VECTOR ALGEBRA AND SWERVE

ThunderChickens – FRC Team 217

Swerve drivebase example

WHAT IS A VECTOR?

- Has a direction and magnitude
  - Velocity
  - Position
  - Force
- Has a head and a tail
  - The tail is the starting point of the vector
  - The head is the pointed end of the vector arrow
- Can be broken into components
- Can be added and subtracted
- Has three different methods of "multiplication"
  - By a scalar
  - By another vector:
    - Dot product (gives us a scalar)
    - Cross product (also called the vector product because it gives us a vector)

WHAT IS A SCALAR?

- Has a magnitude but no direction
  - Speed
  - Volume
  - Mass
- Described by real numbers*
  - *Some branches of mathematics also use complex (imaginary) numbers

BREAKING DOWN 2-D VECTORS

- Given a 2-D vector $\vec{v}$ with magnitude $||\vec{v}||$ and direction θ:
  - $\vec{v} = \langle\, ||\vec{v}|| \cos(\theta)\, , ||\vec{v}|| \sin(\theta)\, \rangle$
    $= ||\vec{v}|| \cos(\theta)\, \hat{\imath} + ||\vec{v}|| \sin(\theta)\, \hat{\jmath}$
    - $||\vec{v}|| \cos(\theta)$ is the $\hat{\imath}$ component, $||\vec{v}|| \sin(\theta)$ is the $\hat{\jmath}$ component
    - $\hat{\imath}$ is a unit vector pointing along x
    - $\hat{\jmath}$ is a unit vector pointing along y
    - A unit vector has a magnitude of 1
- You can also go backwards. Given $\vec{v} = \langle x, y \rangle$:
  - $||\vec{v}|| = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(\frac{y}{x})$

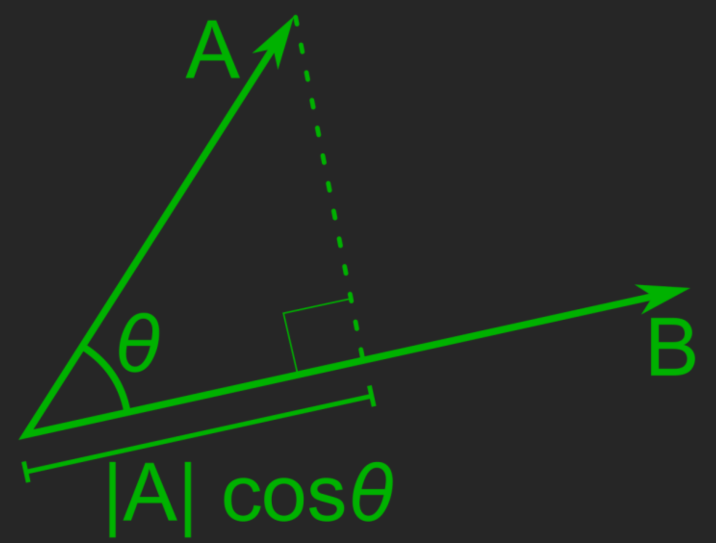
SCALING VECTORS

- Given $\vec{v} = \langle x, y \rangle$ and a scalar c:
  - $c\vec{v} = \langle cx, cy \rangle$
  - $||\vec{v}||$ is scaled by c
  - θ is unchanged

ADDING AND SUBTRACTING VECTORS

- Given $\vec{u} = \langle u_1, u_2 \rangle$ and $\vec{v} = \langle v_1, v_2 \rangle$:
  - $\vec{u} + \vec{v} = \langle u_1 + v_1, u_2 + v_2 \rangle$
    - NOTE: Adding vectors does NOT add the vector magnitudes together; it adds the component magnitudes together
  - $\vec{u} - \vec{v} = \langle u_1 - v_1, u_2 - v_2 \rangle$
- Geometrically:
  - Adding vectors attaches the tail of the second vector to the head of the first
  - Subtracting vectors:
    - Flips the second vector's head and tail (scales it by -1)
    - Attaches the tail of the flipped second vector to the head of the first

DOT PRODUCT



- Given $\vec{u} = \langle u_1, u_2 \rangle$ and $\vec{v} = \langle v_1, v_2 \rangle$:
  - $\vec{u} \cdot \vec{v} = u_1 v_1 + u_2 v_2$
- What does this mean?
- Given $\|\vec{u}\|$, $\|\vec{v}\|$, and the angle θ between $\vec{u}$ and $\vec{v}$:
  - $\vec{u} \cdot \vec{v} = \|\vec{u}\| \|\vec{v}\| \cos(\theta)$
- Dot product $\vec{u} \cdot \vec{v}$ gives us the magnitude of $\vec{u}$ in the direction of $\vec{v}$, and vice versa

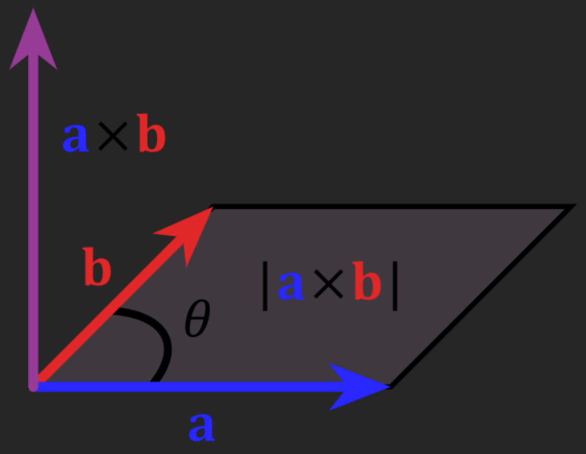
CROSS PRODUCT

- Given $\vec{u} = \langle u_1, u_2, u_3 \rangle$ and $\vec{v} = \langle v_1, v_2, v_3 \rangle$:
  - $\vec{u} \times \vec{v} = \det\left(\begin{bmatrix} \hat{\imath} & u_1 & v_1 \\ \hat{\jmath} & u_2 & v_2 \\ \hat{k} & u_3 & v_3 \end{bmatrix}\right)$

$$= \hat{\imath}(u_2 v_3 - u_3 v_2) - \hat{\jmath}(u_1 v_3 - u_3 v_1) + \hat{k}(u_1 v_2 - u_2 v_1)$$

$$= \begin{bmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{bmatrix}$$

- What does this mean?
- Given $\|\vec{u}\|$, $\|\vec{v}\|$, and the angle θ between $\vec{u}$ and $\vec{v}$:
  - $\|\vec{u} \times \vec{v}\| = \|\vec{u}\|\|\vec{v}\||\sin(\theta)| = Area\ of\ a\ parallelogram$
- Cross product $\vec{u} \times \vec{v}$ gives us a vector perpendicular to both $\vec{u}$ and $\vec{v}$ with magnitude equivalent to the area of the parallelogram formed by $\vec{u}$ and $\vec{v}$



CROSS PRODUCT – PROOF

- Given $\vec{u} = \langle u_1, u_2, u_3 \rangle$ and $\vec{v} = \langle v_1, v_2, v_3 \rangle$:
  - $\vec{u} \times \vec{v} = \begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix}$
  - $\begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = u_1(u_2v_3 - u_3v_2) + u_2(u_3v_1 - u_1v_3) + u_3(u_1v_2 - u_2v_1)$
    $= u_1u_2v_3 - u_1u_3v_2 + u_2u_3v_1 - u_1u_2v_3 + u_1u_3v_2 - u_2u_3v_1 = 0$
  - $\begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = v_1(u_2v_3 - u_3v_2) + v_2(u_3v_1 - u_1v_3) + v_3(u_1v_2 - u_2v_1)$
    $= u_2v_1v_3 - u_3v_1v_2 + u_3v_1v_2 - u_1v_2v_3 + u_1v_2v_3 - u_2v_1v_3 = 0$

PROPERTIES OF VECTORS

- Commutative property: $\vec{u} + \vec{v} = \vec{v} + \vec{u}$
- Associative property: $\vec{u} + (\vec{v} + \vec{w}) = (\vec{u} + \vec{v}) + \vec{w}$
- Identity property: $1\vec{v} = \vec{v}$
- Zero property: $0\vec{v} = \vec{0}$
  - $\vec{0} = \langle 0,0 \rangle$
- Distributive property:
  - $c(\vec{u} + \vec{v}) = c\vec{u} + c\vec{v}$
- $(c + d)\vec{v} = c\vec{v} + d\vec{v}$
- Two non-zero vectors $\vec{u}$ and $\vec{v}$ are parallel if $\vec{u} = c\vec{v}$ for some scalar c
- Cancellation property: If $\vec{u} + \vec{v} = \vec{u} + \vec{w}$, then $\vec{v} = \vec{w}$

PROPERTIES OF THE DOT PRODUCT

- Commutative property: $\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{u}$
- Not associative
  - The dot product returns a scalar, so $\vec{u} \cdot \vec{v} \cdot \vec{w}$ is invalid
- Scalar multiplication property: $c(\vec{u} \cdot \vec{v}) = c\vec{u} \cdot \vec{v} = \vec{u} \cdot c\vec{v}$
- Zero property: $\vec{v} \cdot \vec{0} = 0$
- Distributive property: $\vec{u} \cdot (\vec{v} + \vec{w}) = \vec{u} \cdot \vec{v} + \vec{u} \cdot \vec{w}$
- Bilinear property: $\vec{u} \cdot (c\vec{v} + \vec{w}) = c(\vec{u} \cdot \vec{v}) + \vec{u} \cdot \vec{w}$
- Two non-zero vectors $\vec{u}$ and $\vec{v}$ are orthogonal if $\vec{u} \cdot \vec{v} = 0$
- No cancellation
  - $\langle 6,3 \rangle \cdot \langle 3,2 \rangle = 24 = \langle 6,3 \rangle \cdot \langle 4,0 \rangle$, but $\langle 3,2 \rangle \neq \langle 4,0 \rangle$
  - Using the distributive property, $\vec{u} \cdot (\vec{v} - \vec{w}) = 0$, so $\langle 3,2 \rangle - \langle 4,0 \rangle = \langle -1,2 \rangle$ is orthogonal to $\langle 6,3 \rangle$

PROPERTIES OF THE CROSS PRODUCT

- Anti-commutative property: $\vec{u} \times \vec{v} = -(\vec{v} \times \vec{u})$
- Not associative
  - Satisfies the Jacobi Identity: $\vec{u} \times (\vec{v} \times \vec{w}) - \vec{v} \times (\vec{u} \times \vec{w}) + \vec{w} \times (\vec{u} \times \vec{v}) = 0$
- Scalar multiplication property: $c(\vec{u} \times \vec{v}) = c\vec{u} \times \vec{v} = \vec{u} \times c\vec{v}$
- Zero property: $\vec{v} \times \vec{0} = \vec{0}$
- Distributive property: $\vec{u} \times (\vec{v} + \vec{w}) = \vec{u} \times \vec{v} + \vec{u} \times \vec{w}$
- Bilinear property: $\vec{u} \times (c\vec{v} + \vec{w}) = c(\vec{u} \times \vec{v}) + \vec{u} \times \vec{w}$
- Two non-zero vectors $\vec{u}$ and $\vec{v}$ are parallel if $\vec{u} \times \vec{v} = 0$
- No cancellation
  - $\vec{u} \times \vec{v} = \vec{u} \times \vec{w}$ does not imply $\vec{v} = \vec{w}$
  - Using the distributive property, $\vec{u} \times (\vec{v} - \vec{w}) = 0$

VECTORS IN SWERVE: OVERVIEW

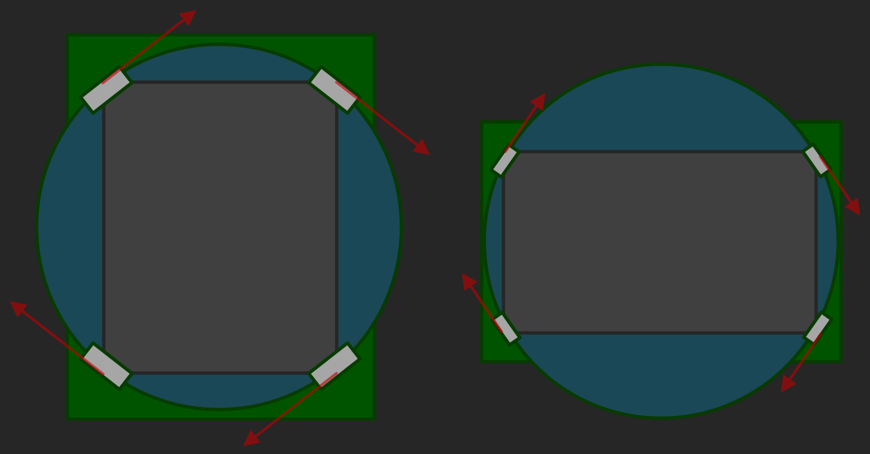
- The four wheels in swerve can each be represented by vectors
  - Magnitude is speed
  - Direction is wheel angle
  - Swerve wheels operate in 2-D space, so we can break down the vectors into $\langle x, y \rangle$
- A matrix is a collection of vectors, so we can represent the wheel vectors in a 4x2 matrix (4 rows, 2 columns => 4 wheels, $\langle x, y \rangle$)
  - In code, a matrix is a 2-D array `wheels[rows][columns]`
  - We will start at the front left wheel and go around clockwise when numbering our rows

VECTORS IN SWERVE: SPEED AND STRAFE

- Speed (forwards and backwards) and strafe (left and right) can be represented as a vector: $\langle \text{strafe}, \text{speed} \rangle$
  - strafe is left/right = x, speed is fwd/back = y
- All wheels have the same speed/strafe vector
  - All wheels point in the same direction and travel the same speed when strafing

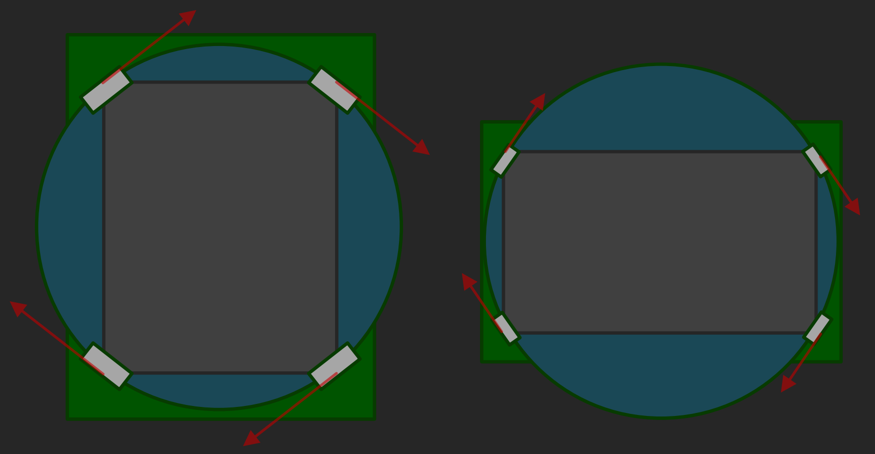
```java
double[][] strafeMatrix = new double[4][2]; // 4 vectors, each containing 2 elements (x and y)
for (double[] strafeVector : strafeMatrix) {
    strafeVector[0] = strafe; // strafe is x
    strafeVector[1] = speed; // speed is y
}
return strafeMatrix;
```

VECTORS IN SWERVE: TURNING



- When turning in swerve, the wheels lie tangent to a circle
- Vectors point clockwise when turning right, counter-clockwise when turning left
- The vector is related to the distance between the front and back wheels, and between the left and right wheels:
  - $\vec{v} = \langle length, width \rangle$
  - As the bot gets longer, the wheels point farther from forward
  - As the bot gets wider, the wheels point closer to forward
- Clockwise turn: $\begin{bmatrix} length & width \\ length & -width \\ -length & -width \\ -length & width \end{bmatrix} * 1/\sqrt{length^2 + width^2}$
  - Base turning vectors must be unit vectors, scalar is $1/\sqrt{length^2 + width^2}$

VECTORS IN SWERVE: TURNING



- The turning matrix is scaled by the magnitude of the input turn value
  - Since the base turn vectors are unit vectors, this makes each wheel have a turn speed equal to the input turn value
  - A negative turn value flips the direction of all vectors, resulting in a counter-clockwise turn

```
turnMatrix[0][0] = kBotLength;
turnMatrix[0][1] = kBotWidth;

turnMatrix[1][0] = kBotLength;
turnMatrix[1][1] = -kBotWidth;

turnMatrix[2][0] = -kBotLength;
turnMatrix[2][1] = -kBotWidth;

turnMatrix[3][0] = -kBotLength;
turnMatrix[3][1] = kBotWidth;
```

```
double botMagnitude = Num.distance(kBotLength, kBotWidth);
for (double[] turnVector : turnMatrix) {
    // convert turnVector into unit vector
    turnVector[0] /= botMagnitude;
    turnVector[1] /= botMagnitude;
    // multiply by scalar (turn)
    turnVector[0] *= turn;
    turnVector[1] *= turn;
}

return turnMatrix;
```

VECTORS IN SWERVE: COMBINING THE VECTORS

- The final movement vector for the wheels is the sum of the two vectors (speed and strafe, and turn)
  - For the two vectors, larger magnitude = more influence on the final vector
- The speed of a wheel cannot be greater than 1.0, so scale down all vectors by the largest magnitude or 1.0, whichever is greater
  - If the largest magnitude is less than 1, we don’t want to scale it down; doing so will result in constant max speed

```java
// NOTE: matrix starts with the leftMaster and goes clockwise
double[][] wheelMatrix = getStrafeMatrix(speed, strafe);

// need to keep track of the max magnitude of the wheel vectors
// since none of them can be greater than 1
double maxWheelMagnitude = 1;

// variables in this block are not accessible elsewhere
{
    // get turn matrix
    double[][] turnMatrix = getTurnMatrix(turn);
```

```java
    for (int i = 0; i < wheelMatrix.length; i++) {
        // add turnMatrix to wheel matrix
        wheelMatrix[i][0] += turnMatrix[i][0];
        wheelMatrix[i][1] += turnMatrix[i][1];
        // update max magnitude
        double wheelMagnitude = Num.distance(wheelMatrix[i]); // fun fact: Num.distance(double... axis) can accept a 1-D array
        maxWheelMagnitude = Math.max(maxWheelMagnitude, wheelMagnitude);
    }
}

// divide all wheel vectors by the max magnitude so none exceed 1
for (double[] wheelVector : wheelMatrix) {
    wheelVector[0] /= maxWheelMagnitude;
    wheelVector[1] /= maxWheelMagnitude;
}
```

SETTING MOTOR OUTPUT

- Given a wheel vector $\vec{v} = \langle x, y \rangle$:
  - $speed = \|\vec{v}\|$
  - $angle = \theta = \tan^{-1}\left(\frac{x}{y}\right)$
    - Forward is 0 rad, increases turning clockwise, so we flip x and y
- Set the percent output of the drive motor to `speed`
- Use Motion Magic to set the turn motor to `Converter.radToEnc(angle, ticksPerRev)` (from FRC-217-Libraries)

```java
// speed is magnitude of vector
double speed = Num.distance(wheelMatrix[i]);
// only calculate angle if we're trying to move
if (speed != 0) {
    // calculate angle
    // NOTE: 0 rad is up, increases going clockwise, so x and y are flipped
    double angle = Math.atan2(wheelMatrix[i][0], wheelMatrix[i][1]);

    // set the turn motor
    kTurnMotors[i].set(ControlMode.MotionMagic, Converter.radToEnc(angle, 4096));
}
// set the drive motor
kDriveMotors[i].set(ControlMode.PercentOutput, speed);
```

SETTING MOTOR OUTPUT (SIMPLIFIED MATH)

- We can combine all the vectors together and calculate a simple equation for each wheel instead of doing it step by step
- Front left example:

$$r = Num.distance(botLength, botWidth)$$

$$x = strafe + turn\frac{botLength}{r}$$

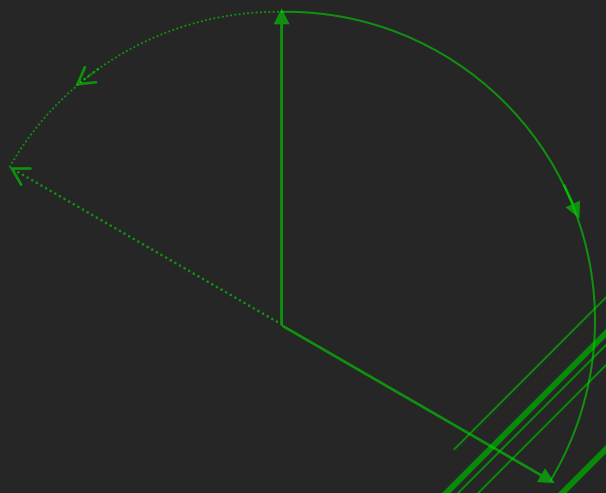
$$y = speed + turn\frac{botWidth}{r}$$

$$double\ flSpeed = Num.distance(x, y)$$

$$double\ flAngle = \tan^{-1}\left(\frac{x}{y}\right)$$

OPTIMIZE SWERVE ANGLE

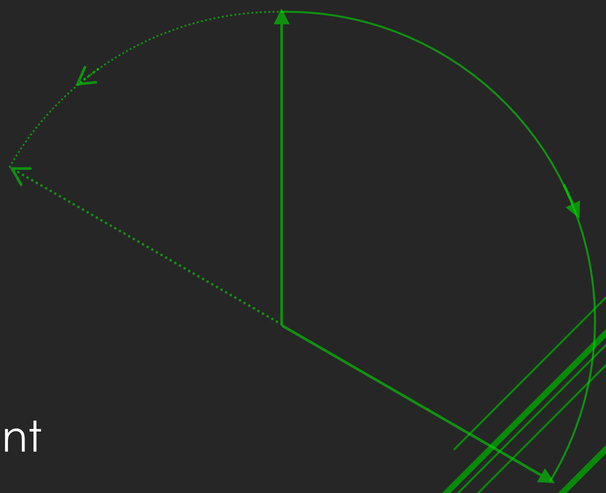
- If the last wheel angle was 0 rad, and the new target angle is $\frac{2\pi}{3}$ rad, we normally turn the wheel $\frac{2\pi}{3}$ rad
  - Inefficient for the turning motor
- Instead, flip the drive direction and turn to $-\frac{\pi}{3}$ rad
  - Turns the wheel half the distance



```
/*
 * We want to make it so the wheels turn as little as possible, so
 * we need to optimize the angle. Theoretically, a wheel should never
 * have to turn more than 90 degrees from its current position.
 *
 * The code block below is equivalent to and more efficient than:
 * while (angle - lastAngle[i] > Math.PI / 2) {
 *     angle -= Math.PI;
 *     speed *= -1;
 * }
 * while (angle - lastAngle[i] < -Math.PI / 2) {
 *     angle += Math.PI;
 *     speed *= -1;
 * }
 */
```

```
double angleDiff = angle - lastAngle[i];
// get how many half rotations we have to make to get within 90 degrees of lastAngle
// add signof(angleDiff) * Math.PI / 2 to angleDiff so we get within 90 degrees and not 180
int numHalfRotations = (int)((angleDiff + Math.signum(angleDiff) * Math.PI / 2) / Math.PI);
// subtract off that many half rotations
angle -= numHalfRotations * Math.PI;
if (numHalfRotations % 2 == 1) {
    // every half rotation, the wheel is flipped, so we need to flip speed
    // odd numbers of half rotations (% 2 == 1) results in a flipped speed
    speed *= -1;
}
lastAngle[i] = angle;
```

OPTIMIZE SWERVE ANGLE – EXPLANATION OF EFFICIENT ALGORITHM



- While the angle diff $> \frac{\pi}{2}$, subtract π and flip direction of speed
  - O(n) (linear) time complexity, takes longer if our wheels are rotated around multiple times
    - For absolute encoders that only read one rotation, this is very efficient
    - For relative encoders that read to `Integer.MAX_VALUE`, this is very inefficient
- Instead, for relative encoders:
  1. Calculate how many rotations of π we need
     - Any decimal remainder is our angle, so cast to int (truncate)
  2. Subtract off that many rotations of π
  3. Odd number of rotations = flip direction of speed
     - Rotating by π is a semicircle; an even number of π rotations is a full circle

FIELD SENSE

- Instead of speed and strafe being relative to the robot, make them relative to the field
  - Get angle from gyro/PigeonIMU
  - Find difference between angle and the "zero" angle for Field Sense
    - The "zero" angle is set when Field Sense toggles from disabled to enabled
  - Modify the speed and strafe inputs to form a vector with angle:
    - $\theta - gyroDiff$ if both $gyroDiff$ and the wheel angle each increase as they turn the same direction
      - Ex: as both turn right, both angles increase
      - $strafe = strafe\cos(gyroDiff) - speed\sin(gyroDiff)$
      - $speed = speed\cos(gyroDiff) + strafe\sin(gyroDiff)$
    - $\theta + gyroDiff$ if $gyroDiff$ and the wheel angle each increases as they turn opposite directions
      - Ex: the wheel angle increases as it turns right, but the gyro increases as it turns left
      - $strafe = strafe\cos(gyroDiff) + speed\sin(gyroDiff)$
      - $speed = speed\cos(gyroDiff) - strafe\sin(gyroDiff)$

FIELD SENSE – EXPLANATION OF MATH

- To rotate a vector, one must perform a linear transformation using the rotation matrix:

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

  - When $\theta = 0$, $R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = [\hat{\imath} \quad \hat{\jmath}]$
  - Increasing θ positively rotates vectors counter-clockwise
  - $\cos(\theta) = \cos(-\theta)$, but $\sin(\theta) = -\sin(-\theta)$, so adding vs subtracting $gyroDiff$ only flips the sign of $\sin(\theta)$
  - When we subtract $gyroDiff$, we want to turn counter-clockwise as $gyroDiff$ increases, which is equivalent to increasing θ positively, so $\theta = gyroDiff$
  - Matrix algebra:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = x \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} + y \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix} = \begin{bmatrix} x\cos(\theta) - y\sin(\theta) \\ y\cos(\theta) + x\sin(\theta) \end{bmatrix}$$