Swerve Programming

FRC team 6831 - A05annex, 25-Aug-2020

Last revision: 17 January 2022
github:

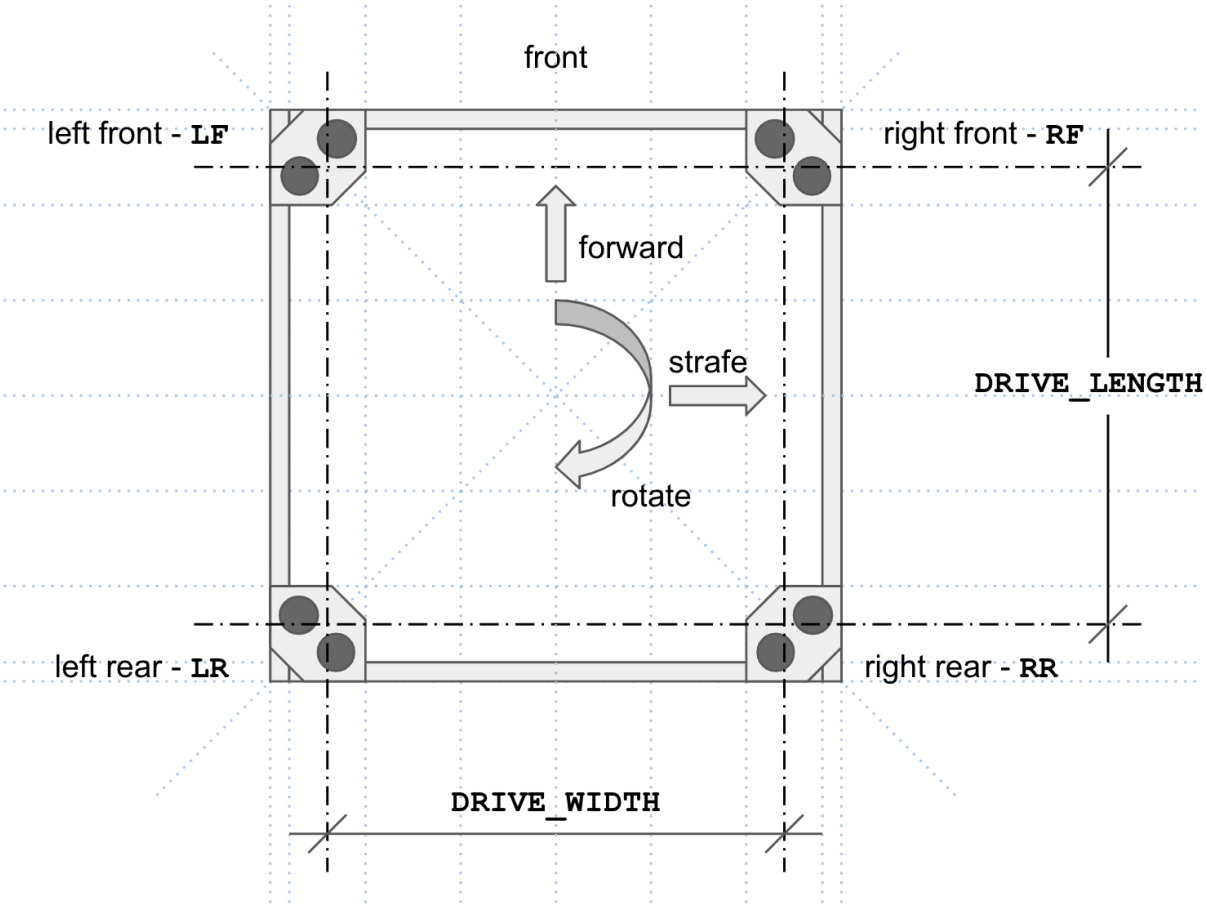
A lot of teams are using swerve drives and there are a lot of good tips, tricks, code, and lessons from their experiences. We built our first swerve prototype base using Swerve Drive Specialties MK2 Swerve Modules for the 2021 season. For the 2022 season we upgraded to Swerve Drive Specialties MK4 Swerve Modules (the last revision to this document). This document describes where we are with swerve, how we got here, and the other teams whose shoulders we've stood on to get here.

Contents

General Configuration

There are some conventions that show up in many of the team posts, and are useful to adopt. If we have more than one robot we need to follow the same conventions with both so programming is interchangeable.

This is our basic configuration - specifically, how are the CAN Ids assigned. Note that the MK2 to MK4 upgrade replaced a gear train to an analog encoder for direction (spin) calibration with a CANcoder directly sensing the spin shaft position (removes gear train backlash and simplifies the module) - so the major changes in this document revision are related to this module change.



module		CAN	encoder
RF	RF_DRIVE	1	NEO builtin plugged into spark motor controller
	RF_SPIN	2	NEO builtin plugged into spark motor controller
	RF_CAL	20	Calibration CANcoder
RR	RR_DRIVE	3	NEO builtin plugged into spark motor controller
	RR_SPIN	4	NEO builtin plugged into spark motor controller
	RR_CAL	21	Calibration CANcoder
LR	LR_DRIVE	5	NEO builtin plugged into spark motor controller
	LR_SPIN	6	NEO builtin plugged into spark motor controller
	LR_CAL	22	Calibration CANcoder
LF	LF_DRIVE	7	NEO builtin plugged into spark motor controller
	LF_SPIN	8	NEO builtin plugged into spark motor controller
	LF_CAL	23	Calibration CANcoder

```
DRIVE_LENGTH = 0.574          // meters, physically measured
DRIVE_WIDTH = 0.577           // meters, physically measured
MAX_METERS_PER_SECOND = 2.68  // from MK4 specs for our gear ratios
```

Programming

Programming swerve appears to have 3 major aspects:

- Controlling a drive module
- Mapping the stick to the robot - and everything I've run across is driver-field relative, not robot relative as we have always done in the past.
- Autonomous path planning

There is a ton of stuff online about algorithms for the second and third bullets; and almost universal angst that the programming really sucks on the first bullet. And let's face it - if your control of the drive module is not bulletproof, then it doesn't matter whether you do a good job on the other parts. Let's focus on module control first.

NOTE: we program in SI units, so while we include angles in degrees in the explanations, all diagrams use radians.

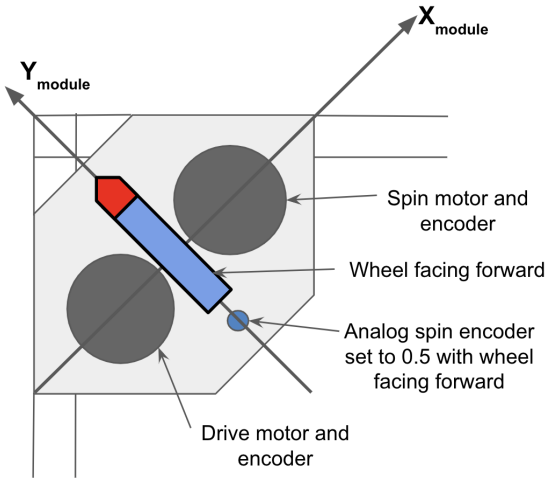
Swerve Module Control

The big thing we are seeing in the various threads and discussions is really around getting the module control right. The primary things are:

- Getting the alignment for spin 0.0rad or 0° (directly forward in robot coordinates) correct.
- Direction control:
    - Handling the -**π**rad to +**π**rad, or +180° to -180°, boundary so the spin is never greater than **π**rad, or 180°, during a command cycle..
    - Realizing that a greater than (**π**/2)rad, or 90°, and forward speed is exactly the same as a less than (**π**/2)rad, or 90°,rotation in the opposite position and the same speed in the backwards direction. This means the spin is never greater than (**π**/2)rad, or 90°, during a command cycle..
    - Choosing whether to use the analog direction encoder through a roborio PID loop or Neo motor encoder through the Spark controller for spin control..

When we were doing the arm for our 2018-2019 robot we used and analog encoder - we wanted to use a motor encoders through the talons for control of the lower and upper arms, but, at the time we did not have a motor-mounted encoder, and even if we did, we didn't know how to zero it (remember we needed to put the arm on the 2019-2020 robot in the correct initial position before we initialize the robot to set the bucket encoder? Same problem).

Our plan was to use the analog encoder to detect the current arm position and then zero the motor encoders (if we had them) and use motor encoders for the real arm control. We think that would be the ideal way to approach the swerve module control - i.e.:



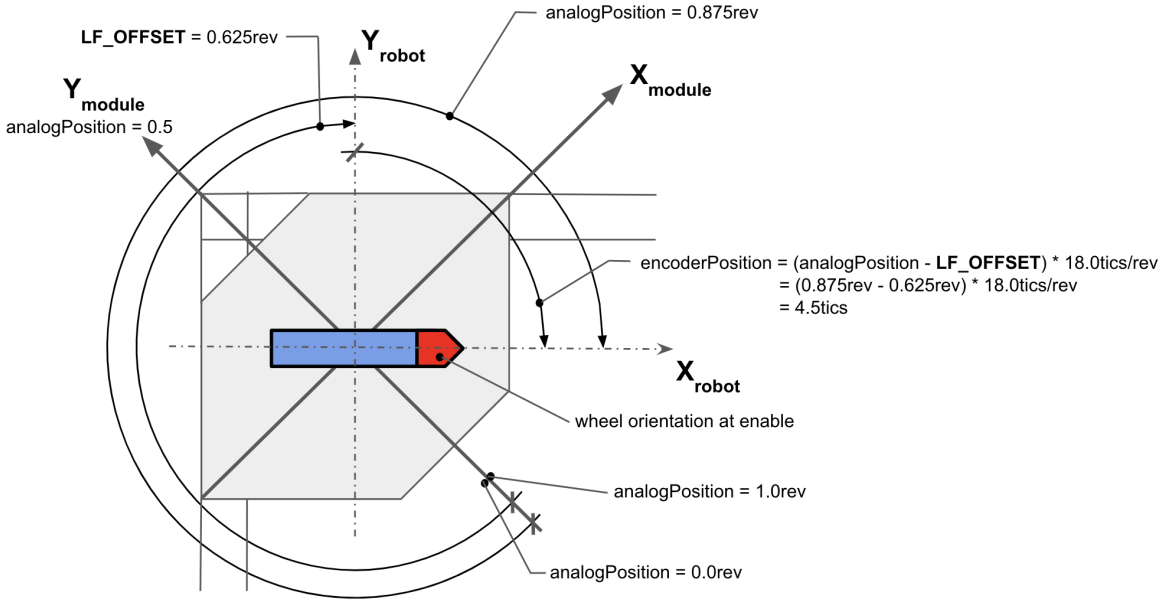
- Set the analog encoder to a known position at module assembly: see diagram to the left, the wheel should be oriented so forward is towards the corner of the robot when the analog encoder reads 0.5 (originally we thought 0.0, but the 0.0-1.0 boundary is pretty unstable). ***OK, this was the original plan - but MK4 encoders require the encoder magnet be glued into the direction shaft, and there is no adjustment. That means, instead, the encoder offset for each module when the wheel direction is directly forward must be carefully measured and recorded on the module so the module calibration offset constant can be updated if a module is replaced. The position encoder is initialized to report the position as 0 to 2π.***
- When the robot starts:
  - read the CANcoder to determine the current spin position;
  - Set the spin motor encoder to the current spin position.
- For the calibrated module, at robot enable, set the spin angle and speed to 0.0, 0.0 to bring the wheel to facing forward. All wheels should then be facing forward, so we can easily visually confirm correct initialization..

MK4 NOTE: Manually rotating the direction 10 spins resulted in a measured encoder change of 127.999, or 12.799 tics per spin.

MK2 NOTE: initial experiments with the spin encoder showed that 18 revolutions of the NEO motor resulted in a full 360° spin of the wheel, or, a single NEO motor revolution is equivalent to a 20° rotation {spin) of the wheel.

NOTE: drive motor control is pretty simple and is not discussed further here. We are using the closed loop position functionality of the SPARK MAX and setting the speed as described in the REV Velocity Closed Loop Control code example.

Swerve Module Calibration



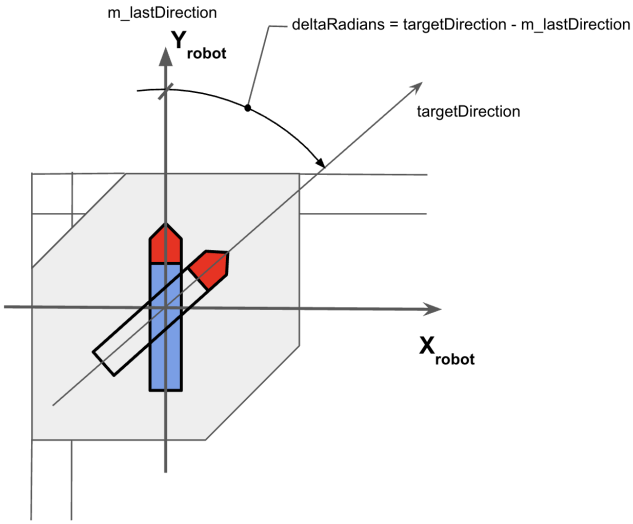
Once the spin encoder is zeroed the spin can be set simply by using the closed loop position functionality of the SPARK MAX and setting the position as described in the REV Position Closed Loop Control code example. We set the Kp to 0.25 and the spin was very snappy.

Swerve Module Spin Control

There are a few nuances for good spin control - lets layout the steps:

- Step 1: get naive control to work - be able to set a direction and speed and have the module get there.
- Step 2: spin in the smallest angle and go forward - which will be a maximum wheel spin of $\pm\pi$, or $\pm 180°$.
- Step 3: spin in the smallest direction for either forward or backward drive, and go the appropriate direction - which will be a maximum spin of $\pm\pi/2$, or $\pm 90°$.

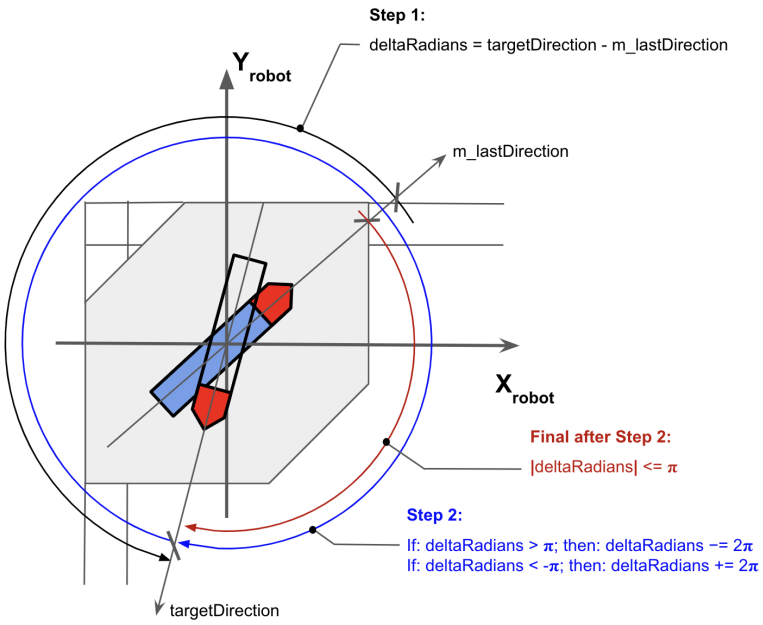
Step 1: Naive Control



Assume module state is set as a target direction (direction relative to Y_{robot} in the range ±π radians, or ±180°). This is pretty easy to program, since we've got the module direction calibrated. Assume that after calibration we set the direction to 0.0 (i.e. set the encoder to 0.0 so the wheel is facing forward) and saved them as: m_lastDirection as 0.0; and m_lastEncoder = 0.0.

Setting direction is now converting **`deltaDegrees`** to a **`deltaEncoder`**, and updating the encode position - remember that **`deltaDegrees/20.0`** maps degrees into the delta for the encoder - and since we have the last encoder position, it is easy to apply this delta and set the new encoder position. Remember, save **`m_lastDirection`** and **`m_lastEncoder`** so you are ready for the next control cycle.

Step 2: Rotate the Smallest Angle



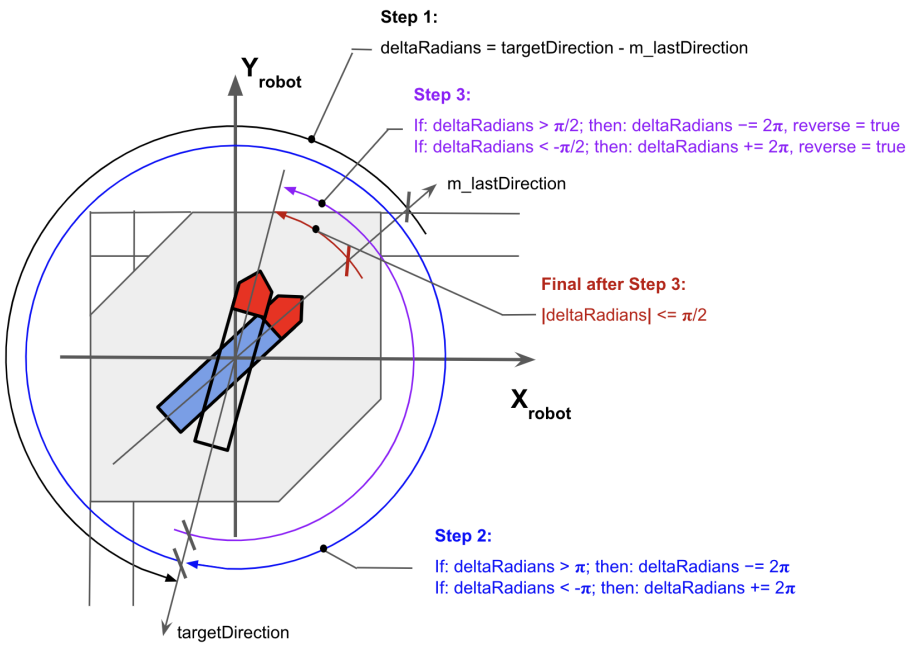
The primary problem with naive control is that you are often setting a large spin at the same time as you are setting a drive speed - we would like spin to be as small as possible so the drive speed is taking the module in the right direction - not accelerating in the wrong direction while the spin is trying to get to the right direction.

For this example, let's start at the **`m_lastDirection`** from the last step, and have a **`targetDirection`** nearly opposite the last direction (.i.e. the driver decided to rotate the chassis the other way, so the wheel is pointed in the wrong direction). In this case, computing **`deltaDegrees = targetDirection - m_lastDirection`** produces a rotation greater than

180° - which means the spin is in the wrong direction - specifically, the computed **`deltaDegrees`** is a bigger spin than is necessary. This would be particularly important if the **`m_lastDirection`** were 179° and the **`targetDirection`** were -179°, changing a counterclockwise spin of -358° to a clockwise spin of 2°. What does this look like in the module code? Pretty trivial, after you compute **`deltaDegrees`**, adjust it if it is more than 180° in either direction:

```
if (deltaDegrees > 180.0) {
   deltaDegrees -= 360.0;
} else if (deltaDegrees < -180.0) {
   deltaDegrees += 360.0;
}
```

Step 3: Rotate the Smallest Angle and Go Backwards or Forwards



Let's review the step 2 example a little more, and notice that we could have spun the wheel a very small amount and set the speed backward rather than forward.

If we are going to allow forward-backward switching we need a variable to track direction. We added **`m_speedMultiplier`**, which will be 1.0 if the speed is forward, and -1.0 if the speed is backwards.

Similarly to the last step, once we have computed the **`deltaDegrees`**, we check whether it is more than necessary (in this case a rotation more than 90°) and adjust it by 180° and set the **`m_speedMultiplier`** if necessary.

What does this look like in the module code? Again, pretty trivial, after you compute **`deltaDegrees`**, (as described in step 2), adjust it if it is more than 90° in either direction:

```
if (deltaDegrees > 90.0) {
   deltaDegrees -= 180.0;
   m_speedMultiplier = -1.0;
} else if (deltaDegrees < -90.0) {
   deltaDegrees += 180.0;
   m_speedMultiplier = -1.0;
}
```

Note, multiply the **`speed`** by **`m_speedMultiplier`** before you set the drive motor speed.

One more thing, the **`m_lastDirection`** is the previous **`targetDirection`** - which is now not necessarily the direction of the front of the wheel. When we start the **`deltaDegrees`** computation, we need to check **`m_speedMultiplier`**, and if it is negative, we need to compute the direction of the front of the wheel and use that in our computation of **`deltaDegrees,`** so this is the code at the beginning of the set speed and direction for the module:

```
double realLastForward = (m_speedMultiplier > 0.0) ? m_lastAngle :
       (m_lastAngle < 0.0) ? m_lastAngle + 180.0 : m_lastAngle - 180.0;
double deltaDegrees = targetDegrees - realLastForward;
m_speedMultiplier = 1.0;
```

Swerve Driver Control

The first thing we programmed was driver-in-the-robot control - because that is easy and lets you know that your swerve module code is working correctly. But, we soon discovered that it is really uncontrollable in terms of achieving anything the driver intends to achieve, so we switched to field relative.

Two important questions we discovered:

- **What is the module wheel direction when the stick is very close to centered?** The problem is the stick doesn't necessarily spring back to (x,y) = (0.0,0.0), so the direction is unstable, and any touch of the stick will flip the wheel direction. So we added deadband, but, since speed is a function of how far you push the stick in any direction, we used the actual distance from 0,0 for the deadband and speed computation (and later application of a power function to increase sensitivity near 0.0). We used the raw stick X,Y values in the **`Math.atan2()`** method to get the direction, so by the time the stick is out of the deadband the X,Y values are large enough that the **`Math.atan2()`** function is reliable.

- **What is the module wheel direction when speed is 0.0?** The answer we came up with is that the module direction remains unchanged when the speed is set to 0.0, but that is not controlled by the code module. It is controlled instead, by the drive subsystem that uses the module because the subsystem knows the context of use. The rationale is that it is better not to change the direction until you know where the module will go next.

Swerve Driver Control - Robot Relative

There are a lot of references on swerve math, and we can't determine the original sources for any of this material. We started with a jan 2011 post to chief delphi: Paper: 4 wheel independent drive & independent steering ("swerve"), and specifically these two attached papers (original sources are unclear):

- Derivation of Inverse Kinematics for Swerve (48.1 KB)
- Calculate Swerve Wheel Speeds and Steering angles (15.3 KB)

These papers/presentations give you all of the diagrams and math to understand what needs to happen in robot-relative swerve programming. We will just focus on some of the details here.

The first part of the implementation is just a vanilla translation from the papers:

```
private void swerveDriveComponents(double forward, double strafe,
                                   double rotation) {

   // calculate a, b, c and d variables
   double a = strafe - (rotation * LENGTH_OVER_DIAGONAL);
   double b = strafe + (rotation * LENGTH_OVER_DIAGONAL);
   double c = forward - (rotation * WIDTH_OVER_DIAGONAL);
   double d = forward + (rotation * WIDTH_OVER_DIAGONAL);

   // calculate wheel speeds
   double rfSpeed = Utl.length(b, c);
   double lfSpeed = Utl.length(b, d);
   double lrSpeed = Utl.length(a, d);
   double rrSpeed = Utl.length(a, c);
```

`LENGTH_OVER_DIAGONAL` and **`WIDTH_OVER_DIAGONAL`** and set during the drive subsystem instantiation. **`Utl.length()`** is a method to get the length from an n-dimensional set of components.

The next bit is a scaling function that limits the maximum motor speed to 1.0:

```
    // normalize speeds
    double max = Utl.max(rfSpeed, lfSpeed, lrSpeed, rrSpeed);
    if (max > 1.0) {
       rfSpeed /= max;
       lfSpeed /= max;
       lrSpeed /= max;
       rrSpeed /= max;
       forward /= max;
       strafe /= max;
       rotation /= max;
    }
```

Note that forward, strafe, and rotation are also scaled because we will be saving them for use in odometry.

The next step is setting the module directions. Note that when we do this we will be using the **`Math.atan2()`** method, and when the speed is near 0.0 the two arguments will be near zero which make the result questionable, and if they are both are 0.0 (the speed is 0.0) then the **`atan2`** is mathematically undefined - and in Java this condition defaults to 0.0.

So the question is where should the wheel point when the when speed approaches 0.0, and the best guess we could come up with is that the wheel speed goes to 0.0 because that wheel is temporarily the center of rotation for the robot, or the driver stopped momentarily and will do something, probably related to the last motion), and that the transition should be as smooth as possible. We concluded it was most likely that the next movement of the module would be closely related to the last, so keeping the wheel in the last orientation would be a better choice than setting some arbitrary position (like 0.0).

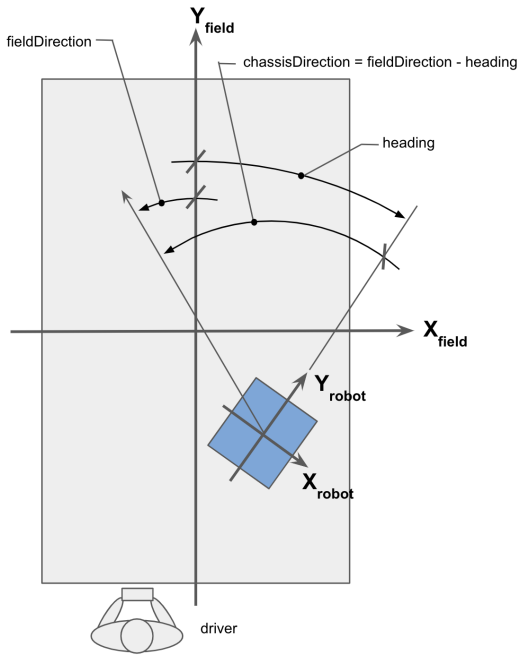
```
    // if speed is small or 0, (i.e. essentially stopped), use the last
    // angle because its next motion will probably be very close to its
    // current last motion - i.e. the next direction will probably
    // be very close to the last direction.
    m_RF_lastRadians = (rfSpeed < Constants.SMALL) ?
        m_RF_lastRadians : Math.atan2(b, c);
    m_LF_lastRadians = (lfSpeed < Constants.SMALL) ?
        m_LF_lastRadians : Math.atan2(b, d);
    m_LR_lastRadians = (lrSpeed < Constants.SMALL) ?
        m_LR_lastRadians : Math.atan2(a, d);
    m_RR_lastRadians = (rrSpeed < Constants.SMALL) ?
        m_RR_lastRadians : Math.atan2(a, c);

    // run wheels at speeds and angles
    m_rf.setRadiansAndSpeed(m_RF_lastRadians, rfSpeed);
    m_lf.setRadiansAndSpeed(m_LF_lastRadians, lfSpeed);
    m_lr.setRadiansAndSpeed(m_LR_lastRadians, lrSpeed);
    m_rr.setRadiansAndSpeed(m_RR_lastRadians, rrSpeed);
```

The last thing here is to save the forward, stafe, and rotation so they can be used in the odometry:

```
  // save the values we set for use in odometry calculations
   m_thisChassisForward = forward;
   m_thisChassisStrafe = strafe;
   m_thisChassisRotation = rotation;
}
```

Swerve Driver Control - Field Relative



Field relative simply means the motion of the robot is always specified relative to the field (or if the driver is stationary, relative to the driver). In this case the field is given an axis system so +Y is away from the driver and the direction the robot should move is the **`fieldDirection`** as shown in the figure.

In order to make this work you need to know the robot heading which we track using a NaxX board mounted to the roborio.

Field relative drive is pretty simple if the robot drive is specified by direction, speed, and rotation. In this case we need to change the **`fieldDirection`** into a robot relative direction, the **`chassisDirection`**, which is simply computed as:

```
chassisDirection = fieldDirection - heading;
```

Now that we know the mapping between field direction and velocity, and the chassis direction and velocity (note, the field velocity and the chassis velocity are the same)

Swerve Odometry

Looking at the WPIlib code we noticed that there was a swerve odometry class, and elected to do our own so we really understood how it worked. Here's how it works:

- Set the field location/heading of the robot the start of autonomous;
- Save the forward, strafe, and heading from the last command cycle;
- In the drive subsystem periodic:
    - update the robot heading
    - get the average heading (this heading and the heading the last time module speed and direction were set.
    - use the average forward and strafe speeds along the average heading to compute field delta X,Y and update the field location.
    - save all the current settings info to use as the previous state the next time we update the field position,

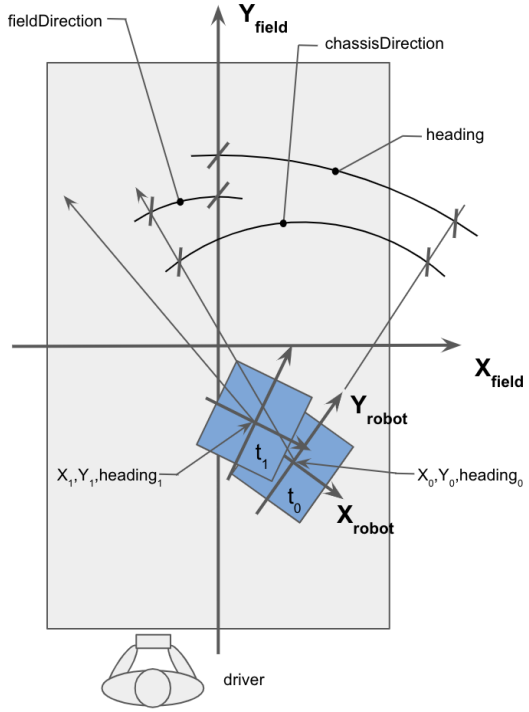
In order to get this to work you need to know how a module speed (0.0 to 1.0) and rotation (-1.0 to +1.0) map to the actual linear speed and/or rotation of the robot.

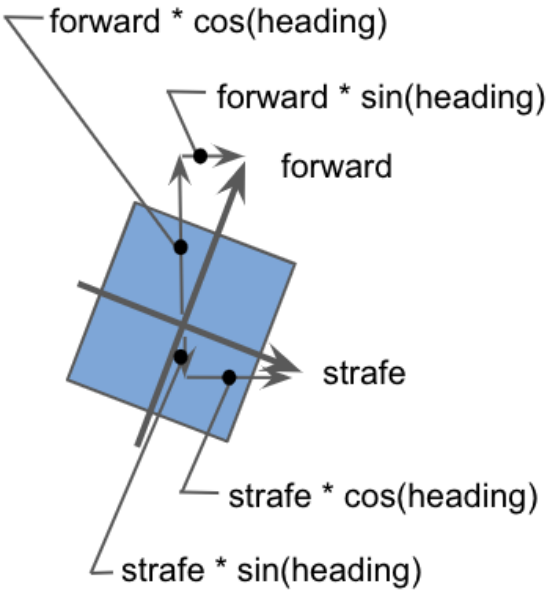
Let's start with the technical calculations based on motor specs, gear ratios, and geometry:

- REV Neo brushless motors, 12V nominal, 5676 RPM free speed (no load) - which we have seen as we started to program the modules.

- REV Neo motor encoders - 42 counts per rev. The REV software libraries account for this, so all velocity PID loop stuff is around the 5676RPM speed.
- Add a constant for our expected maximum speed set to 5000RPM - this is about 10% lower than the free speed. Why? To allow a little headroom for drag, load, and low voltage.
- For our Swerve Drive Specialties MK2 Module with our 14:42, 26:18, 15:60 gearing (overall ratio 8.31:1) the max free speed is specified to be 11.9 ft/sec, or, 3.62712 m/s.
- Scaling the max free linear speed based on our maximum motor speed we get a linear speed of 3.1951m/s at 5000PRM (need to field check this so we know this is a correct constant).
- Our robot has a centroid to module distance of 0.407m, which lets us compute a maximum angular velocity of 7.8505 radians/second. NOTE, max angular speed is the maximum linear speed divided by the radius from the centroid of the robot to the drive wheel center, so it is computed in the constants file using the robot geometry and max linear speed.

Now we have a starting guess for the mapping of the 0.0 to 1.0 speed to a real linear speed, and mapping the -1.0 to 1.0 rotation into a real angular velocity. Since we have tuned the velocity PID loops, the actual velocity should scale with the 0.0 to 1.0 requested velocity.





Let's start our discussion of odometry in the context of the command loop. At the beginning of each command loop the drive subsystem updates the heading, and will update the odometry. In the diagram, this update happens at position t_1, and the update from the previous command cycle happened at position t_0. If we know the field position, $(X_0,Y_0,heading_0)$, at t_0 can we compute the field position, $(X_1,Y_1,heading_1)$, at t_1? If we can do that reliably, we can determine our field position as a piecewise integration of the motion curve of the robot on the field. The normal command cycle is 20ms, so we should be updating this integration 50 times per second.

Assume we know the starting position-heading of the robot on the field along with its forward and strafe velocities (the t_0 location and state). The next command cycle will sense the heading and set the new forward and strafe velocities. If there is an angular velocity the path of the robot will be an arc we assume that using the average forward, strafe, and heading Approximates moving along the chord of that arc.

Once we have the average forward and strafe velocities along with the average heading we get the field x and y velocities as:

$$field_x = (forward * \sin(heading)) + (strafe * \cos(heading)$$

$$field_y = (forward * \cos(heading)) - (strafe * \sin(heading)$$

The drive system is responsible for updating the heading in the `periodic()` method, so the current heading is always available. We elected to add the odometry there so it is reliably updated in each command cycle.

```
public void periodic() {
    // Update the NavX heading
    m_navx.recomputeHeading(false);

    // Get the average speed and heading for this interval
    double currentHeading = m_navx.getHeading();
    double aveHeading = (m_lastHeading + currentHeading) * 0.5;
    double aveForward = (m_lastChassisForward + m_thisChassisForward) * 0.5;
    double aveStrafe = (m_lastChassisStrafe + m_thisChassisStrafe) * 0.5;

    // the the maximum distance we could travel in this interval at max speed
    long now = System.currentTimeMillis();
    double maxDistanceInInterval = Constants.MAX_METERS_PER_SEC *
                                   (double)(now - m_lastTime) / 1000.0;

    // compute the distance in field X and Y and update the field position
    double sinHeading = Math.sin(aveHeading);
    double cosHeading = Math.cos(aveHeading);
    m_fieldX += ((aveForward * sinHeading) + (aveStrafe * cosHeading)) *
                maxDistanceInInterval;
    m_fieldY += ((aveForward * cosHeading) - (aveStrafe * sinHeading)) *
                maxDistanceInInterval;

    // save the current state as the last state
    m_lastHeading = m_fieldHeading = currentHeading;
    m_lastChassisForward = m_thisChassisForward;
    m_lastChassisStrafe = m_thisChassisStrafe;
    m_lastChassisRotation = m_thisChassisRotation;
    m_lastTime = now;
}
```

This code should be pretty self explanatory. Note that we don't assume a common cycle time of 20ms, but we compute it for each cycle so we can compute the maximum distance if traveling at a normalized speed of 1.0. Initial testing of the odometry was promising, but was compromised by being in a rough non-level parking lot rather than a level carpeted surface.

Once we have a field surface we will need to do some physical measurements to confirm our technical calculations for **`MAX_METERS_PER_SEC`**, reset the constant and perform more controlled testing of odometry.

Lessons from Other Teams

Here are a couple tips we picked up:

- Calibrate your orientation (spin) encoder relative to the drive module, not the robot. Why? Because then all of the drive modules are identical and you can have a spare drive (or two) that is pre-calibrated and can replace any of the drive modules without recalibration.

- Put a wiring harness on the drive module that ends in a set of connectors for the motors and encoders so you can unplug and replace a module without tracing unbundling connecting and rebundling wires.