUTODESK
Fusion 360

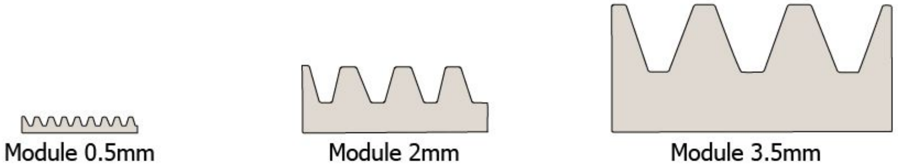
Types of Gears

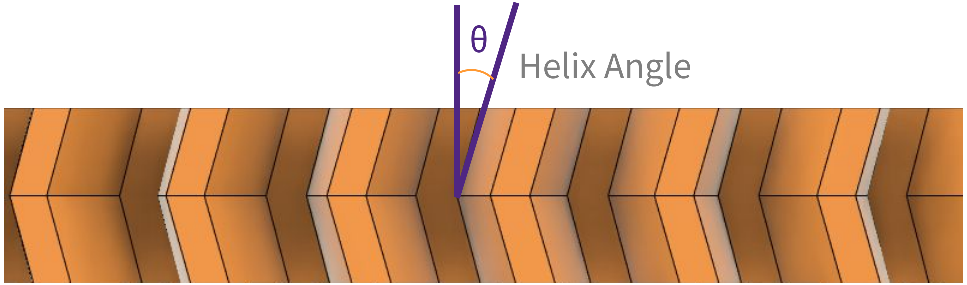
- Spur/Herringbone gears
  - Spur gears are simple to make
  - Herringbone gears help with stability
  - Ratios of Pod turning vs Wheel turning are different
  - Ratios affect robot tracking
  - Odd gear ratios (e.g. 21:10) = even teeth wear
- Bevel Gears
  - Might be difficult to CAD
  - Look for tutorials
  - Size affects ground clearance

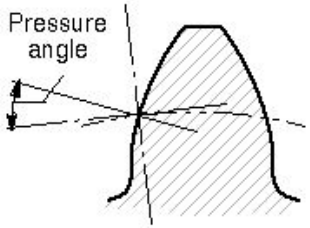


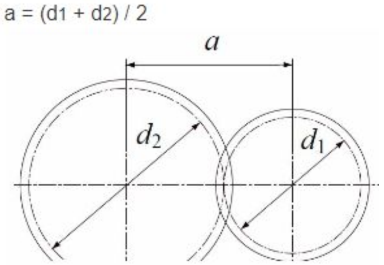
Gear Parameters

- Gear module
  - How big/small a gear is
  - We chose 2mm
- Helix angle
  - Angle of gears
  - Don't choose too high or low
  - Ours is 15°
- Pressure angle
  - angle of forces between gears
  - We did 20°
- Distance between gears
  - Sum of two gear's diameters divided by two







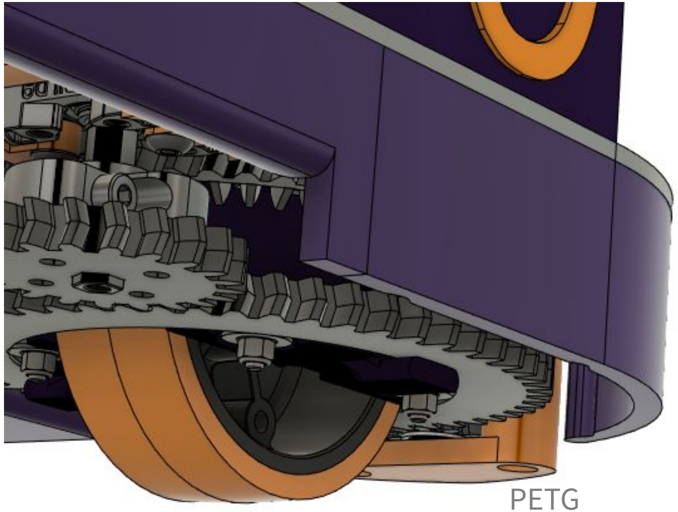


Complexity and Space

- A swerve drive is complex
  - Requires many parts that must work together seamlessly
  - Make sure enough time is allocated for design
  - Talking to engineers will greatly help
  - Understand how long it will take to repair a pod
  - Most teams create theirs during the off-season
- Keep in mind the space the pods will take up
  - Remember that gears need their own space
  - Bearings/screws also take up space
  - Cramming everything as compact as possible might make it difficult to clean or even repair

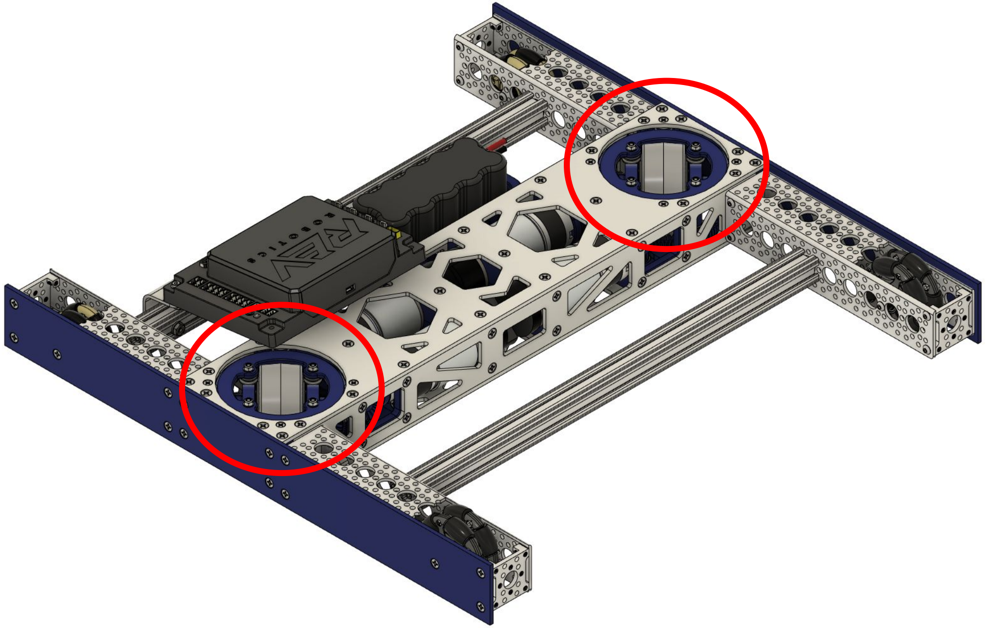
Considerations & Materials

- Gears should be protected to prevent debris from getting inside
- PLA - shell
  - Cheapest
  - Hard and brittle
  - Avoid friction fits
- PETG - supports
  - More expensive
  - Stronger than PLA
  - Softer but can handle more stress
  - Harder to print with
- Nylon - gears
  - Most expensive
  - Can handle a lot of stress
  - Low friction and high durability



Differential Drive Variations

- 2 - 3 Pod Swerve
  - Benefits
  - Considerations
- Motor Placement
  - Vertical and Horizontal
- Belts and more gears
  - Replace gears with belts
  - Gear ratio

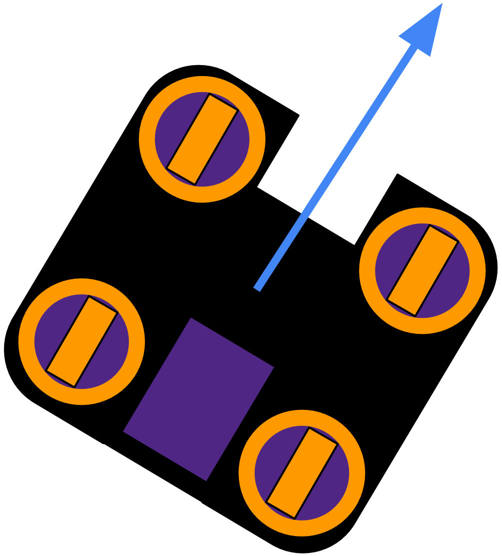


Gluten Free's Swerve Drive

Programming

Terminology

- **Delta:** The change in something
  - (i.e. delta position = change in position)
- **Vector:** A value that represents a direction and magnitude
  - (i.e. a velocity vector represents how fast something is moving and what direction it's traveling in)
  - Most arrows in this presentation represent a vector



Robot diagram with velocity vector

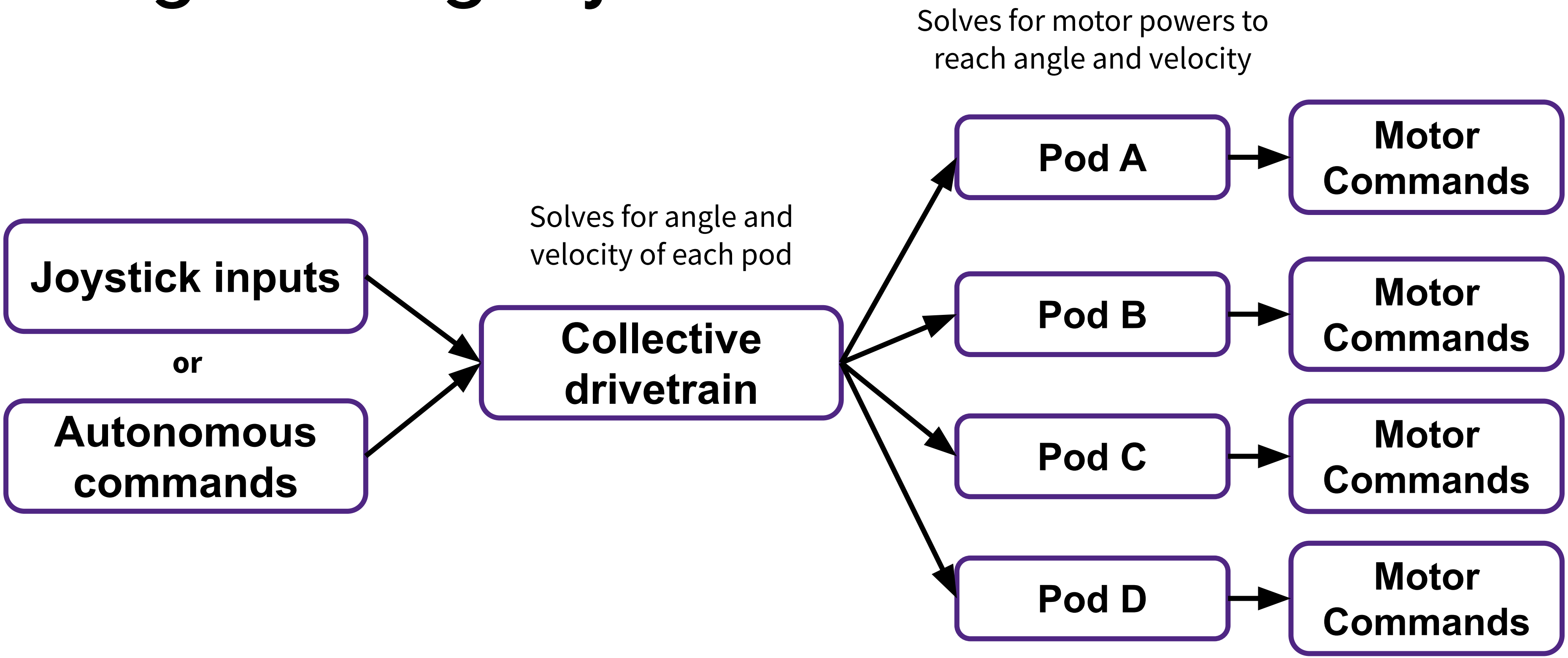
Before Programming a Swerve Drive…

You should be familiar with:

- Basic drivetrain controls
- PID Controllers
- Inverse Kinematics(IK)
- Using Java libraries for FTC

CTRL ALT FTC is an incredible resource for learning about control theory in FTC!
https://www.ctrlaltftc.com/

Programming Layout

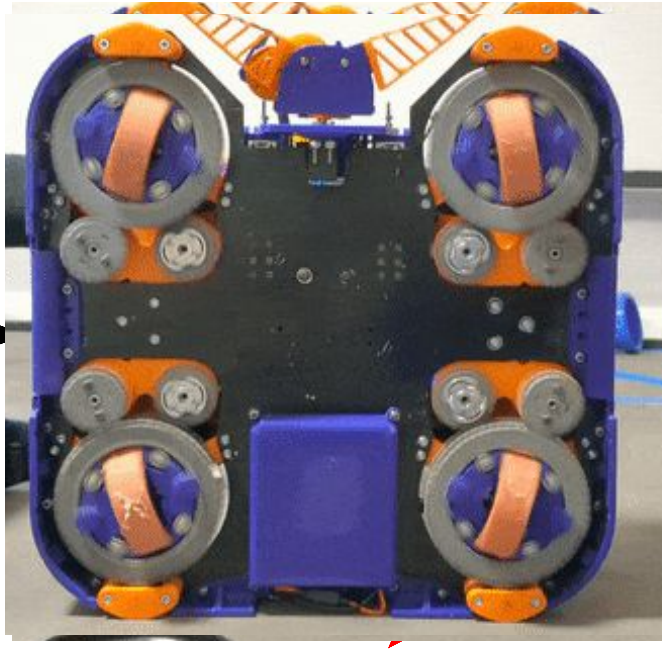


Controller Inputs

- Left Joystick Y: Forwards/backwards movements
- Left Joystick X: Horizontal Movements
- Right Joystick X: Rotation

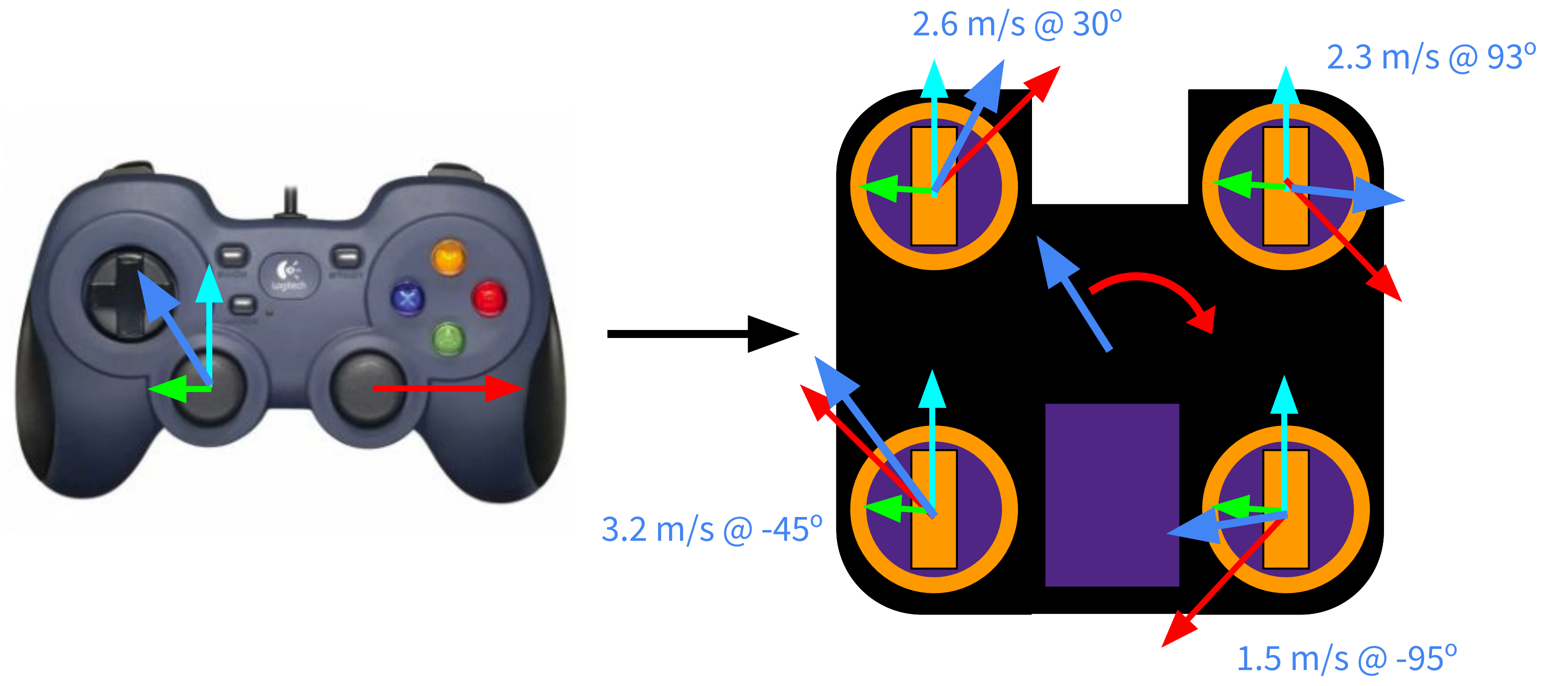
Converting to Drivetrain Inputs Pt. 1

Inverse Kinematics(IK)!!
(Math that converts joystick inputs to pod headings and powers)



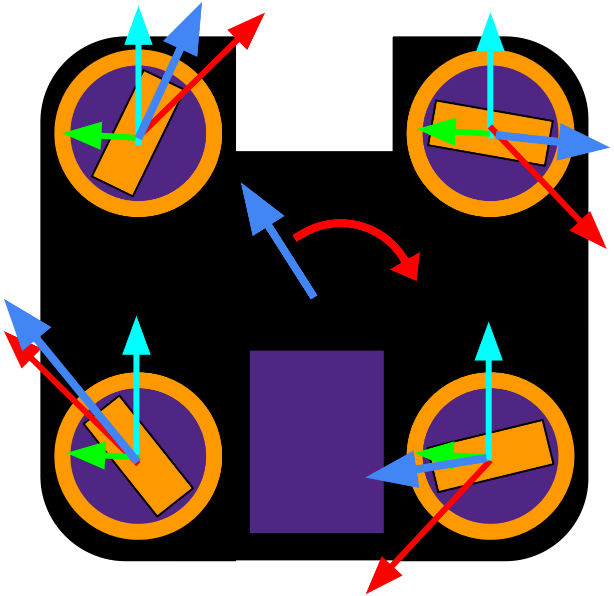
To solve IK: Imagine moving just one dimension at a time, considering what each component would need to do to travel in that dimension

Converting to Drivetrain Inputs Pt. 2

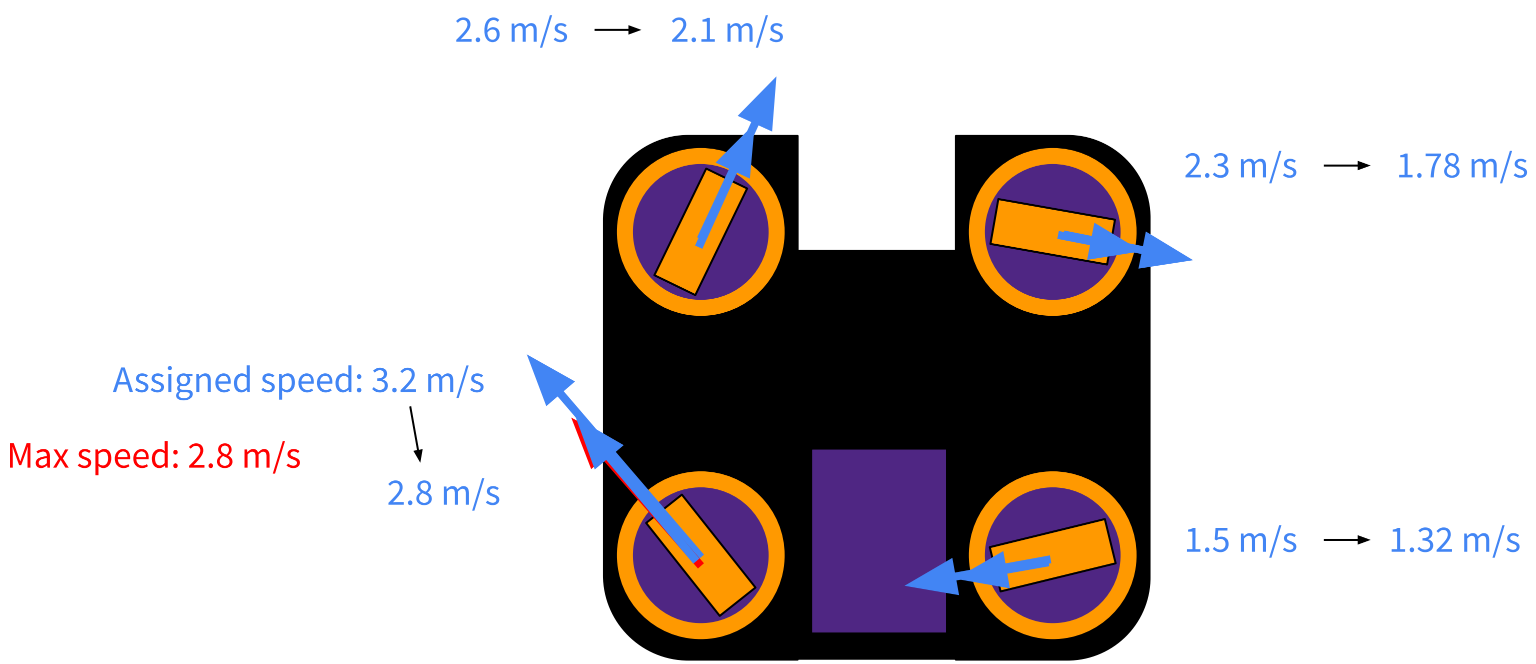


Converting to Drivetrain Inputs Pt. 3

1. Convert X and Y inputs into vectors, assign to each pod
2. Convert rotation input into vector, assign to each pod based on it's position on the robot
3. Average out assigned vectors in each pod (blue arrows)
4. Send angle and power of net vector to each pod



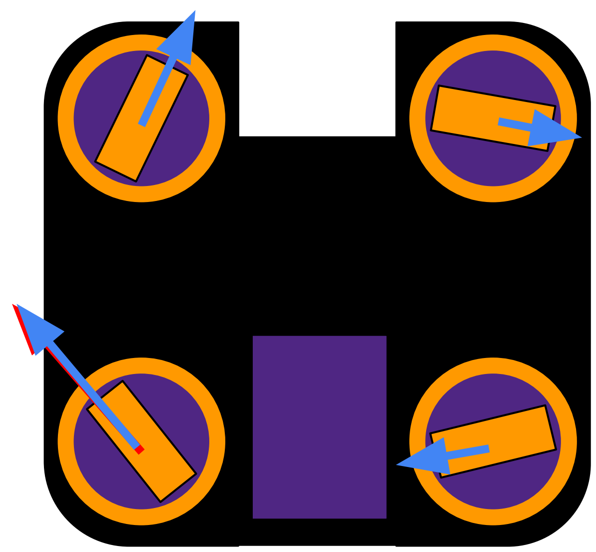
Normalizing the Wheel Speeds



Normalizing the Wheel Speeds

To Normalize wheel speeds:
Reduce all values by the same amount until they are all slower than the max speed.

Pod_n_Speed = Pod_n_speed * (Max_speed / Highest_Pod_speed)



Converting to Drivetrain Inputs Pt. 4

- **Our program uses FTC Lib:** Premade Library which does the kinematics for us

FTC Lib's Swerve Kinematics are a work in progress and might not work properly depending on implementation

How we initialize the drivetrain:

```
pods = new SwervePod[4];
pods[0] = frontLeft;
pods[1] = frontRight;
pods[2] = backLeft;
pods[3] = backRight;

frontLeftPod = new Translation2d(FRONT_LEFT_POSITION_METERS.x, FRONT_LEFT_POSITION_METERS.y);
frontRightPod = new Translation2d(FRONT_RIGHT_POSITION_METERS.x, FRONT_RIGHT_POSITION_METERS.y);
backLeftPod = new Translation2d(BACK_LEFT_POSITION_METERS.x, BACK_LEFT_POSITION_METERS.y);
backRightPod = new Translation2d(BACK_RIGHT_POSITION_METERS.x, BACK_RIGHT_POSITION_METERS.y);

swerveDriveKinematics = new SwerveDriveKinematics(frontLeftPod, frontRightPod, backLeftPod, backRightPod);
```

Creates array representing each pod

Assign locations of pods

Create FTC Lib Swerve Kinematics Class

FTC Lib is maintained by the Alpharetta Robotics Club

Converting to Drivetrain Inputs Pt. 5

How we move the drivetrain:

```
public void move(double x, double y, double r, double powerFactor) {
    speed = new ChassisSpeeds(y, -x, -r);
    moduleStates = swerveDriveKinematics.toSwerveModuleStates(speed);

    SwerveDriveKinematics.normalizeWheelSpeeds(moduleStates, attainableMaxSpeedMetersPerSecond: 1.2);

    frontLeft.driveAt(moduleStates[FRONT_LEFT], powerFactor);
    frontRight.driveAt(moduleStates[FRONT_RIGHT], powerFactor);
    backLeft.driveAt(moduleStates[BACK_LEFT], powerFactor);
    backRight.driveAt(moduleStates[BACK_RIGHT], powerFactor);
}
```

Assign desired robot speeds using joystick inputs

Solve Target angle and Velocity for each pod using FTC Lib

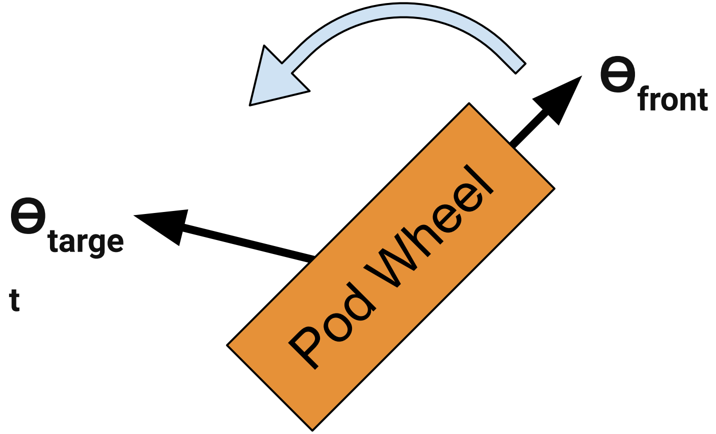
Normalize wheel speeds

Send final target angle and velocity to each pod using moduleStates array

Controlling the Pods

From the Drivetrain class, we get:

- **Target Heading(θ_{target})**
- **Target Wheel velocity**



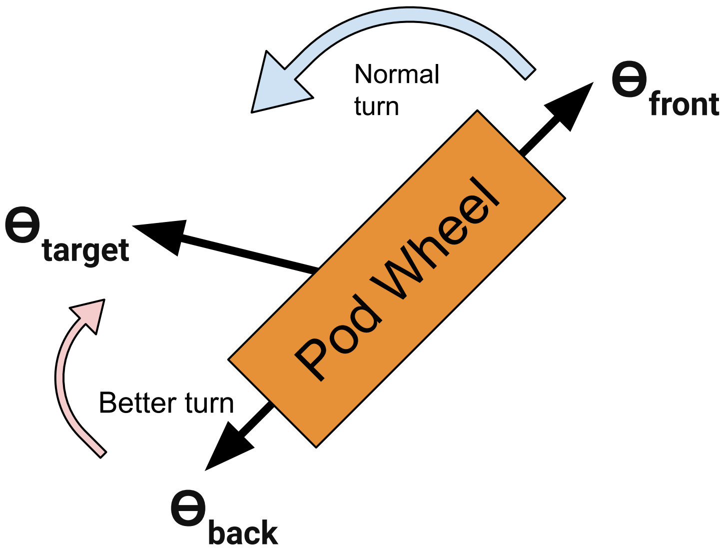
Set Pod Rotation power using PID Controller

```
rPower = PIDr.PIDControl(oppAngleFromTarget);
```

If not facing correct heading, do not move. Otherwise, move based on target wheel velocity from drivetrain class

```
if (Math.abs(angleFromTarget) > ANGLE_MARGIN_OF_ERROR) {
    velocity = 0;
}
```

Pod Optimization pt. 1



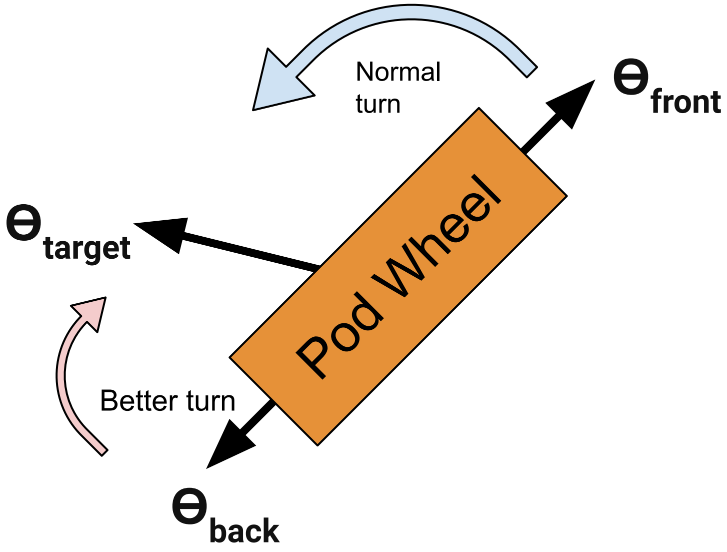
Pod Optimization pt. 2

If front angle is further from target than back angle, rotate based on backangle. Otherwise, rotate based on front angle.

```
if (Math.abs(angleFromTarget) > Math.abs(oppAngleFromTarget)) {
    rPower = PIDr.PIDControl(oppAngleFromTarget);
}
else {
    rPower = PIDr.PIDControl(angleFromTarget);
}
```

If back angle is close enough to target, move wheel backwards. Otherwise, if neither is close enough, don't move wheel.

```
if (Math.abs(oppAngleFromTarget) < ANGLE_MARGIN_OF_ERROR) {
    velocity = -velocity;
}
else if (Math.abs(angleFromTarget) > ANGLE_MARGIN_OF_ERROR) {
    velocity = 0;
}
```



Final Pod Program

```java
//moveTo: Given target angle and velocity, moves the wheel to that angle at that velocity
public void moveTo(double velocity, double targetAngle, double power) {

    currentAngle = getAngle();
    oppAngle = angleWrap(currentAngle + 180); //gives us the opposite side since the wheel

    angleFromTarget = angleWrap(targetAngle - currentAngle);
    oppAngleFromTarget = angleWrap(targetAngle - oppAngle);

    if (Math.abs(angleFromTarget) > Math.abs(oppAngleFromTarget)) {
        rPower = PIDr.PIDControl(oppAngleFromTarget);
    }
    else {
        rPower = PIDr.PIDControl(angleFromTarget);
    }

    if (Math.abs(oppAngleFromTarget) < ANGLE_MARGIN_OF_ERROR) {
        velocity = -velocity;
    }
    else if (Math.abs(angleFromTarget) > ANGLE_MARGIN_OF_ERROR) {
        velocity = 0;
    }

    upperMotor.setVelocity(((-velocity * power) + rPower) * MAX_RPS_TICKS);
    lowerMotor.setVelocity(((velocity * power) + rPower) * MAX_RPS_TICKS);
}
```

Find back angle

Find errors from target angle

Figure out which way to turn

If back side is close to target, run backwards. Don't move if neither side is close

Assign motor powers (on diff. drive)

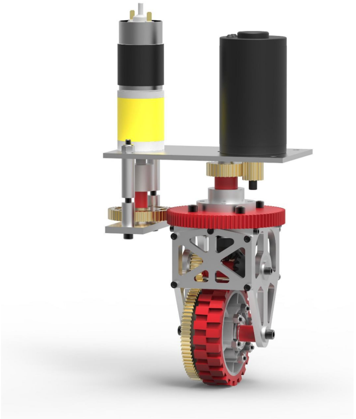
Telling the Motors How to Move

For Coaxial Swerve:

- Assign the **target angle** to the **turning motor**
- Assign the **target velocity** to the **drive motor**

For Differential Swerve:

- Use Inverse Kinematics to control each motor!



Controlling Differential Pods pt. 1

In the same manner as before, imagine each dimension of movement by itself and figure out how the motor moves in those dimensions.

Movement: both motors move in opposite directions

Rotation: both motors move in the same direction

Controlling Differential Pods pt. 2

```
upperMotor.setVelocity(((-velocity * power) + rPower) * MAX_RPS_TICKS);
lowerMotor.setVelocity(((velocity * power) + rPower) * MAX_RPS_TICKS);
```

velocity = wheel movement value

rPower = pod rotation value

Reading the Pod Angle

For Coaxial Swerve:

- Many servos have an internal encoder for use as a rotation motor (e.g. Axon MAX)
- You can also use a separate encoder (e.g. REV through bore encoder)

For Differential Swerve:

- **Using math to derive angle from motor encoders is very effective but not absolute**
- You can also use an external encoder, but with 4 pods you will run out of encoder ports and will have to use an analog encoder (e.g. Andymark MA3)

Reading the Pod Angle with Diffy Swerve

A diffy pod's angle is equal to the mean angle between both motors:

Pod_Angle = (topMotorPosition + bottomMotorPosition) / 2

Rotation has both motors turn equally **in the same direction**

Translation has both motors turn equally **in opposite directions**

Reading the Pod Angle with Diffy Swerve

```
public double getAngle() {
    currentAngleTicks = (upperPos + lowerPos) / 2.0;
    currentAngle = currentAngleTicks / TICKS_PER_DEGREE_BIG_GEAR;

    currentAngle = angleWrap(currentAngle);

    return currentAngle;
}
```

Find the mean encoder value

Convert from encoder ticks to degrees

Anglewrap to keep the value between -180 and 180

Reading Wheel Position with Diffy Swerve

Solving for wheel position is almost identical to solving for pod angle, just negated

Pod_Wheel_position = (topMotorPosition - bottomMotorPosition) / 2

Reading Wheel Position with Diffy Swerve

Negate the average equation to get wheel position instead

```
public double getWheelPositionINCH() {
    currentPositionTicks = (upperPos - lowerPos) / 2.0;
    currentPosition = currentPositionTicks / TICKS_PER_INCH;

    return currentPosition;
}
```

Convert from ticks to inches traveled

Localization

Localization: Finding the robot's position and heading on the field

We always recommend using deadwheel odometry, but for a 4 pod diffy swerve there isn't enough encoders for that to be an option.

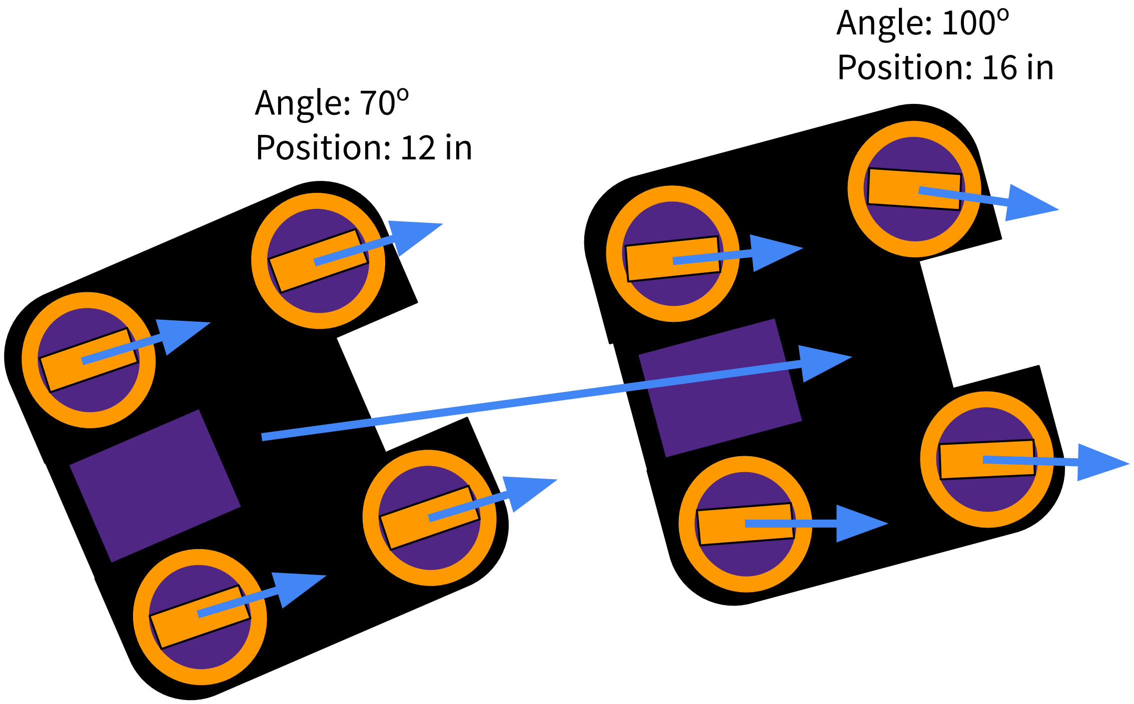
With 4 pod swerve:

- For robot heading, use the rev hub's IMU
- For field position, use swerve odometry

Our deadwheel odometry pod

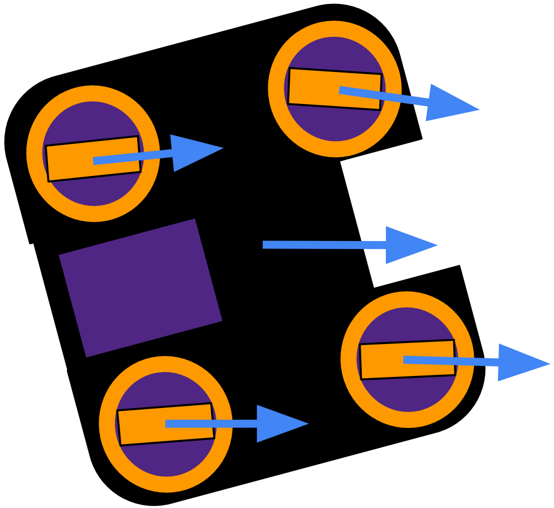
Swerve Odometry Pt. 1

Big thanks to 16379 KookyBotz for creating this process and helping us adapt it for our drivetrain!



Final angle: 100°
Delta wheel position: 4 in

Swerve Odometry Pt. 2



Average out all changes in pod positions to get the **net change in position**

Then, add that **net change in position** to the **last known position** to get your robot's **current position**

Swerve Odometry

```java
@Override
public void update() {
    prevTime = timer.seconds();
    pastPoseEstimate = poseEstimate;

    Vector2d accumulator = new Vector2d();
    double head = Math.toRadians(driveTrain.getHeading());
    for (SwervePod pod : ((SwerveDrivetrain) driveTrain).getSwervePods()) {
        accumulator = accumulator.plus(pod.getDeltaPosition());
    }
    accumulator = accumulator.div(4).rotated(head);

    poseEstimate = new Pose2d(poseEstimate.vec().plus(accumulator), head);
}
```

Find robot heading with IMU

Add the change in positions of each pod together

Solve for average delta position

Add net change to last known position

Swerve Odometry

Road runner variable that represents a vector

Find the change in wheel position

```
public Vector2d getDeltaPosition() {
    deltaPosition = getWheelPositionINCH() - lastPosition;
    lastPosition = getWheelPositionINCH();
    return Vector2d.polar(deltaPosition, Math.toRadians(getAngle()));
}
```

Return change with current angle

Autonomous Movement Control

Autonomous movement can be very similar to normal holonomic movement, even with systems like **pure pursuit** or **Road Runner**.

Here are some things to keep in mind while designing auto programs:

- Swerve pods should be given time to turn towards a target angle before telling them to move to reduce error.
- Swerve Odometry, while more stable, will still accumulate error over time so using many sensors like distance sensors to re-localize is a good idea.
- If a gear jumps, all of your sensor readouts will be wrong.

Additional Resources

PROJECT
ROBOTICA

a Wiki by SponCon FTC 14779

Projectrobotica.wiki

sponconftc.com

Questions? Email us at:
team@spontaneousconstruction.com

Our Swerve Drivetrain

Our Swerve Pod

Our youtube channel

Make sure to come see our robot at Worlds!

Additional Resources Continued

Gluten Free's Swerve Drive

CTRL ALT FTC

FTC Lib Swerve Kinematics

Field centric swerve drift issue & solution

FTC Reddit

FTC Discord server