This is an update of the previous draft of this manual. This iteration attempts to provide foundation of knowledge required by students and to fill in more of the basic programming concepts. It continues to build the description for the CAN bus, vision systems and April tags.

This reorganizing and writing is not complete and may contain errors and omissions. Comments to improve this manual are welcome.

FRC 4513 Robot Manual

A lot of what you want to know about FRC robots,

but are afraid to ask.

Table of Contents

Purpose

The purpose of this manual is to introduce students and mentors to the FRC robots. This is a fairly general discussion and applies to the general archetecture of FRC robots, although it has information which may not be useful for your robot. The colorlary is that it may not have the information that you need. This document is primarily about the electrical and control systems in the robot. It provides an overview of the robot architecture and how that relates to the control architecture and software architecture. It also includes a brief overview of the Java programming language and its characteristics as an Object Oriented Language.

This paper develops the basic skills for working on a robot first with more advanced topics toward the end of the paper. If you (think you) know something, skip it. This is more of a manual than a text book. It just tries to fill in knowledge when it may be missing from other common sources.

More detailed general information can be found at https://docs.wpilib.org.

Basic Skills and Knowledge Required

Working on the robot requires skills. Some you can learn by doing "on the job" and some may have to wait until you are taught the subject in a class. Some you can learn on your own. Some knowledge may be out of reach for some students and even some mentors. Don't worry, there are still plenty of ways that you can contribute to your team. This section talks about some of the basic knowledge required.

The most basic skill is a willingness to dig in and learn. Whether this is riviting together the chassis, making bumpers, advertising the team's accomplishments, or figuring out how to use vision subsystems to the greatest advantage. This desire to learn and do is essential to working on the robot and contributing to the team effort.

Robots have a lot of wiring to deliver power and control to various components. Some knowledge of electricity is useful in doing these tasks correctly. A working knowledge of physics is useful to understand the effect of mass on acceleration and speed both in the robot power train and in various robot subsystems. An understanding of the basic machines explains the use of gear trains and constant force springs to achieve an objective.

Parts of software use a lot of math. In general the more math you know, the more you and your robot can do.

Some Basic Physics

A robot is all about basic physics. This is not necessarily about the formulas used, but the concepts.

FRC is all about speed. Speed is the change in distance over a period of time. You are used to miles per hour or kilometers per seconds, the software expresses speed as meters per seconds (m/s). How fast you change speeds is acceleration and that is expressed as meters per second (m/s^2). You can feel acceleration as your family car or school bus accelerates to reach a particular speed. Likewise you can feel deceleration as the vehicle puts on its brakes. When acceleration or deceleration is rapid, you fell that as well as a jerk (Yes, jerk really is a term in physics). Jerk is measured as meters per second^3 (m/s^3).

Newton's First Law of Mechanics has the following two parts:

- an object at rest will remain at rest until acted up by an external force
- an object in motion remains in motion in a straight line and constant velocity until acted upon by an external force.

This really means that force is required to affect change in the movement of an object. This is expressed with the formula:

$$Force = F$$

$$Mass = M$$

$$acceleration = a$$

$$F = M \cdot a$$

which using algebra can be rewritten as follows to solve for the acceleration.

$$a = F / M$$

What this formula means that with a fixed amount of force, acceleration is adversely affected by mass. The force available to a robot is largely fixed by the rules that limit the size of the battery and the particular motor types used. So the only variable on the right side of the equation is the mass of the robot. The heavier the robot, the slower its acceleration. The short hand way of saying this is that the robot acceleration is inversely proportional to the mass.

Newton's second law of motion states;

- The force acting upon an object is proportional to its mass times the square of its velocity or as expressed in the following formula

$$F = M \cdot v^2$$

What that means to robots is the faster you go, the harder you will hit.

Newton's third law of motion states:

- for every action there is an equal, but opposite reaction.

This has a lot of implications to robots. To accelerate, the robot exerts a force onto the carpet through its wheels. The carpet in turn exerts an equal but opposite force to the robot wheels. If there is no slip, the robot can be propelled in the direction that it desires.

The second and third laws combine and have catastrophic implications to robots. When a robot collides with another robot, all of its momentum and velocity are converted into a force. This force applies to both robots or to the robot and a field piece and can cause great damage to robot that is unable to absorb that force. Field pieces, in general, should be thought to have a large mass as they increase their mass by having a firm attachment to the carpet.

Most of this discussion has been about the forces of moving robots about the field. Many robots have arms and elevators that have to fight the force of gravity. The basic concepts are the same except that the acceleration is replaced with the gravitational constant G.

$$F = M \cdot g$$

where

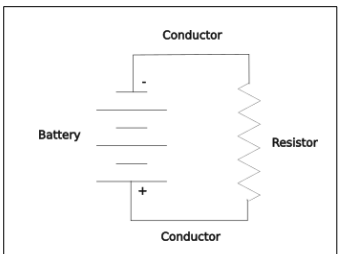
$$g \approx 9.8\, m/s^2$$

Gravity always pulls down on elevators and arms and this force must be accounted for in motor controllers usually as a feed forward term.

Some Basic Electrical and Electronics

An electrical circuit is a loop which moves electrons from a source to a component which consumes power and back to the source. A source is typically a battery, but it can be a power supply connected to power lines. The consuming component may be things like a resistive heating element, a light, a motor.

The battery could be "shorted" where its positive terminal is connected directly to the negative terminal. This is not a good idea, because the interconnecting wire was little resistance and a large flow of electrons will occur rapidly heating the wire. With a large enough battery, like that used in FRC robots, this heat can burn body parts or the conductor itself. This is one of the most dangerous parts of the robot so it should be treated with respect.



A simple circuit

In this current the current flows from the positive terminal of the battery through the conductors and the resistor to the negative terminal of the battery. The resistor resists the current flow and heats up. In this circuit the current is called **Direct Current** or DC as it flows in only one direction, positive to negative.

There are few concepts about this circuit. The battery supplies power. It has a **voltage** which the pressure that moves the electrons. Higher voltages have higher pressures. The **current** flow is measured in **amperes** or amps. **Resistance** is measured in **ohms**. A resister with more resistance has more ohms. Power is suppied by the battery and consumed by the resistor. Power is measured in watts which

These three values can be measured with a multimeter. Voltage is measured across a component like the battery or resister. This connection is in **parallel** with the component. DC current is measured by **breaking** the circuit and then recompleting the circuit with the meter. The meter is in **series** with the circuit.

TODO show how to measure votage.

TODO show how to measure current.

TODO show how to measure resistance with a warning against measuring in circuit or under power.

TODO show how to uses a switch to control the ciruit… show both open and closed.

TODO show how a fuse or circuit breaker can control the circuit sort of like a switch, but not designed for repeated open and close cycles.

Power in your home or school is called **Alternating Current** or AC. It goes through a cycle of sending the current in one directed and then sending the current in the opposite direction. AC is used in power distribution because it is easier to change its voltage up or down using transformers. A transformer has two coils where the magnetic field of one coil is coupled to the other coil. This coupling allows current in one winding be induced into the winding of the other coil and hence the current is able to pass through the transformer. The ratio of the number of windings of the two coils determines the ratio of the input voltage to the output voltage. So if the input coil has twice as many winding as the output coil, the output will have half the voltage as the input. This is a step down transformer. If the ratios were reversed, the output would have twice the voltage as the input. This is a step up transformer. AC current measurement may be measured like DC currents or it can use a "clamp" device to encircle a conductor and sence the current in the wire inductively just like a transformer. This works only for AC because the electrical inductance requires that the voltages be changing for currents to be induced.

The relationship between voltage (E), current (I) and resistance (R) is show in the following:

$$E = I \cdot R$$

This may be rewritten as the following to solve for current or resistance

$$I = E / R$$

$$R = E / I$$

a fourth electrical measurement is power (P) expressed as watts. It is the product of voltage and current:

$$P = E \cdot I$$

This can be rewritten when combined with the previous expressions as:

$$P = I^2 \cdot R$$

$$P = E^2 / R$$

These last two relationships show something interesting when trying to move power from one place to another. All real world conductors (which rule out super conductors in a laboratory environment) have resistance and that resistance means some power is lost in delivering power. Bigger conductors of the same material have less resistance than smaller conductors. Some materials like gold and silver conduct with less resistance than copper or aluminum, but are also more expensive. For a given power line with a fixed amount of resistance, more power can be delivered with less loss using a higher voltage than a lower voltage. Also the same power can be more efficiently delivered with a higher voltage and a lower current and a smaller more economical wire. Transmission lines have traditionally used alternating current because of its ease of transforming its voltage.

Energy is power used over time and is typically measured in watt-hours or kilowatt-hours. Batteries take a slightly different approach and use amp-hours or ampere-hours as they have a relatively fixed voltage.

Robots use direct current because it is convenient to do so. Batteries provide a portable source of direct current. Robots use a lot of current. The main breaker is 120 amps. Older homes may have 100 amps or less, while modern homes have at least 200 amps. Of course a home uses more power by using a higher voltage of 220 volts split into two sides of 110 volts each. The point is that the robot uses a lot of current and has to have heavy wiring to accommodate the high current that it uses. One can see the large wires coming off the battery. These carry the entire 120 amps through a circuit breaker to the distribution panel. This circuit breaker is a safety device design to “trip” or break the circuit should more than 120 amps be used. The power distribution panel has access to all 120 amps, but it parcels out the power to individual circuits, each with their own circuit breaker or fuse that protects that individual circuits. Circuits for motors use 30 to 40 amps and some circuits for electronics may be five to tens amps. One can see different sized wires in the robot. Large current circuits use large wire and small current wires use smaller wires. [If you are following the math you should note that you really cannot have four 40 amp motors running at the same time, The various controller have to limit the current comsumed so that there is enough for all required functions.]

In the example of a simple circuit a battery was shown with a positive and a negative terminal. The battery is said to be **polarized**. Within the robot many components are also polarized, especially components with electronic controllers. It is import that the polarity of the circuits be maintained through the robot so that components are not destroyed. Red is positive and Black is negative. There is no chassis ground in FRC robots, so the chaisis should be neutral. This is done for safety so that should a wire come in contact with the chassis it will not complete an existing circuit.

A robot will have analog and digital circuits. Many circuits on the robot are either on or off and just carry a fairly even voltage. Power distribution throughout the robot is analog. Some sensors provide an analog signal as a voltage. Digital circuits are characterized by sending information as a series of ones and zeros encoded to a specification of the interface. The digital circuits are typically communication channels like Ethernet, USB (universal serial bus), CAN (controller area network) or I^2C.

Switched Circuits

TODO fill this out.

Open vs closed circuits

How to switch

mechanical switch.

fuse

circuit breaker

electronic

Analog vs. Digital

TODO fill this out.

Analog is continuously variable voltage. It may have limits (usually the power supply limits).

Digits is view on having two distinct values: 1 and 0. These values may be expressed in different ways a voltage, a current, a differential voltage. Ultimately all circuits are analog and the limitations of digital signals is because of their inherent analog limitations. Transistor-transistor-logic (TTL) used a +5v power supply. A zero was a voltage under 0.8v. A one was a voltage over 2.4 volts. These voltages are what is indicated by the opening and closing of the transistors or a logic gate. Different logic families use different voltages and may not be compatible with each other. In between the maximum voltage expressing a zero and the minimum voltage expressing a one is ambiguous, but that zone has to be crossed in changing a signal from a one to a zero or vice versa. Although this transition can be fairly quick it does take a finite amount of time. One strategy is use a clock signal to time when to look at a changing signal to sample in the middle of a bit time and avoid the two edge transitions.

Power Distribution

TODO fill this out.

Components of Distribution device are rated for full current.

Fuses or circuit breakers limit the current and hence the power of branch circuits

One motor per power distribution circuit to avoid masking problems.

Field Programmable Gate Array

TODO fill this out.

Device which can be programmed to be a set of logic elements to perform a specific function. This replacees the technology where this logic was custom buit using specific logic elements.

Very Fast

Used in roboRIO to provide custom circuitry for PWM and for controlling LEDs

LEDs

TODO fill this out.

How they work

- each WS2812 cell is an RGB LED and has its own microcontroller and oscillator
- Once a color for a cell is set, the controller keeps the LED at the same color with its own PWM controller for each LED color

each cell draws 50ma, so a dense strip of 144 cells/m draws 7.2 amps which is a lot on a robot. Decrease power by:

  - decreasing the brightness
  - avoiding white which uses all three LED colors
  - use patterns that do not use every LED
  - use flashing or other periodic technique to increase off time

- Talk is to add a 220 to 470 ohm resister to the data line and a 100uF to 1000uF capacitor across the supply of the strip to smooth out power and reduce noise.
- WPI LIB use 4 bytes per cell but does not support RGBW (as of 2024).
- send data at 800Kbps NRZ (not return to zero)l…. really a modified Manchester code to be self-clocking. There is a positive transition at the beginning every bit and a negative transition part 1/3 or 2/3 of the way through. The value at the mid way point is the value of that bit.
- 0 = T0H + T0L
- 1 = T1H + T1L
- T0H = 0.4us +/- 150ns typ
- T1H = 0.8us
- T0L = .85us
- T1H = 0.45s
- (bit is 1.25us)
- reset = Treset > 50us
- cascade din==dout---din==dout--din===dout…
- send bits one ofter the other in blocks of 24 with reset code in between complete strings
- Each block is: G7…G0-R7…R0-B7…B0
- really should say strings are sent one 24-bit block after another with a reset in between strings
- strip timing is 30 x n +50 … so 144 LED strip takes 4.37 ms
- Command strategy
  - clear buffer (may not be necessary with a bit of planning e.g., if only updating changed sections)
  - use a modulo of the time to get a number that sweeps though a set of scenarios. e.g. use a number to move a spot of light from one end of a section to the other at some rate. The base modulo should be the period of the section display cycle. Individual steps may be a sub-modulo within that period.
  - write into each section the updates that are needed
  - send the buffer to the LED strip

P-I-D Controllers

P-I-D is a control algorithm which is base on Proportion-Integral-Differential parameters. It uses feedback to find out the error between the current position and the desired position. A constant factor called K_i is multiplied against that error and fed as an input to control the motor. A low K_p will be slow to reach the goal and a high K_p will overshoot the goal. A K_p that is just right will quickly reach the goal and reduce the error to near zero. The K_i constant is a multiplied against the integral of past error values, A successful K_i value will reduce the integral to zero. The K_d term is multiplied by the differential of the past error values. It anticipates future values based on the change of past values. Tuning is the process of selecting the correct values of K_p, K_i, and K_d often using a trial and error technique.

Many motors in current robots have built-in controllers that use the P-I-D algorithm.

Torque is dependent on applied powerPower Distribution

TODO fill this out.

Electric Motors

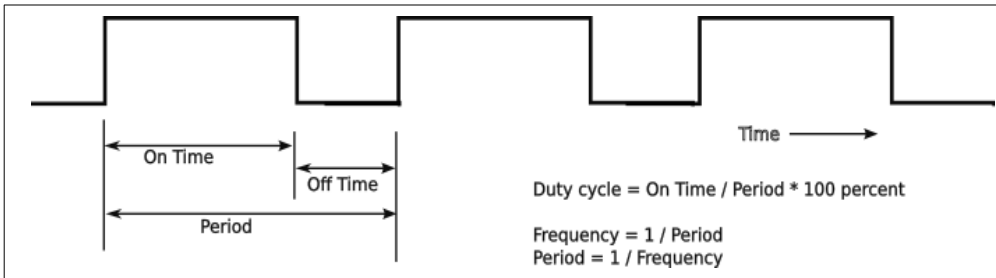
TODO fill this out.

Speed is dependent on applied power

Torque is dependent on applied power

Pulse Width Modulation (PWM)

Some motors and lights can be controlled with a **Pulse Width Modulation (PWM)** signal. In PWM the voltage is turned on and off at a set frequency or period. The duty cycle of the on pulse is varied. The percentage of on time to pulse period is the duty cycle and that cycle determines how fast a motor can go or how bright a light will be. A 100% duty cycle is on all of the time; 50% is on half the time; 10% is on 10% of the time. This means that a 50% duty cycle 12 volt motor will run at the same speed if it were supplied 6 volts. Some care must be used with PWM to ensure that the correct base frequency is used. The roboRIO uses three periods 5, 10 and 20 ms (or the frequencies 200, 100 or 50 pulses per second (pps) respectively). Motors have a lot of mass compared to most electrical components so they treat a PWM voltage almost the same as the equivalent analog voltage. It is easier for computer controlled circuits to provide the PWM voltage than the equivalent analog voltage.



Duty Cycle of a Voltage Waveform for use as Pulse Width Modulation

In the roboRIO the PWM signals are generated by a programmable gate array which is a programmable circuit. This flexibility allows the same circuitry to provide the low level signal required by addressable LED strings.

Gyroscope

TODO fill this out.

How do they work.

What output do they have.

What needs to be done to make them useful.

Some Basic Math

TODO fill this out to introduce the topic. Arrest concerns about trig, geometry, vector math.

Distance and Angular Measurements

Most of the math for a robot has to do with knowing the robot's position. This may be how far a wheel has turned, the position of an arm or the speed of a shooter. Distances in the field are measured in two units: inches when you read field measurement and CAD drawings and meters when using the code libraries.

$$1\,meter = 39.37\,inches = 3.3\,feet$$

Distance is a major factor during autonomous, but during teleop, movements are specified in velocities rather than distance. You use a joystick to control the velocity of the robot. Velocities are measured in meters per second (mps or m/s). When you get into the controllers you will find accelerations or changes in velocity measured in meters per second per second (m/s^2). Advanced controllers worry about the smoothness of the control and uses a measurement of the changes of acceleration or jerk as meters per second per second per second (m/s^3).

Measurement	**Name**	**Example units**
distance	distance	meters (m), inches (in), kilometers (km), miles (mi)
change of distance over time	velocity or speed	m/s, km/hr, mi/hr
change in velocity over time	acceleration	m/s^2
change in acceleraton over time	jerk	m/s^3

Angular measurement is a bit more difficult. We have be trained to use degrees and we intuitively know what 360, 180, 90 and 45 degrees mean. So the human interface usually use degrees to express an angular measurement. Math routines in software like angles expressed in radians. It is easy to convert degrees and radians. Shaft encoders report angular measurement as clicks, an analog voltage, or a number. To simplify the confusing, the software libraries use the notion of rotations. It is convenient for calculations and it is easy to convert it to other angular measurement systems as required.

$$2 \cdot \pi radians = 360\,° = 1\,rotation$$

A note about angular measurements and negative numbers. A positive rotation can be though to be clockwise and a negative rotation can be thought to be counter clockwise. Angles can be accumulated to be more than one rotation positively or negatively. Sometimes this is useful, like when you want to know how far a wheel has turned. But if you want to know what position a wheel or shaft is in now, you want to convert the indicated reading into an absolute angle. This is done by

1. taking the modulo of the angle in the units per rotation. This results in a number that is between -1 rotation and +1 rotation in the units per rotation.
2. Add the number of units per rotation to the results of #1. This results in a number that is between 0 and +2 rotations in the units per rotation.

3. Lastly take the modulo of the angle in the units per rotation. This results in a number that is between 0 and +1 rotations in the units per rotation.

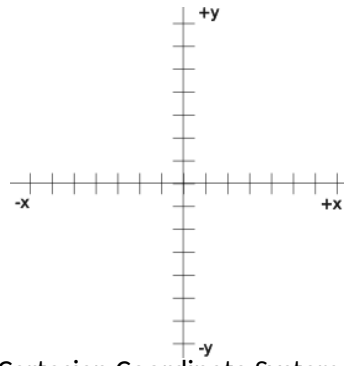
The problem in measuring angles in relative terms is that there is a discontinuity or anomaly at the boundary between 0 and 1 rotations. Some software takes care of this with a “continuous” flag which allows calculations to bridge over the anomaly.

TODO generate graphics to show this anomaly 0-359 or -179, +179

Coordinate Systems

It is assumed that you know about the Cartesian Coordinate system. It locates a point in two or three dimensional space. This simple coordinate system is expanded upon in robotics to form a set of reference systems. This allow a point to be located on a field or on a robot and to go between these different reference systems.

Positions within a robot are assigned on a cartesian coordinate system based on the physical center of the robot and are measured in inches from the center point. The front and back of the robot are a little ambiguous, but must be defined and this agreement must stick for the life of the robot to avoid confusion as it forms the basis for many calculations involving components of the robot.



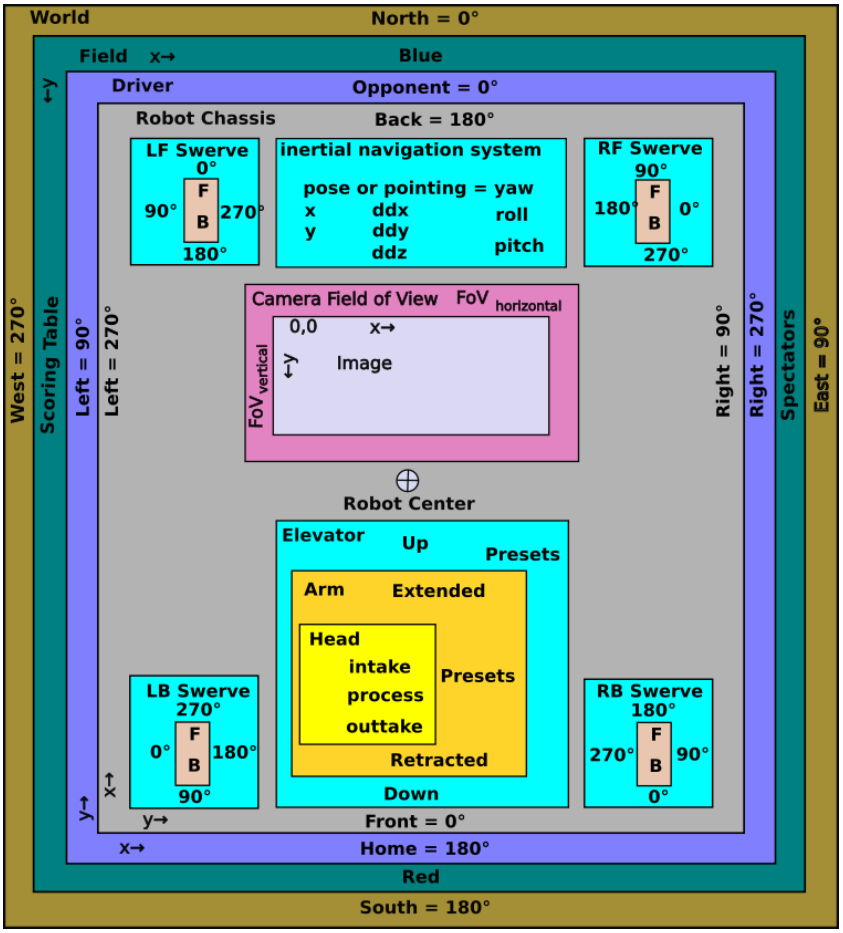
Cartesian Coordinate System

Frames of Reference

Many robots in the FRC do not care much about coordinate systems. They just drive based on relative positions and do not take into consideration their position within the field. If your robot fits into that description, you can skip this section.

A frame of reference basically is a coordinate system. In FRC robotics several frames of reference are used, some with the same origin, some with different origins, some with different orientations, and some with different rotations. These frames of reference should be kept in mind when converting from one frame of reference to another.

Most robot components are mounted in fixed positions relative to the center of the robot. Some robot components are mounted on things which move relative to the center of the robot like arms, intakes and elevators. If ongoing measurements are required from moving components then a translation or transformation must be computed to resolve the location of the component to the robot and to the field.



Robot Frames of Reference

A frame of reference defines where angles are measured from and where distances are measured from. So it changes depending on which frame of reference you are using. A camera is always in the same postion on a robot, but what I sees depends on where the robot is located on the field and where it is pointed. Measurements between a particular frame of reference and the containing one are used to convert a location in one frame of reference to another. Most devices on a robot are do not share the robot center as their center so their frame of reference must be translated to accommodate the difference in center points.

Nearly every device on the robot has at least one angle or position that it is concerned about. In simple robots, everything can reference the robot chassis. Its front is at 0 degrees and angles increase counter-clockwise looking down from above (to agree with mathematical conventions). Driving the robot is much easier when looking at the field from the driver perspective. Down field is 0° and degrees, as in math, increase counter clockwise, so left is 90°, backward is 180° and right is 270°.

In a swerve drive train there are four swerve drive assemblies. These each have a different positional relationship with respect to the chassis. Not only is their x and y coordinate different for each corner, but the orientation of each drive is different. Further it must be noted which direction if forward for each motor, as in a swerve drive it is hard to tell by looking. This can be expressed as a x, y measurement from the center of the robot to the vertical axis of the swerve module. It is also important to know at what angle forward is within the robot. Luckily this calibration normally only has to happen once during the build, although it may have to be re-calibrated in the field as a result of a collision or loss of data.

In the world frame of reference, degrees increase clockwise, so north is 0°, east is 90°, south is 180° and west is 270°. At least so far, this frame of reference is not used very much, but it could be if the magnetic compass sensor or a GPS sensor is used. Rarely will a field be laid out on a north-south line, but will be on some other angle. Robots are normally initialized to the field orientation and are not concerned with the real world.

Mirroring

In recent competitions the layout of the field from the red alliance perspective is a mirror image of the blue alliance perspective. This was supposed to be supported by the path planner in 2024. The math for mirroring blue alliance coordinates to red alliance coordinates is:

$$x_{red} = x_{blue}$$

$$y_{red} = fieldLength - y_{blue}$$

$$angle_{red} = angle_{blue} + 180\,°$$

This has an impact to automated sequences during both the autonomous period and the teleop period. The path planning routes are supposed to be reflected about the center, so that a single path can be used on either the red or the blue alliance side of the field. [We did not have much luck doing this in 2024.]

The odometry subsystem keeps track of where the robot is during a competion. It keep track of how many turns a wheel turns and in what direction. Specifically it looks at each wheel angle with respect to the field and adds in the average wheel velocity for a period times the length of the period. All four wheels are accounted for including their position with respect to the robot center (as recorded in the robot kinematics configuration). These incremental movements are added as a vector sum to determine the x-y motion of the robot and the robot's angular rotation. Luckily this is all done under the covers of the robot library. These measurements are done with as small of an increment as possible to increase its accuracy. Even so this is susceptable to errors introduced by spinning wheels, flying robots and collisions. Vision processing of April tags is used by some teams to reset the odometry to keep it accurate.

Pose

The WPI LIB uses a construct called Pose to express the position of the robot (and other objects) on the field from the field frame of reference. The common pose uses a Pose2D object consisting of three attributes:

- x-coordinate expressed in meters (although field drawing use inches)
- y-coordinate express in meters.
- Facing angle expressed as rotations (but easily converted to degrees or radians).

This if fine for things that are on the floor of the field, but a lot ot things like April Tags, cameras, shooters and scoring targets are off the floor. To express their location WPILIB uses the Pose3D object which consists of six parts:

- x-coordinate expressed in meters (although field drawing use inches)
- y-coordinate express in meters.
- z-coordinate expressed in meters.
- Yaw or facing angle expressed as rotations.
- Pitch or angle about a horizontal axis perpendicular to the facing expressed as rotations .
- Roll or angle about a horizontal axis parallel to the facing angle expressed as rotations .

Geometry

Geometry is one of those things that after a while you take for granted and do not know that your are using it. The geometric theorems may be useful at times to solve some angle problems. In general most robot math does not require this. Some of the most important things to remember is just some basic rules like:

- The sum of the angles that form a straight line total 180 degrees.
- The sum of the angles in a triangle is 180 degrees.

- The Pythagorean theorem for finding the sides of a right angle triangle: $a^2+b^2=c^2$

  a simple proof of the theorem (not really necessary, but fun):

  1. construct a square with sides a + b
  2. connect the a-sides with the b-sides to form four triangles a-b-c and opposite angles A, B, C.
  3. The angles of the triangles are A, B, and C, so:
     $A+B+C=180°$
  4. Since C is both a corner of the original square and the corner of the triangle a-b-c:
     $C=90°$
  5. This means that since:
     $A+B+C=180°$ and
     $C=90°$, that:
     $A+B=90°$
  6. since a straight line has 180°
     $A+B+D=180°$ and
     $A+B=90°$, so:
     $D=90°=C$ and the inner area c-c-c-c is a square
  7. The area of the inner square is just:

     $$Area\ inner\ square=c\cdot c=c^2$$

  8. The area of each triangle is:

     $$Area\ triangle=(base\cdot height)/2=(a\cdot b)/2$$

  9. The relationship between the area ot the outer square and its constituent triangles and inner square is:

     $$Area\ outer\ square=4\cdot((ab)/2)+c^2=2ab+c^2$$

  10. Just squaring the side to find the area of the outer square and reducing:

      $$Area\ outer\ square=(a+b)(a+b)=(a+b)^2$$

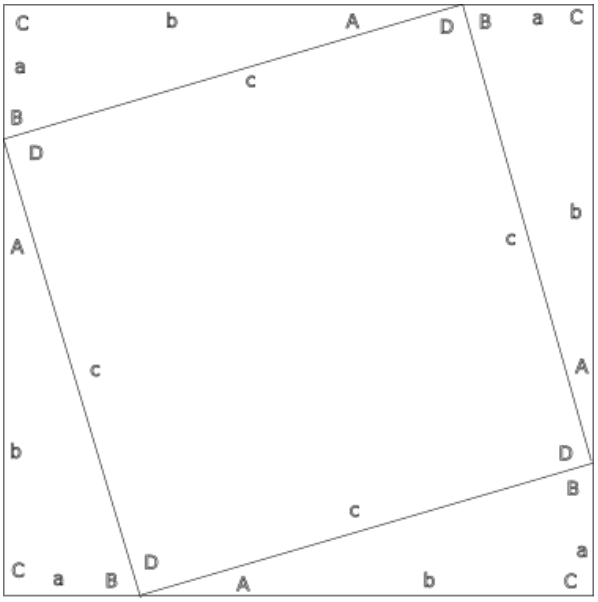
      $$Area\ outer\ square=a(a+b)+b(a+b)$$

      $$Area\ outer\ square=a^2+ab+ab+b^2$$

      $$Area\ outer\ square=a^2+2ab+b^2$$

  11. Combining 9 and 10 above the area of the outer square is:

      $$Area\ outer\ square=a^2+2ab+b^2=2ab+c^2$$



Proof of Pythagorean Theorem

12. Subtracting *2ab* from both sides leaves:

$$a^2+b^2=c^2$$

$$Q.E.D.$$

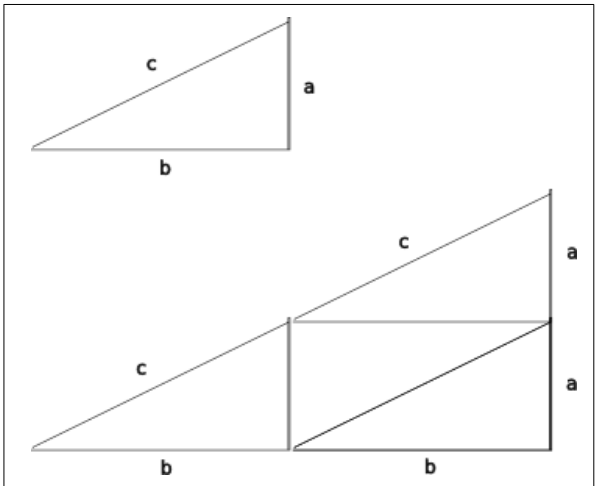
- The Pythagorean theorem can be rewritten as follows to solve for the length of the (*c*) or one of the short sides (*a*):

$$c=\sqrt{a^2+b^2}$$

$$a=\sqrt{c^2-b^2}$$

Trigonometry

Trigonometry is used quite a bit in odometry, swerve drive subsystems and in April tags so a short discussion is warranted. Trigonometry, or trig, is cool because it allows finding the length of sides and angles of triangles when only three are known (i.e., three sides, three angles, an angle and two sides, or two angles and a side).



Intuitive Trig Proof

The fundamental observation of trigonometry is that the ratio between the lengths of the sides of triangle with the same angles does not change with the lengths of the sides are increased. If you double the length of the sides of a triangle the ratios between the sides remain the same.

$$a^2+b^2=c^2$$

$$3^2+4^2=5^2$$

$$area=\frac{ab}{2}=3\cdot 4/2=12/2=6$$

double each side, and the calculations become:

$$a^2+b^2=c^2$$

$$(2\cdot 3)^2+(2\cdot 4)^2=6^2+8^2=36+64=100=10^2=(2\cdot 5)^2$$

$$area=\frac{2\cdot a\cdot 2\cdot b}{2}=\frac{2\cdot 3\cdot 2\cdot 4}{2}=\frac{6\cdot 8}{2}=\frac{48}{2}=24=2^2\cdot 6$$

Trigonometry refers to an angle as being adjacent or opposite to an angle (and it ignores the right angle) so a sine (abbreviated sin) is the ratio between the opposite side and the long side or hypotenuse:

$$\sin A=\frac{opposite}{hypotenuse}=\frac{a}{c}$$

Cosine (abbreviated cos) uses the ratio between the adjacent side and the hypotenuse:

$$\cos A=\frac{adjacent}{hypotenuse}=\frac{b}{c}$$

Tangent (abbreviated tan) uses the ratio between the opposite and adjacent sides:

$$\tan A = \frac{opposite}{adjacent} = \frac{a}{b}$$

Cotangent (abbreviated cot) uses the ratio between the adjacent and opposite sides:

$$\cot A = \frac{adjacent}{opposite} = \frac{b}{a}$$

When you know the angles, but not the angle there is an inverse set of functions. arcsine (abbreviated asin or $\sin^{-1}$) is the angle derived from the ratio between the opposite side and the long side or hypotenuse:

$$asin(\frac{opposite}{hypotenuse}) = asin(\frac{a}{c}) = A$$

Arc cosine (abbreviated acos or $\cos^{-1}$) uses the ratio between the adjacent side and the hypoteneuse to find the angle:

$$acos(\frac{adjacent}{hypotenuse}) = acos(\frac{b}{c}) = A$$

Arc tangent (abbreviated atan or $\tan^{-1}$) uses the ratio between the opposite and adjacent sides to find the angle:

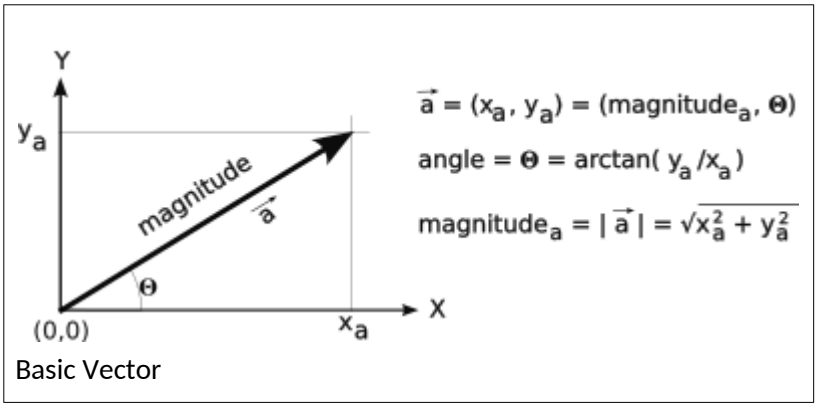
$$atan(\frac{opposite}{adjacent}) = atan(\frac{a}{b}) = A$$

There isn't really an arc cotangent as you can just work with the other angle and an arc tangent function to find both angles.

Some Advanced Math Topics

Vectors

Vectors simplify the discussion of how some things in robots are calculated and should be considered an advanced topic. Vectors can be used in translating a desired movement of the robot to its drive train, whether it is a tank drive or a swerve drive. Vectors can also be used in translating between frames of reference.



Basic Vector

In its simplest form a vector is a direction and a magnitude. It could be something list a 30 MPH wind from the northwest. The magnitude in this case is 30 MPH and the direction is from the northwest (or toward the southeast). For robot movements in the WPILIB the magnitude is expressed in units of meters per second. Rotational speeds are expressed in radians per second or rotations per second. Radians are used by the Java (and other programming languages) math routines.

$2 \cdot \pi \cdot radians = 360° = 1 rotation$

$2 \cdot \pi \cdot radians = 360° = 1 rotation$

A vector can be expressed as its angle and magnitude or as its x and y components as shown in the figure above.

$\vec{a} = |\vec{a}| \angle \Theta = (x_a, y_a)$ *where* $\vec{a}$ *is the shorthand notation for vector a*

where:

$\vec{a}$ *is the shorthand notation for vector a*

$|\vec{a}|$ *is the notation for the magnitude of vector a*

$\angle \Theta$ *is the notation for the angle* Θ *of vector a*

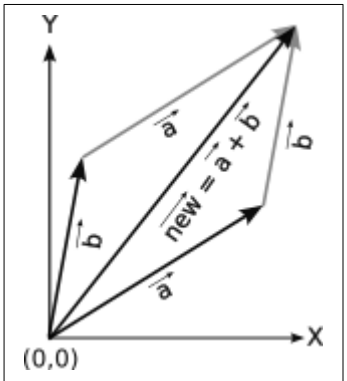
(x_a, y_a) *is the notation for the x , y components of vector a*

Vector Addition

Vectors are useful for calculuating the interaction between two or more forces. Let's say you want to swim across a flowing river. You want to cross the river perpendicular to its flow. As you head out the force of the flowing water carries you downstream. If you continue swimming you eventually will reach the other side, but further down stream perhaps than you intended.

There are two vectors in play here: the force of you swimming perpendicular to the river and the force of the flowing river parallel to the river. If you add these two vectors you can find your actual path across the river.

A vector sum basically is to place the tail of the first vector at the origin and the tail of the second vector at the head of first vector. The resulting vector is the vector from the origin to the head of the second vector. This is the same thing as adding the x components of the two vectors to form the x component of the resulting vector. Do likewise for the y component. That is:



The vector sum of the vectors a and b

$x_{new}^{\rightarrow} = x_a + x_b$

$y_{new}^{\rightarrow} = y_a + y_b$

The resulting vector angle can be found by the formula:

$$angle_{new} = \arctan\left(\frac{y_{new}}{x_{new}}\right)$$

The resulting vector magnitude can be found using the Pythagorean theorem:

$$magnitude_{new} = \sqrt{x_{new}^2 + y_{new}^2}$$

You will sometimes see this written in the more compact vector notation as:

$\vec{a} + \vec{b} = \overrightarrow{new}$

TODO work in the WPI LIB classes for Translation2D (a two-dimensional vector), Transform3D (a three-dimensional vector) and a Rotation3D (yaw, roll, pitch all in rotations). These are like the Pose2D and Pose3D constructs but add a magnitude component (expressed as x, y and z) usually used in robotics for calculating distances using vector addition. WATCH OUT for x being forward and y being side to side.

Odometry

Odometry is the subsystem that keeps track of where the robot is on the field and the direction the robot is pointed by accumulating the small incremental movements of the robot. This location information is useful to automate the movements of the robot to predefined points on the field like a scoring position, a game piece pickup location, etc. Most teams rely on this during the autonomous period to move the robot to precise positions on the field. Advanced teams use this during the teleop to make the robot move smoother and faster than is possible with a human driver.

The next section on kinematics will periodically provide the robot's velocity up or down field ($velocity_y$) the robot's velocity across the field ($velocity_x$) and the robot's rotational velocity ($velocity_{rotation}$). The rotational velocity and even its position can more easily be provided by the robot's gyroscope. We will assume that for the rest of this section.

Movement is simply the distance traveled. This can be calculated from velocity as:

$$movement = velocity \cdot time$$

or a bit more accurately using

$$movement = \frac{velocity_t + velocity_{t-1}}{2} \cdot time_t$$

which is taking the average velocity between the current velocity (velocity at time t) and the velocity of the last measurement (velocity at time t-1) and multiplying that by the duration of time at time t.

The robot movement can be represented as a vector which is a movement of a certain distance in a certain direction. This can be a direction and the magnitude or velocity of the robot and its direction relative to the field frame of reference. It can also be represented as its *x* and *y* components. We then need to sum up all of the movements from time (t) at the beginning (t=0) until the current time (t=n). This is represented as the following:

$$x = \sum_{t=0}^{n} movement_{xt} = \sum_{t=0}^{n} \frac{velocity_{xt-1} + velocity_{xt}}{2} \cdot time_t$$

$$y = \sum_{t=0}^{n} movement_{yt} = \sum_{t=0}^{n} \frac{velocity_{yt-1} + velocity_{tt}}{2} \cdot time_t$$

In general, the smaller the increment of time the more accurate the resulting movement. However, any errors in the velocity measurement will also accumulate. A further problem is that the motion reported by kinematics does not account for wheel slippage or sliding. This error is relatively small for the autonomous period, but it too great for the teleop period without some means of resetting it from time to time (when the robot is at a known position or by using a position reported by the April Tag subsystem.

As a footnote, GPS has been a wonderful technology for locating people in the real world. In general it is not precise enough to locate a robot on a playing field. Today it normally can locate to within a meter or two, so it is not precise enough to locate on a playing field. Some GPS corrective technologies may work in the future for FRC robots, but not today.

As a second footnote, another technique for location measurement is inertial guidance that sums up the incremental movements in three dimensions using the yaw, pitch and roll values supplied by the

gyroscope. This technique is used for missile, aircraft, submarine and quadcopters, but is not known to be used in robotics. It seems like this should be possible, but the gyroscopes may not be able to handle the instantaneous velocity changes caused by collisions. Even so it seems like it would be better than dealing with wheel slips and slides.

Kinematics

TODO reread this to make sure it makes sense. Also check if this is classic kinematics which is somewhat closer to odometry x = x + vt or x x + vt + ½ att. Maybe this just calculates the motion of the robot and odometry keeps track of the little bits (integrates).

In robotics kinematics is the math that calculates the motion of a robot from the rotation of its drive wheels. If all wheels of the robot are in the same direction and speed (have the same vector), the robot moves with the same direction and speed. If the wheels on one side of the robot turn faster than the wheels on the other side, the robot will turn toward the slower turning wheels and its velocity or speed will be roughly the average of the speeds of the two sides. Swerve drives add a little twist to the calculation, as the robot chassis can turn independent of its direction of travel. So why worry about this? It is useful to know where on the field the robot is for path planning and for calculating the power and angle for shooting mechanisms. In general the awareness of field position is the job of odometry discussed in the previous section, but that depends on knowing how the robot is moving given a particular movement command over time.

In general each wheel contributes a directed force, a vector, to move the robot. This vector is usually applied some distance from the center of the robot and hence has some leverage on the robot. The addition of all of the wheel vectors results in a robot movement vector. Differences in wheel vectors result in a robot rotation vector.

Break this down a bit:

- tank drive
  - model as a single wheel on axis of robot
  - forward velocity = average of the two wheel forward velocity

$$rotational\ velocity_{robot} = \frac{velocity_{leftside} - velocity_{rightside}}{2}$$

    - if right > left, rotating toward left with pivot point left of left sid
    - if right > 0 and left = 0, pivoting about the right wheel
    - if right > 0 and left < 0, pivot point between wheels
    - if right == -left, pivoting about the center of the robot
    - need to do reciprocal cases and negative cases to be complete and a bit more logical

TODO Add pictures for tank drive going straight, turning while moving, twisting about a point.

TODO a bit more complicated as the two vectors are defining an arc about an imaginary point . The inner wedge is the radius of robot – distance of left drive wheels to robot center = $r_{leftside}$

The outer wedge is the radius of robot + distance of right drive wheels to robot center = $r_{rightside}$

the height of the inner wedge is the distance traveled by the left wheels = $d_{leftside}$

the height of the outer wedge is the distance traveled by the right wheels = $d_{rightside}$

The inner angle of the wedge can be approximated as follows for small ratios:

$$\tan\theta \simeq \frac{opposite}{adjacent} = \frac{d_{leftside}}{r_{leftside}} = \frac{d_{rightside}}{r_{rightside}}$$

solve for $r_{leftside}$, the inside radius of the turn

$$r_{rightside} = r_{leftside} + width$$

$$d_{leftside} \cdot r_{rightside} = d_{rightside} \cdot r_{leftside} = d_{leftside} \cdot (r_{leftside} + width) = d_{leftside} \cdot r_{leftside} + d_{leftside} \cdot width$$

$$d_{rightside} \cdot r_{leftside} - d_{leftside} \cdot r_{leftside} = d_{leftside} \cdot width$$

$$r_{leftside} \cdot (d_{rightside} - d_{leftside}) = d_{leftside} \cdot width$$

$$r_{leftside} = \frac{d_{leftside} \cdot width}{d_{rightside} - d_{leftside}}$$

$$r_{robot} = r_{leftside} + \frac{width}{2}$$

The deviation angle is the difference of the path actually taken verus the path initially set upon

$$angle_{deviation} = \frac{180 - \theta}{2}$$

now you can solve for the x and y… the robot remains tangential to the turn radius through out the movement, so the robot is turning through the movement. **Θ** is the angle that you turned and the radius of the turn does not change, so the forward vector is pulled perpendicularly to follow the tangent of the arc.

Without turning the x and y are derived as:

$$x_{change} = d_{robot} \sin(heading)$$

$$y_{change} = d_{robot} \cos(heading)$$

with turning x, y and heading are derived as:

$$x_{change} = d_{robot} \sin(heading - angle_{deviation})$$

$$y_{change} = d_{robot} \cos(heading - angle_{deviation})$$

$$heading = heading - angle_{deviation}$$

- West coast drives TODO Subheadings
  - complicates turn calculation because the center of rotation moves depending which wheels support the robot which in turn depends on on robot acceleration
    - acceleration pushes turn center back
    - deceleration pushes turn center forward

TODO Figures showing the two centers of rotation for a turn a curve and twisting at a point. Also point out the the pitch angle from the gyroscope is used to determine where the center of gravity it.

- 2 motor swerve drive
- 4 motor swerve drive
- NOTES
  Swerve drives are controlled by specifying a desired x velocity, y velocity and pose angle of the robot. Odometry is simplified by each module returning the x and y distance traveled and the rotation.

TODO This is where the vector discussion about swerve drive should be, even though it breaks into two parts: how the drive train subystem calculates the drive velocity and steering angle for each swerve module and how kinematics calulates the current postion of the robot based on the steering angle and drive wheel velocity of all of the swerve modules. Similar but different.

Desired movement→drive train→ swerve modules(4)

swerve modules(4)→ measurement measured movement→kinematics→odometry

Vision and April Tags

- TODO supply the name of the major vision software systems. OpenCV or open source compuer vision is the basis for most vision processing system. It allows for picking out or "seeing" particular objects within a field of vision.
- TODO Separate
  - vision processing for "seeing" an object
    - openCV
  - April Tags
    - PhotonVision (open source) or LimeLightOS (both are based on openCV) to find the April Tag, determine its value, calculate the AprilTag to robot pose3D
    - Use of coprocessors like Raspberry Pi or Orange Pi
    - Use of different types of camera
      - Global shutter. Entire image is locked before scanning. There is no skew within the image. Should be more accurate.
      - (normal) shutter. Image is scanned from top to bottom so there may be some skew in a moving image.
  - system for steaming video to the drive station

Odometry has a couple of inherent problems. It cannot accurately account for all robot movement, like when the wheels spin, the robot slides due to a collision, or the robot is airborne (like crossing a cable protector or jumping off the charge station in the 2023 game). There needs to be a reliable periodic way to reset the odometry pose quickly and accurately.

One solution is April Tags and vision processing. The basic idea is that from the vision system can provide information to reset the robot x and y field position as the robot moves about the field. It does this by computing the angles between the robot and the tag to ascertain the robot's position.

Odometry calculates a Pose2D object for the robot. This object has three attributes, that form a vector to locate a robot and its orientation on the field:

- x
- y
- angle

For odometry these attributes are with respect to the field frame of reference, but other poses may not be.

There is a set of configuration data that is used with the vision equations. This information includes:

- Tag information
  - Pose3D: x, y, z, roll, pitch, and yaw relative to the field frame of reference.
  - Actuarial value (its number or identity)
- Camera location on robot
  - Pose3D: x, y, z, roll, pitch, and yaw relative to the robot frame of reference.
- Camera calibration data. The vision system calibrates the camera to be able to map from an image pixel location to an angular measurement. The calibration process is deep within the vision subsystem and is a way to account for the non-linear mapping of a spherical world through sperical lenses onto a flat image detector. Think of this process as breaking the surface of a sphere into a bunch of flat patches that behave linearly, just as a circle can be described as a set of short straight segments. As a general statement, the better a camera is calibrated, the smaller these patches and the better its measurements will be.

This augmented with other data such as

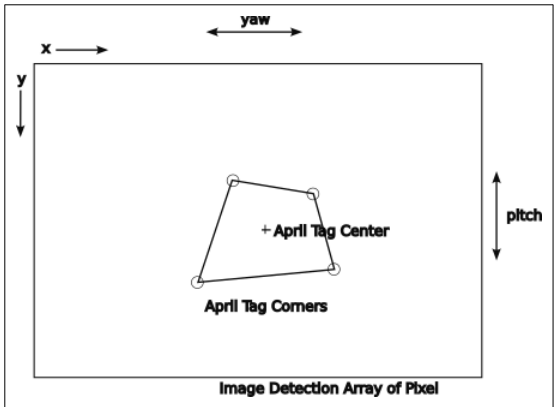
- the robot yaw angle from the gyroscope with respect to the field frame of reference.
- The yaw and pitch angles of the tag with respect to the camera frame of reference.

Since this is the first time you may have encountered roll, pitch and yaw, let's define them. Think of an airplane. Yaw is the horizontal angular direction that the airplane is pointed, like east, northeast, 15°. Pitch is the angle up or down and also an angular measurement. Roll is like banking or the side to side rotational angle.

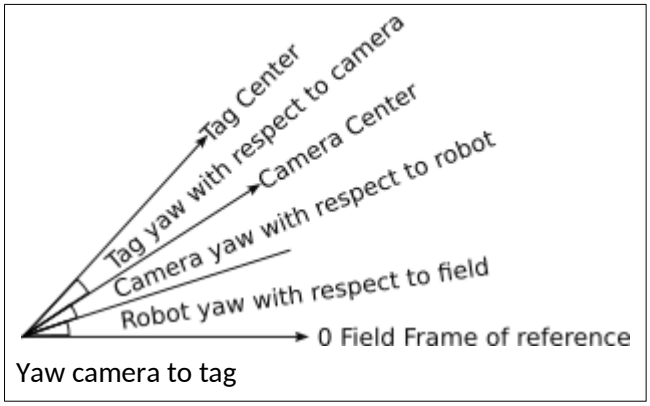
April tags have to work in three-dimensions because the tags are not on the floor, the tags have various mounting angles with respect to the field axes and the robot cameras that "see" them are not on the floor.

The roll angle of the April tags is zero. With careful installation the roll angle of the camera can also be zero. We will work with that assumption through the following calculations.

Let's work through this one step at a time. First the robot vision system "sees" an April tag. It recognizes its "actuarial" value (its number), and its corners of the tag within a camera image in terms of pixels. These pixel addresses are converted to yaw (side-to-side) and pitch (up-down) angles with respect to the robot camera. The camera roll angle (twist) is pretty much assumed to be



Camera frame of reference



Yaw camera to tag

zero, but this assumption depends on a square and secure camera mounting on the robot. From the tag corner angle measurements, the tag center yaw and pitch can be determined through averages.

$$yaw_{cameraToTag} = \frac{yaw_{cameraToTagUpperLeft} + yaw_{cameraToTagUpperRight} + yaw_{cammerToTagLowerLeft} + yaw_{cameraToTagLowerRight}}{4}$$

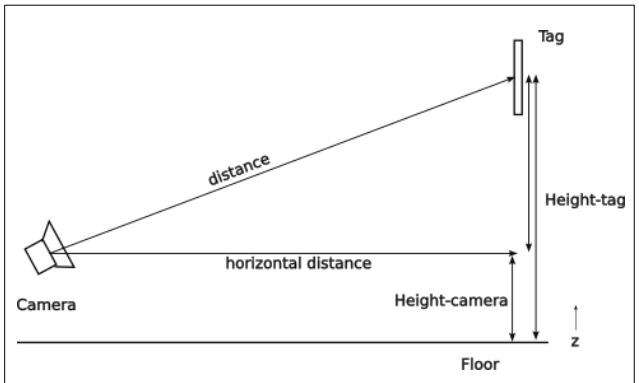
$$pitch_{cameraToTag} = \frac{pitch_{cameraToTagUpperLeft} + pitch_{cameraToTagUpperRight} + pitch_{cammerToTagLowerLeft} + pitch_{cameraToTagLowerRight}}{4}$$

The mounting of the camera with respect to the robot is a basic known piece of information kept in the configuration data as a Pose3D object with its x, y, z, roll, pitch and yaw angles with repect to the robot robot frame of reference.

The height of the April tag above the camera can be found with arithmetic and configuration data for the height of the April tag and the height of the camera with repect to the robot:

$$height_{cameraToTag} = height_{tag} - height_{camera}$$

Now with the pitch of the tag we can find the distance of the camera from the tag:



Camera tag vertical plane

$$\frac{height_{cameraToTag}}{horizontal\,distance_{cameraToTag}} = \tan\left(pitch_{cameraToTag}\right)$$

which can be re-written as follows to solve for the distance between the camera and tag:

$$distance_{cameraToTag} = \frac{height_{cameraToTag}}{\tan\left(pitch_{cameraToTag}\right)}$$

To find the yaw angle of the camera to the tag with respect to the field:

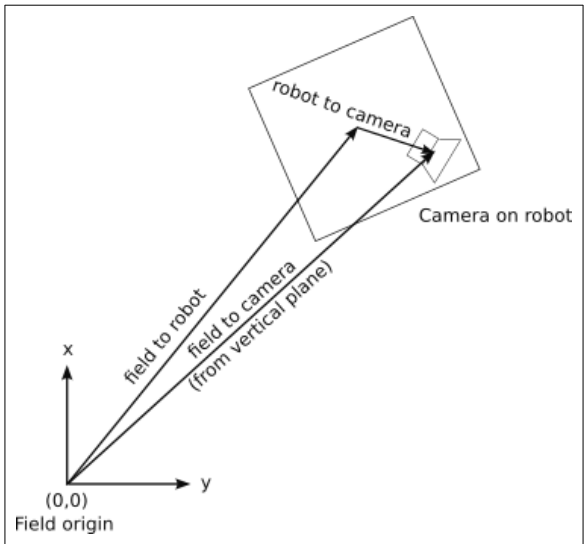
$$yaw_{cameraFieldToTag} = yaw_{fieldToRobot} + yaw_{robotToView} + yaw_{view\,ToTag}$$

From this find the field x and y coordinates of camera using the known field coordinates of the tag (is just a 2D vector addition of the of the field to tag vector and the inverse (flip the direction) of the camera to tag vector):

$$\vec{fieldToCamera} = \vec{fieldToTag} - \vec{cameraToTag}$$

The position of the robot within the field can be found by subtracting the robot so camera vector from the field to camera vector:

The robot to camera vector is in the robot configuration data as the camera x, y and z position relative to the robot center. For this calculation we need only the x and y values. However since the camera can rotate about the robot's center, we do



Field to camera horizontal plane

have to be concerned about the angle of the robot to camera vector with respect to the field. So we have to convert the x and y values into magnitute and angle values, rotate the vector and convert it back to its field based x and y values.

The position of the camera with repect to the field in a 2D horizontal plane is to rotate the vector from the robot center to the camera by the robot yaw and the robot to came yaw:

$$yaw_{fieldToCamera} = yaw_{fieldToRobot} + yaw_{robotToCamera}$$

$$yaw_{robotToCamera} = \arctan\left(\frac{x_{robotToCamera}}{y_{robotToCamera}}\right)$$

$$distance_{robotToCamera} = \sqrt{x^2_{robotToCamera} + y^2_{robotToCamera}}$$

$$x_{fieldToCamera} = x_{fieldToCamera} - distance_{robotToCamera} \cos\left(yaw_{fieldToCamera}\right)$$

$$y_{fieldToCamera} = x_{fieldToCamera} - distance_{robotToCamera} \sin\left(yaw_{fieldToCamera}\right)$$

The foregoing calculations are dependent on the robot yaw angle with respect to the field. This is supplied by the gyroscope device. This device drifts and may be subject to errors induced by collisions. It can handle 200 TODO degrees per second, but a collision can skew a robot 30° in 100 ms or less. This would translate to 300 degrees per second exceeding the specification of the gyroscope.

This impart 30 degrees in a few milliseconds and that exceeds the __ degrees per second limit. Its drift can be on the order of 2° over the course of a match. At twenty feet or 120 inches a 2° error is ±8.38 inches which is significant in most cases.

The vision system can return the three dimensional pose of the camera from April tag simplifying the calculations with the tag's known three dimensional field position (x, y, z) as:

$$\vec{fieldToCamera} = \vec{fieldToTag} + \vec{tagToCamera}$$

$$\vec{fieldToRobot} = \vec{fieldToCamera} - \vec{robotToCamera}$$

The accuracy of this equation is subject to the accuracy of the vision system (and the security of the camera mounting to the robot), but that should be significantly better than the robot yaw angle. Additionally it could also update the robot Pose2D angle which is the same as the robotToTag angle.

$$\vec{robotToTag} = \vec{robotToCamera} - \vec{tagToCamera}$$

$$\angle\, robotToTag = \arctan\left(\frac{x_{robotToTag}}{y_{robotToTag}}\right)$$

Networking Basics

Fix this paragraph TODO. Introduce the topic. Include something about wired ethernet and WiFi and the internet. Not directly Linux related although Linux supported (access point, DHCP, others…) Discuss difference between a WiFi Access Point and a WiFi Modem/router and an ethernet router and a eternet switch.

Linux is known for its networking capabilities. In modern Linux systems, the primary network used is Ethernet. The roboRIO uses wired internet to connect to the radio module and it can be used to interconnect to cameras, co-processors, and to a tethered driver station. Since there is only one Ethernet port on the roboRIO a network switch or router is required to interconnect to the desired number of devices.

Tethering of the drive station may be used in testing the robot at home. Tethering is required for testing the robot at FRC competitions and other places where wireless connections are not allowed. Some robots provide a dedicated port for conveniently connecting the tether. The tether should be long enough to allow the desired robot movement.

The roboRIO uses IPv4 network addresses, just like the rest of the Internet. These addresses are specified in the **dotted quad** or **dotted decimal** notation, which is four groups of numbers separated by a period. Each of the four numbers is between 0 and 255. The first group specifies the network group. To avoid conflicts with the public internet addresses, there are special numbers which are set aside for private use, as shown in the following table.

Range beginning	Range end	Range use
127.0.0.0	127.0.0.255	Local computer
10.0.0.0	10.255.255.255	Corporate private address space
169.254.0.0	169.254.255.255	Private address space used by Microsoft computers
172.22.11.0	172.22.11.255	Private? address assigned by a USB connection to the roboRIO. This is not a "normal" network connection, so may be outside of normal network rules.
192.168.0.0	192.169.255.255	Home or small business private address space

Computers use 127.0.0.1 as a loop back address and is some times knows as home. Most routers uses either the 10.x.x.x or the 192.168.x.x addresses by default. The roboRIO uses the address 10.TE.AM.xx for its use, where TE is the first two (or three(?) for five digit team numbers) digits of the team number and AM is the last two digits of the team number. The IP addresses for Team 4512. are 10.45.13.xx. Common address assignments are shown in the following table:

Device	Low order network address	Network address
Mesh Radio	1	10.TE.AM.1
roboRIO	2	10.TE.AM.2
Driver Station	5	10.TE.AM.5
IP Camera	11	10.TE.AM.11

The open mesh radio reserves the address 1, to it would be 10.45.13.1 for Team 4513. The roboRIO reserves address 2, so it would be 10.45.13.2 for Team 4513.

Network addresses are dynamically assigned by a protocol called DHCP, or the **Dynamic Host Configuration Protocol**. This allows connecting previously unknown devices to a local network without having to manually manage and the network addresses. A minor problem is that the IP address assigned to a device may not be known. You may need a tool like Angry IP scanner to scan the local network for connected devices (like vision coprocessors) to find their address.

The Field Management System does limit the bandwidth available to 4Mb/s to robots during competitions. They also prioritize traffic based on the port used:

- drive station control and status
- network tables
- everything else (video)

The Field Management System also limits what ports can be used during competions. A port is an address within an Ethernet endpoint, which generally supports a single specified service. The FMS allows:

- UDP/TCP 1180 - 1190: Camera Data
- TCP 1735: SmartDashboard
- UDP 1130: DS-to-Robot control data
- UDP 1140: Robot-to-DS status data
- HTTP 80: Camera/web interface
- HTTP 443: Camera/web interface (secure)
- UDP/TCP 554: Real-Time Streaming Protocol for h.264 camera streaming
- UDP/TCP 5800-5810: Team Use
- More information about Ethernet networking and IP addresses can be found on the Internet or in the WPI documentation.

Robot Architecture

The robot architecture is divided into several areas, just as many other systems are. This allows for division of the responsibility of the components. At the highest level the robot breaks down into the following layers:

- **Mechanical** defines the physical robot components from the chassis, drive train, wiring harnesses, subsystem mechanisms, etc. The intent of this architecture is hold the robot together mechanically and survive the hardships of competitions.
- **Hardware** defines the motors and actuators that interact with the mechanical components to make the robot move.
- **Electrical** defines the distribution of power through the robot to power motors and lights.
- **Control** defines the systems that control the robot both from the perspective of the competition field system that provides safety for the robots and participants and the distribution of controls for motors and lights within a robot.
- **Software** is the programs that run on the roboRIO, drive station and field management systems to control the robot.

Robot Mechanical Architecture

Robots are highly varied and original for each team. This individual nature of robots precludes very much discussion of the mechanical architecture other than to refer you to the game rules for a particular year that specify size, height, bumper, and other constraints.

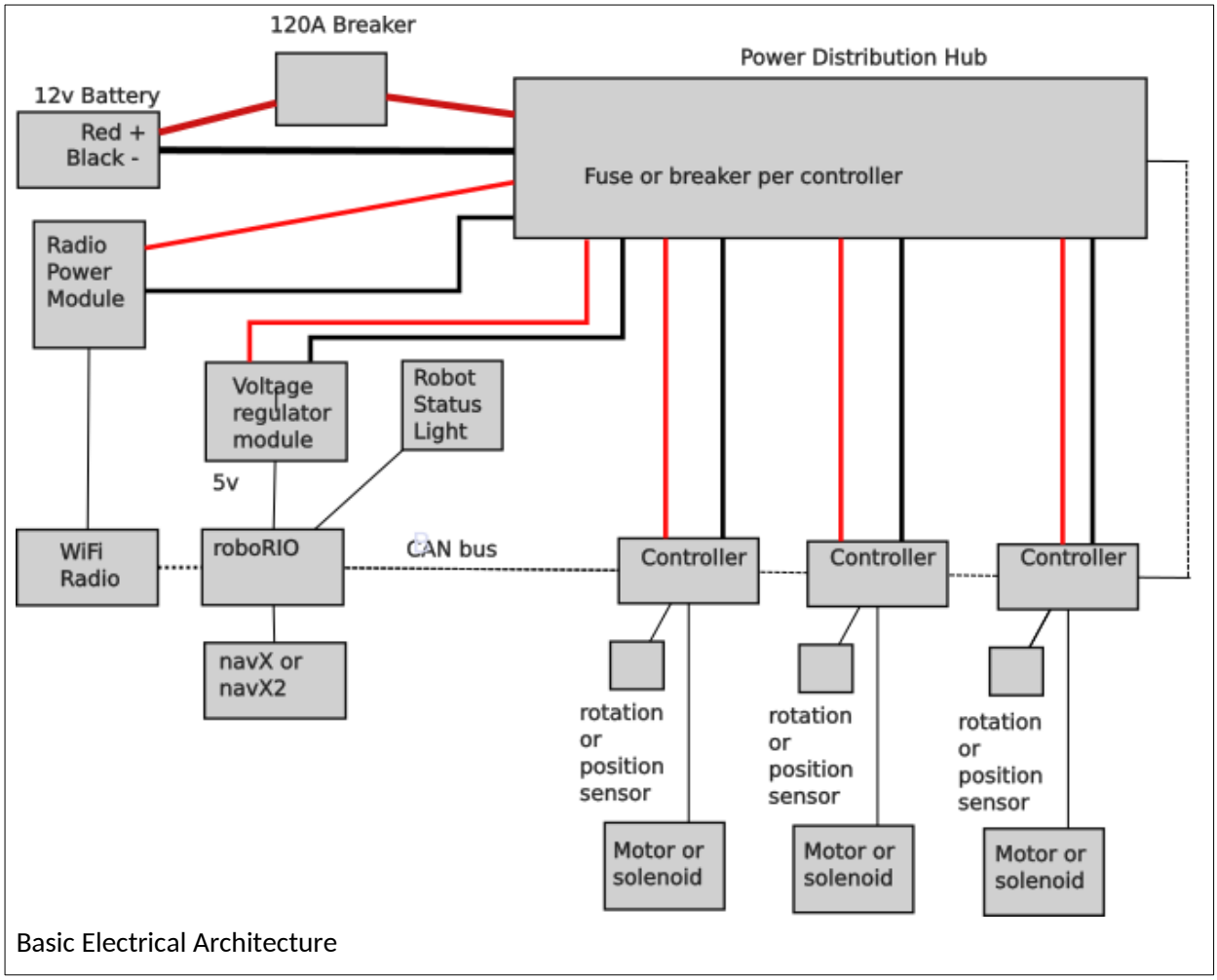
The challenge in a mechanical design is find the winning combination of trade-offs between:

- cost
- weight
- speed
- robustness (ability to survive crashes)
- stability (ability to maintain control)
- functionality
- capabilities of the team

Robots come in many styles, shapes and sizes. All start with a chassis, a frame to which other components are attached. It must be robust to stand up to collisions with other robots and field elements. Bumpers are added to the chassis to help mitigate the effect of collisions, but not as much as you might think. Then there are the other components and hardware that give the robot its functionality and individuality.

Robot Electrical Architecture

The hardware architecture includes the electrical and moving elements (motors, relays, and solenoids) of a robot. A more detailed overview of the hardware components including photographs of those components can be found at https://docs.wpilib.org/en/stable/docs/controls-overviews/control-system-hardware.html. That discussion includes components that are not used in every evolution of the robot and is a great reference source.



Basic Electrical Architecture

Required Hardware Components

FRC requires that all robots have the following components so the robots can compete safely:

Battery

One and only one gel cell **battery** can be used. Some peripheral devices may have independent battery power as long as they meet restrictions as specified in the rules for a given season.

Main Breaker

Main breaker to protect all of the components against accidental shorting. This is a 120 A breaker so it will allow a dangerous current flow which can arc and melt metal. The main supply wires should ALWAYS be treated with respect and care.

Power Distribution

Power distribution hub has breakers to protect individual circuits against over current. Every motor in the robot has its own breaker.

roboRio

roboRIO is the main controller for the robot. It has almost all of the team written code for the robot. It's operating system is also part of the field management system that controls the start of the robot software for the autonomous and teleop periods as well provides a way to shut down the entire robot in the event its operation is deemed to be unsafe by the field judges. Some auxiliary co-processors are allowed in things like vision and navigation subsystems. The r**oboRIO** should ultimately be in control of all co-processors and the devices that they may control.

Robot Status Light

Every robot must have a clearly visible **Robot Status Light (RSL)**. It shows:WiFi Radio

A **WiFi radio** provides communication between the robot and the field management system and through it, the driver station. This radio ensures that only the field management system can communicate to the robots on the field and that it can enforce bandwidth restrictions so that all robots on the field have access to the same amount of bandwidth. The radio should be mounted where it has minimal electrical interference from other components such as motors.

Wifi Radio have been used until the very end of the 2024 competition season. They have been replaced by some other type of radio.

Robot Power Module

A **Robot Power Module (RPM)** specifically to power the **WiFi Radio** using power over Ethernet.

Voltage Regulator

A **voltage regulator** to convert the 12V robot power to 5V power used by some components in the robot.

Drive Train

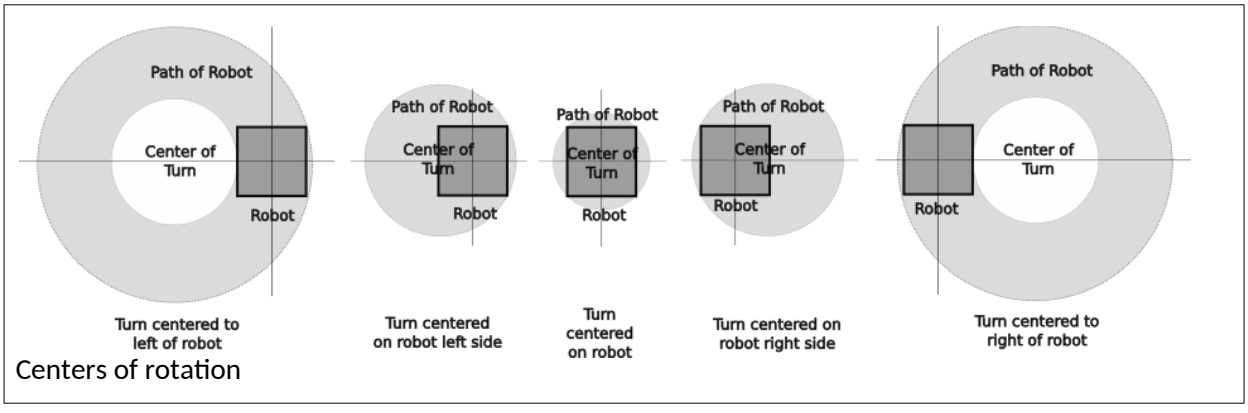
The drive train provides locomotion to the robot and is central to the functionality of the robot. Currently there are four major styles of drive train (in historial order):

- Tank drive
- West coast drive
- Mecanum drve
- Swerve drive

These drives are discussed in more details in the following sections.

Steering a Robot

In general a robot is steered by varying the ration of the speeds of the drive motors on the two sides of a robot will make the robot turn. The center of rotation moves depending on the differences in the rotational speeds. Swerve drive robots have independently steerable drive wheels, but varying the ratios of the drive motors on two sides of the robot will also steer it and more importantly change its rotatation (more on this in the ducussion on swerve drives). The following figure shows different centers of rotation and the path of the robot.



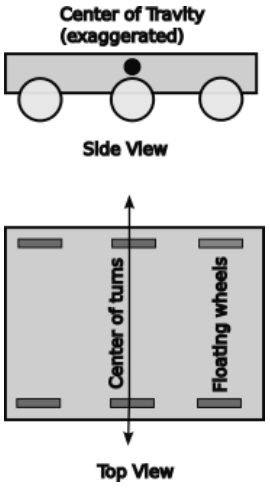
Centers of rotation

From the figure above a more detailed analysis of the movement or the robot and the relationships of the speed of the robot wheel motor to achieve different turn radii and directions.

Center of Rotation Zone	Direction of Rotation	Value of Turn Diameters		Relationship of Turn Diameters
		$D_{leftside}$	$D_{rightside}$	
To left of robot	CW	<0	<0	$D_{leftside} > D_{rightside}$
	CCW	>0	>0	$D_{leftside} < D_{rightside}$
Robot left side	CW	=0	<0	$D_{leftside} > D_{rightside}$
	CCW	=0	>0	$D_{leftside} < D_{rightside}$
Between robot center and left side	CW	>0	<0	$D_{leftside} > D_{rightside}$
	CCW	<0	>0	$D_{leftside} < D_{rightside}$
Center of robot	CW	>0	=<0	$D_{leftside} = D_{rightside}$
	CCW	<0	>0	$D_{leftside} = D_{rightside}$
Between robot center and right side	CW	>0	<0	$D_{leftside} > D_{rightside}$
	CCW	<0	>0	$D_{leftside} < D_{rightside}$
Robot right side	CW	>0	=0	$D_{leftside} > D_{rightside}$
	CCW	<0	=0	$D_{leftside} < D_{rightside}$
To right of robot	CW	>0	>0	$D_{leftside} > D_{rightside}$
	CCW	<0	<0	$D_{leftside} < D_{rightside}$

Tank Drive or Differential Drive

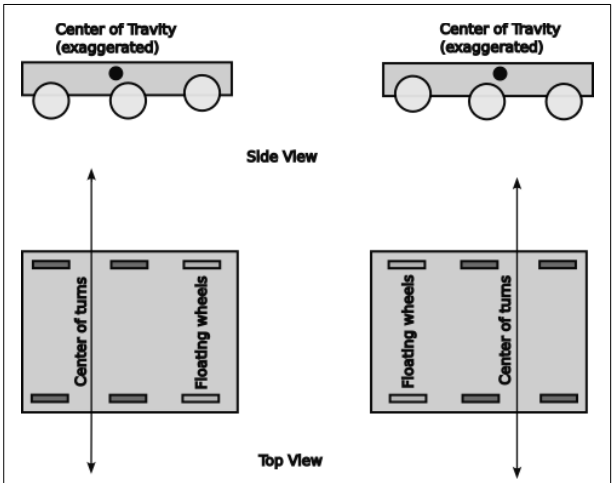
A tank drive robot is driven by two sets of wheels, one on each side. Most tank drives only have two wheels on each side, but may have more. Steering is performed by changing theis speed of one side versus the other differentially, leading to its other name: differential drive. Tank drive robots are the simplest form of FRC robot.



Tank Drive Basics

West coast drive

A west coast drive is a variation of the tank drive where the center wheels on each side are set about an 1/8” lower than the outside wheels. This means that the balance point of the robot can change depending on past momentum shifts. Accelleration can move the balance point back, while deceleration will move the balance point forward. Each side has 3 or 4 wheels. Like a tank drive a west coast drive is steered differentially. The difference is that it has less turning resistance because it does not drag the outer wheels if at all or as much as a tank drive. Another advantage is that it effectively has a shorter wheel base for tighter turns while having the stability of a longer wheel base. The outer wheels may be free turning casters or balls to reduce their drag when the robot is turning.

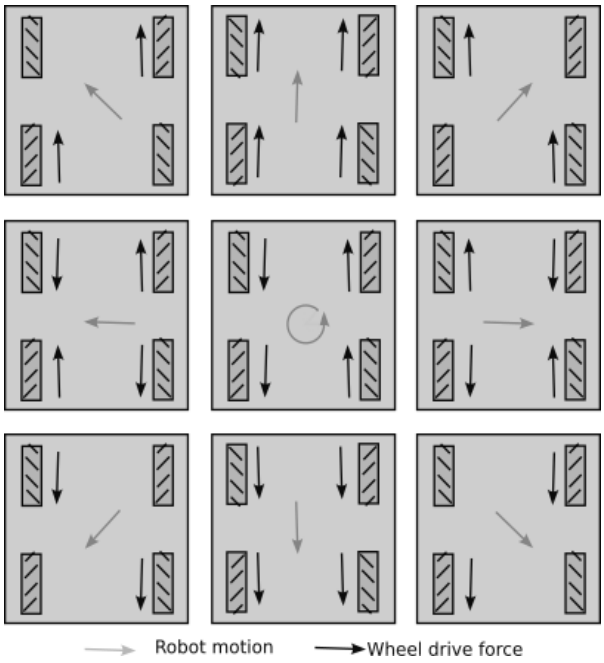


West Coast drive basics

Mecanum drve

Mecanum drive uses four special wheels each with an independent drive. Each wheel has rollers attached at 45° to the direction of rotation. As the wheel turns it applies a force perpendicular to the orientation of the roller. By controlling the relative speeds of the motors the robot can be moved in any direction or turn with any center point. It offers the flexibility of a swerve drive with less motors. However the option of tire tread is limited to the roller so it has less grip on the competition carpet.

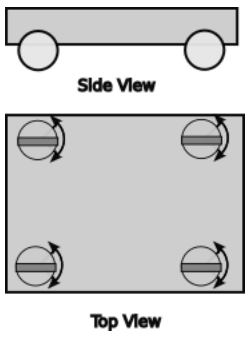
Uranus Omni Directional Robot
by Gwpcmu CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=11440618



Mecanum Drive basic movements

Swerve Drive

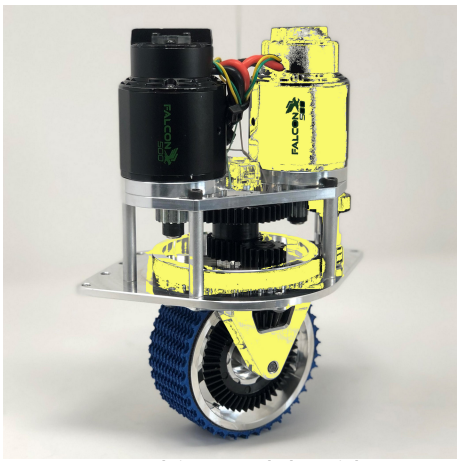
Swerve drive uses two or more swerve drive modules to power the robot. Each module is like a caster using one motor to steer the wheel and another motor to drive the wheel. The center of rotation for turns can be anywhere. Robots with a swerve drive can spin or twist while moving forward, providing for more fluid movement than possible with a tank or west coast drive. It has various wheel, motor and gearing options to suit the requirements of each team.

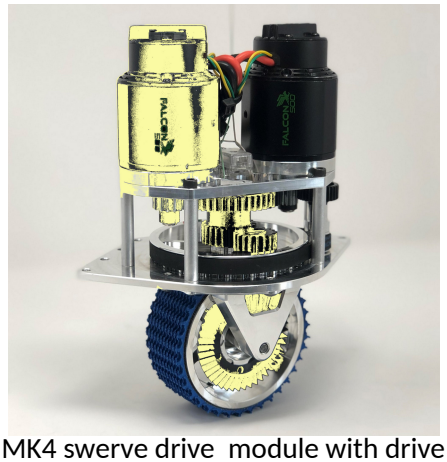


Swerve Drive Basics

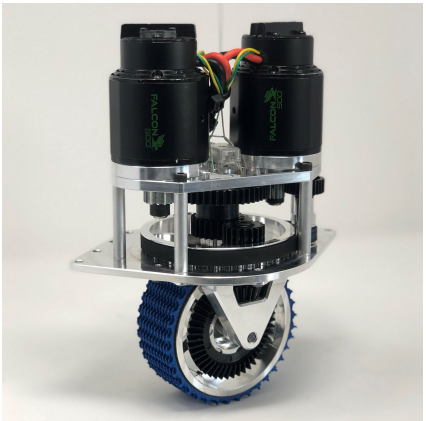
The swerve drivetrain uses four swerve drive modules. Each module has a drive motor connected to a wheel and a steering motor to change the direction of the drive wheel. A swerve drive from Swerve Drive Specialties looks like the photograph to the right.

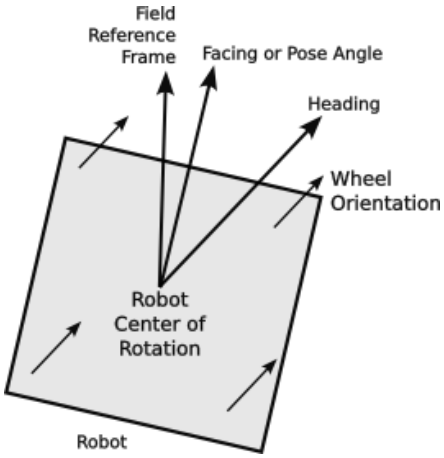
The drive transmission components are highlighted in yellow in the figure to the left. The drive motor is on the left hand side. It drives a gear train which sends power down a shaft along side of the wheel to engage a ring gear with a bevel gear. The intermediate gears are on the outside of the inner axle for the wheel axel assembly. This turns the wheel in the horizontal axis and is independent of where the wheel is pointing.


MK4 swerve drive module with steering gear train highlighted in yellow.


MK4 swerve drive module with drive gear train highlighted in yellow.

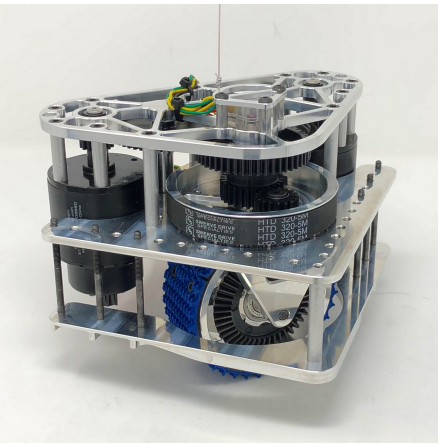
The steering motor is on the right side. It transmits power to a belt that turns the wheel axle assembly about a vertical axis. The position of the wheel is monitored by a sensor at the top of the wheel axle assembly. The vertical axel has bearings in both the top and bottom mounting plates and is inside of the intermediate drive gears.


Swerve Drive Specialties MK4 swerve drive module



Swerve Drive Angles

FRC 4513 uses a variation of the MK4, called the MK4i, which inverts the motors for a lower profile. It is shown at the left



MK4i swerve drive modules with the motors inverted from the Mark 4 modules.

Optional Restricted Hardware Components

Optional components can be used as needed for particular robot designs with some restrictions on the number and supplier:

- Only certain motors can be used and they cannot be modified and the total number is restricted.
- Only certain motor controllers can be used (Talon SRX, Victor SPX, Spark Max…)
- Some motors with integrated speed controllers may be used (Falcon 500, …)
- Pneumatics may be used (as they are in the show bot), but
    - there has to be a **pneumatic hub** which provides electrical controls to the other pneumatic components either directly or via CAN bus commands.
    - there can be only one **compressor** and it must be connected to the pneumatic hub.
    - there can be a number of solenoids to release pressure to particular pneumatic devices.
    - it must have a **pressure sensor** on the pressure holding tank connected to the **pneumatic hub** to prevent the system from building up dangerous pressures.
- **Cameras** can be used to send video back to the drive station as long as the image bandwidth is restricted to what the roboRIO will allow.

Optional Unrestricted Hardware Components

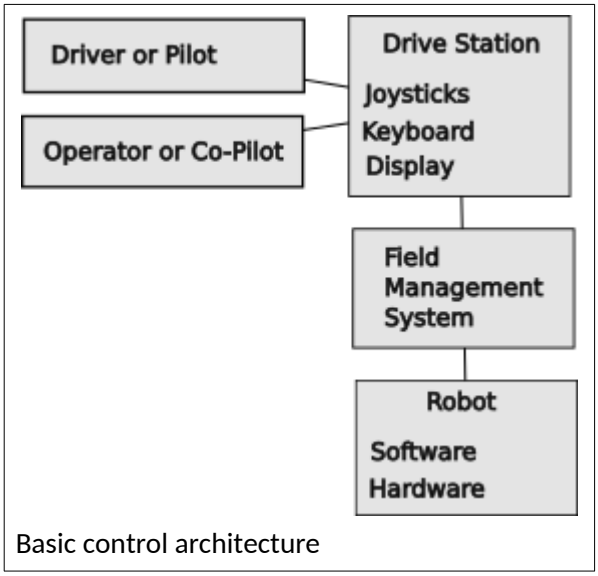
Some components can be used in robots without restriction:

- An Ethernet router or switch to distribute Ethernet to various components within the robot like cameras.
- Limit switches and sensors on elevators, arms, and other moving devices within the robot.
- An integrated navigation subsystem such as the **navX** or **navX2** which connects directly to the roboRIO.
- An independent navigation subsystem such as the **Pigeon** which connects via the CAN bus.

- Using a **CANivore** to control a separate CAN FD bus to allow faster control communication with drive motors to improve their responsiveness and the accuracy of odometry data.
- There can be co-processor devices to assist the roboRIO. These have been used for vision processing, providing inertial guidance information and tighter motor speed controllers that include extra functionality like PID (Proportional Integrated Differential), trapezoidal motion controller or Motion Magic.
- Vision systems are co-processors that have cameras and process images to locate targets, April tags or game pieces.
- Simple vision systems to stream a camera image over the interconnected ethernet to the drive station or a dedicated vision processor.

Robot Control Architecture

Control of the robot is central to robot competition to ensure the safety of the the robots, the competitors and other participants. FRC rules maintain that the match judges have ultimate control over the robot and may deactivate it at any time that they deem that its continued operation is a danger to other robots or participants. To do this the Field Management System or FMS inserts control between the driver station and the robot, so that they can intercept any command at any time. This position also allows it to control the periods of the competition: pre-match, autonomous, teleop, and post-match.



Basic control architecture

- In pre-match they allow the robot to be connected and powered up. The drive teams can communicate with their robots to ensure that communications are established and to select the particular autonomous routine they they want to perform.
- In the autonomous period, the FMS invokes the autonomous method for every robot on the field. This makes all robots start at the same time. The autonomous period continues until it is done for a time determined by the game rules (e.g., 15 seconds).
- After completion of the autonomous period, the teleop period begins and the FMS allows communications from the drive stations allowing the drive teams to fully control the operations of the robot. This period last for a time determined by the game rules (e.g, 130 seconds).
- After completion of the teleop period, power and control is removed from the robot so that the teams can safely remove their robots from the field.

The **drive station** where one or more humans interact with joysticks and other controls to send commands to the robot. They also monitor performance of the robot and take corrective action as necessary. The **driver or pilot** controls the basic movement of the robot about the field. Most teams use a second human called the **operator or co-pilot** who is responsible for the auxiliary functions of the robot.

The robot commands from the **drive station** are required to go though the **field management system**. This system does a couple of things:

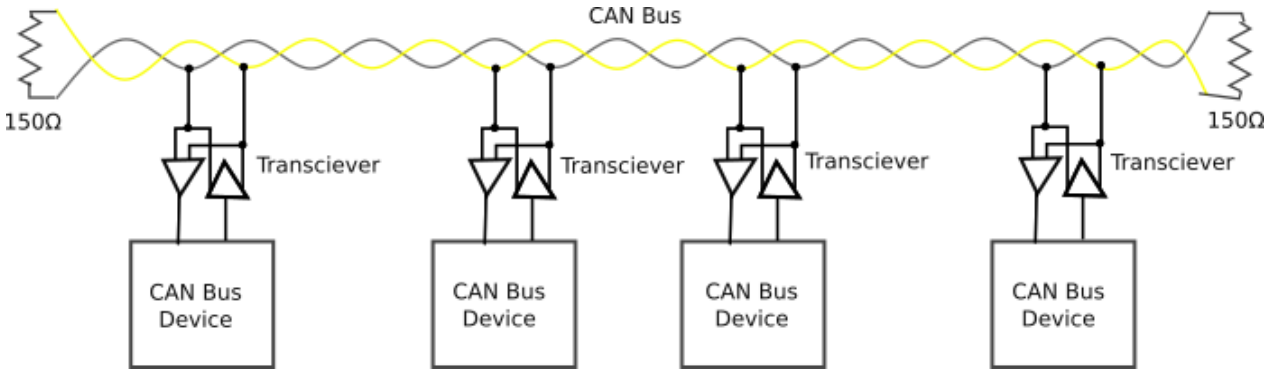
- starts all competing robots at the same time.

- controls the starting of the autonomous period.
- controls the starting of the teleop period and the enabling of the drive stations.
- stops all robots at the end of the teleop period.
- disables any robot that is exhibiting unsafe operation

The **robot** is required to comply with a set of requirements to manage the safety of the robots and their accompanying humans at all times. This includes restrictions on the **hardware** components that can be used as well as the specific way that the controlling **software** is written. More details about this can be found in the robot rules for a given season.

The FMS also controls access over the WiFi network. The drive stations are tied to the FMS over a hardwired Ethernet connections. The FMS then sends commands from a team's drive stations to the team's robot using the team number as an identifier. Some encryption is used to prevent additional control stations from controlling the robot. Because of this encryption, the robot WiFi access point must be reprogrammed for each competition.

CAN Bus



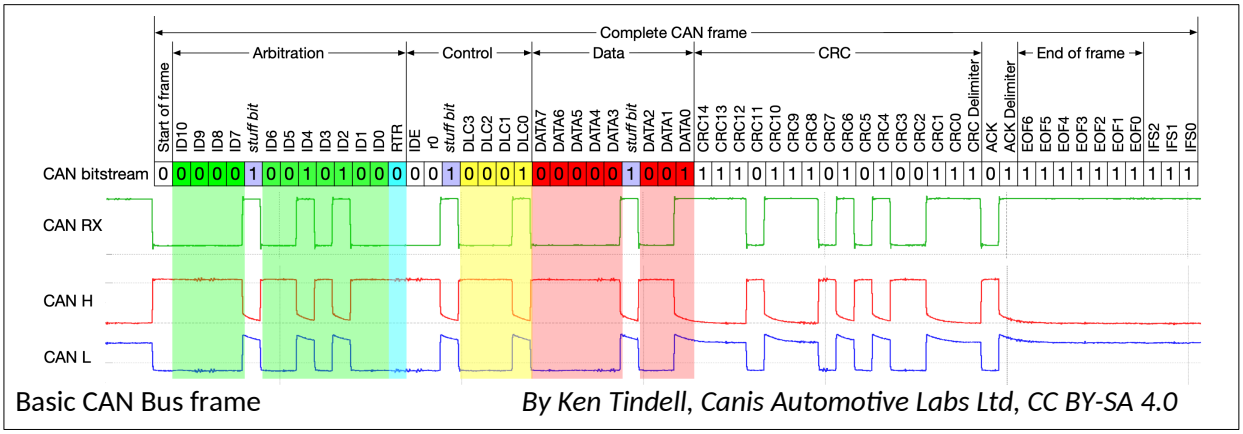
General CAN Bus architecture

The roboRIO communicates with many of the hardware components over a **C**ontroller **A**rea **N**etwork, more commonly know as the CAN bus. This was initially designed to be a control network for vehicles and has been adopted as a robot communication bus. A bus is a signaling network architecture that can have many senders and receivers attached to a single transmission line. The CAN bus is wired as a "daisy chain," where by the each component is a link in a chain and there is a CAN bus input and CAN bus output on every device. The CAN bus is a "transmission line" which is an electrical system for sending signals without loss or distortion. This type of transmission line uses uses a twisted pair of wires "terminated" at its ends to prevent reflections or echos of signals. The CAN bus termination is a 120**Ω** resistor which "eats" the signal rather than reflecting it. This termination resistor is built into the roboRIO and the Power Distribution Panel. If these two devices are at the ends of the CAN bus, they can terminate the bus when the build in terminating resistor is enabled. If these devices are elsewhere in the daisy chain, their built in resistor must be disabled and a terminating resistor must be added to the actual end(s) of the daisy chain.

In FRC robots the twisted pair is typically a green and yellow wire twisted together. One wire is called CAN high or CAN H and the other wire is called CAN low or CAN L. It is important to connect yellow wires to yellow and green to green to keep the polarity straight. The CAN bus signal is differential signal which uses opposite polarity to communicate digital ones and zeros.

Some features of the CAN bus:

- It is a serial protocol meaning that messages are sent as a frame containing a number of bits and each frame is sent one at a time

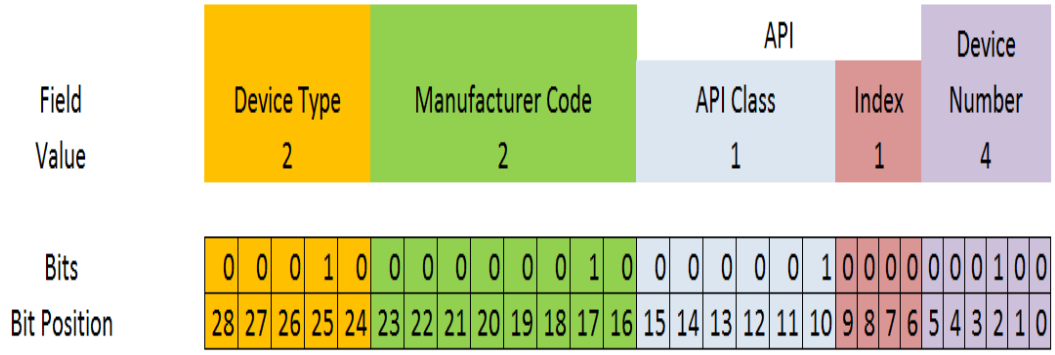


Basic CAN Bus frame *By Ken Tindell, Canis Automotive Labs Ltd, CC BY-SA 4.0*

- The protocol allows multiple masters (nodes which can initiate a data transfer). This allows bi-directional data transfers.
- It has a bus architecture, i.e., it uses a daisy chain with extremely short stubs.
- The CAN bus needs to have a single termination resistor at the two ends of the daisy chain. This is to prevent reflections on the bus which degrade its performance.
- Bits are sent using a differential voltage across a pair of wires. One wire is called CAN H and the other is called CAN L. A logical “1” is passive, not actively driven, with a low voltage on both the CAN H and CAN L leads. A logical “0” actively drives a voltage on both CAN H and CAN L leads. The “0” can overpower a “1.”
- This differential voltage is fairly immune to electrical noise from motors, because motor noise induces a similar voltage into both leads equally and there is very little differential component.
- Collisions are detected when a node sends a non-dominant “1” value and detects a dominant “0” value. Since the identifier is the first data sent in the protocol, this gives priority to lower identifier values.
- The CAN data frame has 44 bits of overhead before bit stuffing and may transfer up to 8 bytes of data.
- The protocol uses bit stuffing to aid in message synchronization by adding a passive “1” stuff bit whenever there are 5 active “0” bits sent in a row. Up to 25 stuff bits may be sent in a CAN frame with 8 data bytes. This is a maximum of 133 bits for a frame carrying 8 bytes. The base rate for the CAN bus is 1Mbps or 1 million bits per second. This works out to 1000 messages every 20ms period of the robot control loop.
- The Data portion of the frame is further encoded using the FRC specification as described in https://docs.wpilib.org/en/stable/docs/software/can-devices/can-addressing.html This leaves 34 bits for devices to use. [Not verified]

Example

Speed Control Mode Disable from Luminary Micro Jaguar Speed Controller (dev # 4)



Example of coding of a CAN Bus command carried in the CAN bus data payload

This appears to be using the extended CAN bus encoding….need to do more research in this.

- In the above example:
  - Device type is the type of device
  - Manufacturer code reflects the company who made the device
  - The API is the message identifier, it has a class for certain controls and an index for individual parameters within the API.
  - The device number is the number of a device of a particular type. This should default to zero, 0x3F may be reserved for device specific broadcast messages.
- Actuators that control motors must ensure that the frame originated from the roboRIO.
- A Universal Heartbeat message is provided by the roboRIO. It used to enable motors. It is normally sent every 20ms. If this message is not seen for 100ms, controllers should assume that the system is disabled. The Universal Heartbeat is encoded as follows:

Description	Byte	Width (bits)
Match time (seconds)	8	8
Match number	6-7	10
Replay number	6	6
Red alliance	5	1
Enabled	5	1
Autonomous mode	5	1
Test mode	5	1
System watchdog[1]	5	1

1 A watch dog is a system which needs to be probed periodically. If the probe is not received, the device should go into a safe mode of operation, like disabling the controller.

Description	Byte	Width (bits)
Tournament type	5	3
Time of day (year)	4	6
Time of day (month)	3-4	4
Time of day (day)	3	5
Time of day (seconds)	2-3	6
Time of day (minutes)	1-2	6
Time of day (hours)	1	5

TODO differences between CAN and CAN FD especially bit rate and number of bits. CAN and CAN FD buses are not directly compatible, so if you want to use CAN FD, you must have both the CAN bus and a separate CAN FD bus. Some devices can handle either protocol. The roboRIO and the Power Distribution Panel can only use the CAN bus and cannot take advantage of the CAN FD bus. Since these two components are required in FRC robots you have to support the CAN FD bus even if only for these two components.

Robot Software Architecture

TODO Rework these paragraphs… mixed in from elsewhere

Robot software is a complex undertaking. There are many pieces and perspectives on the software and its development process. This section attempts to bring all of the pieces together. This section has a lot of details. Not every team requires everything, but this section provides some information for teams desiring understanding of how things work to improve their software and development practices.

The software for a robot can be overwhelming at first, however, by breaking its functionality down into smaller and smaller pieces, it becomes easy to understand and maintain. At the highest level each robot has software that controls various hardware elements on the robot.

In a nutshell, a robot consists of a set of subsystems. Each subsystem controls an aspect of the robot, say an arm, an elevator, a drive train, a climbing mechanism, a throwing mechanism. A subsystem can only do one thing at a time. An elevator can go up or it can go down. It cannot go up and down at the same time with meaning full results. A general way to think about it is that a subsystem controls a single motor. This can be extended to orchestrate a set of subordinate subsystems. The drive train for a tank drive can control two motors. It can go forward, backward, turn to the right, turn to the left, spin right or spin left. In this case the drive train subsystem is simplifying the commands necessary to move the robot by handling the interactions between the two motors. The drive train for a swerve drive is much more complex management of the swerve modules, but it has similar commands to the tank drive, except that it can drive forward in any direction.

Software Organization

TODO Rework these paragraphs… mixed in from elsewhere

- This step is crucial
- form follows function
- directory tree follows functionality
- WPILIB suggests a single folder approach with robot builder
- Spectrum Team 3847 suggests a better, more functionally reflective approach
- Team 4913 took this a step further with more refinements.

- Folder per subroutine
- Separated drive train from accessories

Core Robot Functionality

Blah blah blah TODO to fill in detail.

Robot Functionality

TODO fill out this section better… this may be limited to overall coordination between subsystems

- Subsystems
- Commands
- Libraries

Drive Train

Blah blah blah TODO to fill in detail.

Mechanisms

Blah blah blah TODO to fill in detail.

Libraries

Libraries allow teams to use software written and debugged by others. This save development time and allows use of complex computations without having to understand how everything works. This is like being able to drive a care without knowing how an engine or the CAN bus works. Can you do more if you understand more. Certainly the answer is yes. But is it necessary to understand everything, definitely no.

Java Libraries

Java has libraries used for general functions like handling strings and some mathmatic objects. These libraries help with vector addition (2D translations and 3D transforms).

Wooster Polytechnic Institute Library for FIRST Robotics or WPI LIB

Programmers use libraries of previously coded and debugged software so they do not have to recreate all of that functionality themselves. Java comes with libraries for basic input and output functions as well as mathematic functions for square roots and trigonometry. Wooster Polytechnic Institute has written a library (WPILIB) for the basic functions of a robot so we don't have to write that code. Vendors have added code to that library to interface to the hardware components, again saving us the time of figuring out how these subsystems work at a low level and writing code to control them..

WPILIB defines a robot as a set of subsystems. Each subsystem is largely independent of other subsystems. The majority of the initial programming is to select what software subsystems your robot needs based on the design of the robot. Once you have the subsystems in place, there is some mapping between joystick controls to the commands that request some action from a subsystem. From there you add code to reflect how you want to change the robot or how you specifically want it to behave.

FRC Programming WPILib Docs: https://docs.wpilib.org.

Team Libraries

Individual teams have developed libraries of their own, such as:

- Team ___ Advantage Kit for logging everything replaying the logs as a simulation.

- Advantage Scope for visualizing data. They convert a string of numbers into a graph so that you can see just when voltage spikes or drop outs occur. They also do 2D and 3D visualizations of the robot pose, so you can see how a robot moves through a practice session or even replay a match. This capability has proven useful to try out robot maneuvers while the robot is on blocks, however it may be limited to cases where the gyroscope is not depended upon.
- Spectrum Team 3847 extensions and organization of software modules.

Looking at other teams repositories is a good way to get ideas for your robot and for examples of how others write their robot code.

Auxilliary CoProcessors

Blah blah blah TODO to fill in detail.

Software File Structure

Blah blah blah TODO to fill in detail and introduce this section

Robot Project Directory

The directory hierarchy to a specific robot is **/root/project/robot**

The **root** is the root directory for all robot projects of Team 4513.

A **project** directory in the root directory contains the code for a robot competition season or for special robots like a second robot for a season or for the show bot.

The **robot** directory contains:

- **Main.java** just kick starts robot.java
- **Robot.java** instantiate all subsystems and provide methods for initializing the robot, running periodic commands
- **RobotConfig.java** configuration parameter for robot components. Thinks like the CAN bus identifer of every controller, input ports for limit switches, analog sensors, and pulse width modulator for LED controller.
- **RobotTelemetry.java** creates instances of subsystem telementry objects and maps information known to the robot to specific tabs in the shuffleboard display at the driver station.
- Various subsystem directories to control individual subsystems.

Subsystem Code File Organization

The WPI library dictates the structure of the robot code to some extent. Unfortunately they could do better than just dump all of the files into a single directory as they do with Robot Builder. FRC 4513 has adopted the changes to the file structure used by the Spectrum FRC Team 3847 to better organize the code. They used a directory for each subsystem to keep the code separate and modular. In each subsystem directory are a set of files for specific control of that subsystem, but those files fit into a design pattern which applies to all or most subsystems. This structure makes it easier to reuse, add, delete and copy subsystems from a robot as needed by each season's particular challenge.

A subsystem (XXX) may have the following files (depending on the particular subsystem):

Template	Example name	Use
XXXSubSys.java	ElevatorSubSys.java	Contains the class definition for the XXX subsystem. This class is normally instantiated in Robot.java file.
XXXCommands.java	ElevatorCommands.java	For subsystems with a small number of small commands, all of the commands may be contained in a single file
commands/yyy.java	commands/ ElevatorStow.java	For subsystems with a large number of commands, or larger more complex commands, individual commands are contained in separate files within a command directory.
XXXConfig.java	ElevatorConfig.java	Contains the configuration data for the XXX subsystem. Hard coding configuration parameters in other files is discouraged.
XXXMotorConfig.java	ElevatorFalconConfig.java	Contains configuration data that is common for a particular type of motor, but specific to this instance.
XXXTelemetry.java	ElevatorTelemetry.java	Defines the class for the XXX subsystem data attributes used in drive station suffleboard and logging.
XXXPeriodic.java	ElevatorPeriodic.java	For subsystems that are more modal than command driven, there may be a periodic method that is called periodically by the main robot loop.

File Naming Conventions

The type of a file is indicated by a file extension. File extensions in Linux are a convention to communicate to human programmers rather than windows where extensions are used to select programs for operating with particular file types. A file extensions is a period and few letters at the end of the file name. Some common extensions are:

.class compiled Java byte codes defining a class.

.cnf a configuration file, usually plain text.

.conf a configuration file, usually plain text.

.dat a data file, usually binary data

.java Java source file.

.jpg a graphic file in the joint picture group format.

.json general JSON formatted file. JSON is based on JavaScript Object Notation which is loosely based on the JavaScript syntax for integers, floating point numbers, strings, arrays (lists of values) and dictionaries (attribute-value pairs).

.md a text file in the mark down format usually in the GitHub version of mark down. This allows for simple formatting of a plain text file.

.path JSON formatted file used by the PathPlanner.

.pdf a stand alone document

.png a graphic file in the portable network graphics format.

.sh a Linux shell program that is also plain text. These days it is usually a bash shell program. Check the first line for program that will execute the file.

.txt a plain text file. This is sometimes used to comment out a whole javascript file

.zip a compressed file made by the zip utility.

Hidden Files

Files in Linux and Macintosh may be hidden. On Linux such files start with a period. On the Macitosh hidden files normally cannot be viewed. When copying whole directories between systems, these files will also be copied and may not be useful to the target system. They may contain things like a thumbnail of an image file used by the operation system to display when listing the files.

Example File Structure

The beauty of Spectrum Team 3847 file structure is that it mirrors the subsystem architecture allowing for compartmentalizing each component. This makes it easy to add and drop subsystems as changes to the robot require it. Comments about individual files is highlighted.

```
~/dev/4513/Robot2023SwerveVer03
├── 3rd Part Library Links.txt
├── Autonomous.png      image of paths for autonomous period for alliance consultations
├── bin                 directory for compiled byte codes
│   └── ...
├── build.gradle        build configuration file
├── config.json
├── gradle              directory of gradle files
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── networktables.json
├── pathplanner
│   └── data
│       ├── flutter_assets
│       │   ├── AssetManifest.json
│       │   ├── FontManifest.json
│       │   ├── fonts
│       │   │   └── MaterialIcons-Regular.otf
│       │   ├── images
│       │   │   ├── field22.png
│       │   │   ├── field23.png
│       │   │   └── icon.png
│       │   ├── NOTICES.Z
│       │   ├── packages
│       │   │   ├── cupertino_icons
│       │   │   │   └── assets
│       │   │   │       └── CupertinoIcons.ttf
│       │   │   └── window_manager
│       │   │       └── images
│       │   │           ├── ic_chrome_close.png
│       │   │           ├── ic_chrome_maximize.png
│       │   │           ├── ic_chrome_minimize.png
│       │   │           └── ic_chrome_unmaximize.png
```

```
│       │   └── shaders
│       │       └── ink_sparkle.frag
│       └── icudtl.dat
├── README.md                          Markdown file for github repositry
├── settings.gradle
├── shuffleboard.json                  configuration for shuffleboard
├── src                                source code directory
│   ├── main                           the real source code directory, ill named
│   │   ├── deploy                     files to be downloaded to roboRIO without changes
│   │   │   ├── example.txt            description of files in this directory
│   │   │   └── pathplanner            directory for path planner files
│   │   │       ├── ...
│   │   └── java                       source code directory for Java code
│   │       └── frc                    source code directory for FRC Java code
│   │           ├── lib                source code directory for WPILIP Java library code
│   │           │   ├── ...            usually one directory per subsystem
│   │           ├── Rmath.java         source code for 4513 robot math library code
│   │           └── robot              source code for 4513 robot
│   │               ├── arm            usually one directory per subsystem, arm is typical
│   │               │   ├── ArmConfig.java        arm subsystem configuration constants
│   │               │   ├── ArmSRXMotorConfig.java
│   │               │   ├── ArmSubSys.java       arm subsystem Class definition code
│   │               │   ├── ArmTelemetry.java    arm subsystem telemetry code
│   │               │   └── commands             arm subsystem commands
│   │               │       ├── ArmCalibrateCmd.txt      a source .txt is a comment
│   │               │       ├── ArmCmds.java
│   │               │       ├── ArmDriveForSecondsCmd.java
│   │               │       ├── ArmHoldPositionCmd.txt
│   │               │       ├── ArmReleaseCmd.txt
│   │               │       └── ArmToStowPosCmd.java
│   │               ├── auto             source code for the autonomous period
│   │               │   ├── ...
│   │               ├── ...              more subsystem directories
│   │               ├── Main.java        traditional entry point, transfers control to Robot.java
│   │               ├── RobotConfig.java
│   │               ├── Robot.java
│   │               ├── RobotTelemetry.java
│   │               └── trajectories
│   │                   ├── commands
│   │                   │   ├── FollowOnTheFlyPath.java
│   │                   │   ├── FollowTrajectoryCmd.java
│   │                   │   ├── GeneratePathForScoring.java
│   │                   │   ├── PathBuilderEstimatedPose.java
│   │                   │   ├── PathBuilder.java
│   │                   │   └── TrajectoriesCmds.java
│   │                   ├── README.md
│   │                   ├── TrajectoriesConfig.java
│   │                   └── Trajectories.java
│   └── shuffleboard.json              another suffleboard configuration file
├── sysid_data20230225-174837.json     sysid files contain performance data about robot
├── sysid_data20230225-175221.json
├── sysid_data20230225-182314.json
├── sysid.json
```

```
├── SysID Notes.txt
├── vendordeps                          directory of vendor library files
│   ├── NavX.json
│   ├── PathplannerLib.json
│   ├── Phoenix.json
│   ├── PhoenixProAnd5.json
│   ├── PhoenixPro.json
│   ├── REVLib.json
│   └── WPILibNewCommands.json
├── WPILib-License.md
└── Xbox Ctrlrs Mapping.vsdx
```

Fundamental Robot Design Patterns

Fundamental to working with a complex system is the ability to break the system down into manageable pieces that are self-contained. Manageable means that the modules are easy to understand in all phases of development: writing, debugging, maintaining, and extending. The self-contained aspect means that changes to one module do not affect other modules. Yes there are some dependencies within the modules of a group, but one group does not affect other groups. In robot software, functionality is broken into subsystems. Each subsystem controls one aspect of motion, like a drive train, elevator, arm, shooter or intake. Each subsystem is independent of each other, so that the code that controls an arm does not affect the shooter. Each subsystem is also controlled by a set of commands. An elevator may have a commands to raise, lower, go to a stow position, gonto a shooting position, etc.

At the robot level, efforts by the various subsystems may be coordinated. A command to shoot may be conditioned on a particular robot pose, an elevator in a particular level, an arm at a certain angle, the shooter spun up to speed. This allows the various subsystems to work simultaneously toward the goal of shooting a game piece, but the piece is not shot until all of the conditions are met.

Design patterns are a way for a software developer to reduce the overall complexity of a system. Rather than “reinvent the wheel” each time, the developer uses a familiar pattern over and over to accomplish the task at hand. Doing it the same way each time makes it much faster to understand how a given subsystem works.

At the fundamental core of robot software are two design patterns. A design pattern is a design that is repeated over and over and in so doing it makes the overall software simpler. The WPI Lib encourages these design patterns. One is the use of subsystems to decompose system functionality into smaller and smaller pieces. The other is the use of common command structure that makes the construction and execution of commands much simpler.

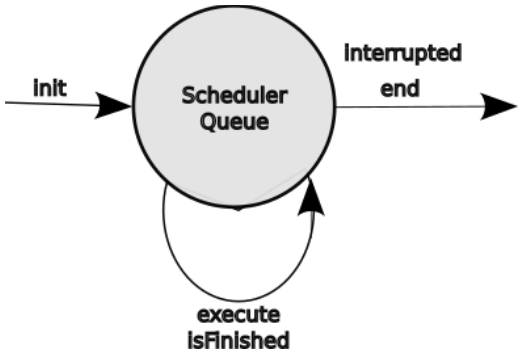
Subsystem Design Pattern

A subsystem controls something. Generally a subsystem controls a motor like an intake mechanism, shooter or elevator. Subsystems are hierarchical so a drive train subsystem can control the four swerve drive subsystems that in turn control a steering and a drive motor. At the top of the hierarchy is the robot subsystem that orchestrates the operation of the other subsystems.

One of the characteristics of a subsystem is that it can execute one command at a time. You can’t command an elevator to go up and to go down at the same time. This is enforced by the command scheduler invoked by the robot subsystem.

These components need to execute quickly and return control to the operating system so that other commands can simultaneously execute.

The second part of the coordination comes from a scheduler. The scheduler has a queue of commands waiting for their **execute** and **isFinished** methods to be periodically executed. The scheduler essentially has a loop so that it attempts to execute everything in its queue every 20 ms. There are other things going on in the roboRIO in addition to the scheduler. It manages several processes of the Linux operating system, runs the WiFi communication to get commands from the joystick and to send status and possibly video to the shuffle board display on the drive station.



Command state transitions within Scheduler

A couple of examples may help clarify the use of commands. Let's say the robot has an elevator. A command may be to raise the elevator to the top.

- **initialize** sets up sets the speed and desired end point for the elevator.
- **execute** is starts the elevator motor running. As the elevator nears the end point, it will slow the motor to ensure a smooth stop. Since the elevator is fighting gravity, it may also hold the elevator at the desired end position indefinitely.
- **isFinished** is checked to determine if the elevator has reached its desired height. It also checks the elevator limit switch just in case the height sensor is not working properly.

  interrupted allow the raising to be interrupted. The driver may have changed his mind about which level to raise the elevator to or has decided to do something else entirely. The elevator command could be ended.
- **end** is invoked to stop the elevator motor.

A second example is the drive chain commands.

- **initialize** sets up the joystick to be used to control the speed and direction of the robot.
- **execute** goes off and reads the current value of the selected joystick and applies it to the x and y velocities of the robot. This may apply algorithms to limit the acceleration and path of the robot so that it drives smoothly without tipping over.
- **isFinished** is will always return true as this command is never really finished.
- **interrupted** allows the driving to be interrupted. This could be an automated routine that takes over the manual driving when the robot approaches a loading or scoring station.
- **end** is invoked to stop the drive train and possibly to lock the wheels.

TODO Should include the default action of the Command class without overridding. This is to minimize code clutter.

Command Design Pattern

A command tells a subsystem to do something. Like a story it has a beginning , middle and end. The tricky part is a command has to fit into the context of multiple systems being controlled simultaneously and yet have the feeling that it is the only thing the robot is concerned about. This is done by breaking the code for doing the control thing into a set of parts of relatively trivial and quickly executed code. The command design pattern allows for the following five methods:

TODO talk about using the Command Class and overlaying the default methods and what those default methods are...

- **initialize** is a method to initialize and setup the parameters, attributes or states necessary for the command to run. It may set up the conditions to be met for completing the command.
- **execute** is a method invoked to start or continue the command every 20 ms. This has a very short duration and must return control to allow other commands to operate. This typically is to start a motor and may be used to dynamically control to the motor to vary its speed.
- **isFinished** a method that is called after every **execute** method to determine if the command is complete by returning a boolean: **true** for finished and **false** for not-finished. If it is finished, the **end** method is invoked. Some commands, like driveByJoystick are never finished by an internal criteria, so they would always return False.
- **interrupted** a method to handle a request to interrupt a command in progress and usually invokes the **end** method. If the robot detects that the elevator has exceeded its limit, say in its periodic method, it can interrupt the elevator subsystem command in progress to immediately stop the elevator.
- **end** a method to complete a command in progress either during or at the end of its execute cycle. This method brings the subsystem to a stable and known state. It typically stops the associated motor.

Commands typically originate from the drive station joysticks. There may be a button associated with a command to raise the elevator or a button to run the shooter while it is pressed. DriveByJoystick commands send velocity requests directory to the drive train subsystem.

Commands may be part of a complex chain of commands where some commands control the execution of other commands. For example a shooter may require that it come up to a minimum speed before it is fed a game piece. Some teams use incredibly complex chains of parallel and dependent commands.

The autonomous or auto subsystem issues commands to various subsystem to perform the autonomous routine.

Software Infrastructure

Software does not exist in a vacuum. It requires a lot of support to make it happen from the operating systems to allow it to run on a target machine, the development systems where the code is written, the repositories where the software is stored and the various tools used to keep everything running smoothly. This section goes through some of that infrastructure.

Linux Operating System

Under the covers. the roboRIO uses the Linux operating system. This is largely invisible to most robot users, but may be useful in some contexts. It is primarily used to manage the files and processes on the roboRIO. Linux, in general, is not known as a real time operating system, but since operations on robots are relatively slow (in computer terms) it is possible for it to mimic real-time services for the robot operations while providing a host of services behind the scenes.

Linux Core Services

The Linux core, like that used in the roboRIO, provides basic service. It provides a file management system so that files can be written and read and organized so that they can be found. It allows both access to its internal mass storage device, but it can also access USB thumb drives to save images or logging data. It provides process management so that the various tasks being preformed by the computer do not interfere with each other and that critical tasks are given more priority that less critical tasks.

At a low level Linux provides interface drivers for various protocols to alows tasks on the roboRIO to communicate with a wide variety of devices. The drives include:

- Wired Ethernet
- CAN bus
- CAN FD bus
- SPI bus
- I^2C bus
- USB
- Ethernet over USB
- Ethernet over WiFi

Linux Services

Linux offers a host of services that may be used to support a particular application.

- **WiFi Access Point** provides the handshaking to allow the FRC WiFi radio to be used as an access point or hot spot. This capability allows software development systems or the drive station to access the robot. The development system can download code and login via SSH to access files stored on the robot.
- **WiFi Client** provides the handshaking to allow the FRC WiFi radio to be used as a WiFi client during competitions to access the Field Management wireless Access Point.
- **DHCP** provides devices (like the drive station or a development system) acessing the roboRio via the WiFi access point with IP addresses. (During competitions, this function is provided by the Field Management system.
- **ssh** allows remote access over Ethernet to a terminal shell within the roboRIO. This allows a development system to access logs or any installed Linux command line function. The access may be from a terminal window in Linux or Macintosh systems and PUTTY or a terminal window in Windows machines. You can set up that access to use a public key exchange rather than passwords to be more convenient.
- **scp** uses an ssh connection to transfer files securely from a terminal window without having to log into the remote system.

Wireless Networking

FRC robots use WiFi connections to provide wireless control over the robots. Within the robot there is a open mesh radio which can be accessed directly for practice sessions on the home field. At competitions this radio must be accessed through the Field Management System both to control access to the radio and to allow the Field Management System to exert safety and event controls over the robots.

Network Tables

Network Tables is a National Instruments implementation of a publish and subscribe service on the roboRIO. Publishers send information to the server which forwards the information to subscribers who have previously requested the information. In this way the publishers do not know or care about the consumers of their data.

Data has two fields: topic and value. The topic is a string that identifies the data. It uses a pseudo file-structure-like format to allow hierarchical specification of the data. Classifications are separated with slash characters. So a topic might be ‘/swerve/right-front-steer/angle’ for the right front steering motor

angle. The value can be any Java type, like double, int or String. The type of the value is set by the first publisher and cannot change for the life of that topic. The server stores a value of each unique topic (called an instance) and overwrites that value when it receives a new value for a that topic. A subscriber requesting a topic that already exists is returned the current value of that topic and any updates to that topic.

There is much more information about Network Tables in the WPI Documentation. WPI uses Network Tables to populate the Shuffleboard and can be used for primitive logging of data.

[This is not as robust as MQTT which allows the use of wild cards in subscription requests.]

WPI Logging

This is the normal logging supplied by the WPI LIB. It easily records data sent on the Network Tables to the shuffleboard, but may include other data. This should include:

- robot x, y field coordinates as they change.
- robot pointing or pose angle as it changes.
- robot x, y and rotational (yaw) velocities (although can be derived from the x and y)
- robot roll and pitch.
- robot roll and pitch velocities to help detect collisions.
- robot drive command as they occur.
- elevator position as it changes.
- elevator commands as they occur.
- arm position as it changes.
- arm commands as they occur.
- intake state as it changes.
- intake commands as they occur.
- other subsystem states as they change.
- other subsystem commands as they occur.
- robot voltage
- robot current if known.
- lime light info
- other telemetry information

Advantage Kit Logging

Team ___ has introduced advantage kit logging. The general idea of this logging is to allow complete replay of any practice run or match by recording all of the inputs to the robot and then replaying the recorded inputs during a simulation run. This is uses an interface abstraction to capture and replay any value.

This really needs to be expanded out TODO.

Software Runtime Environment

- is the environment where the code runsF
- Linux and all of that on the robot
- Drive station
    - interconnects joysticks and other controllers to control the robot
    - button, axis and hat to command assignments
    - dashboard to display the status of the robot
    - shuffleboard to display the status of the robot
    - advanced use of the shuffle board to set control parameter in the such as PID parameters.
- Field Management system
- Wireless communication or tethered operation
- all of the motors that use controllers. Their PID routines must be tuned to work effectively.

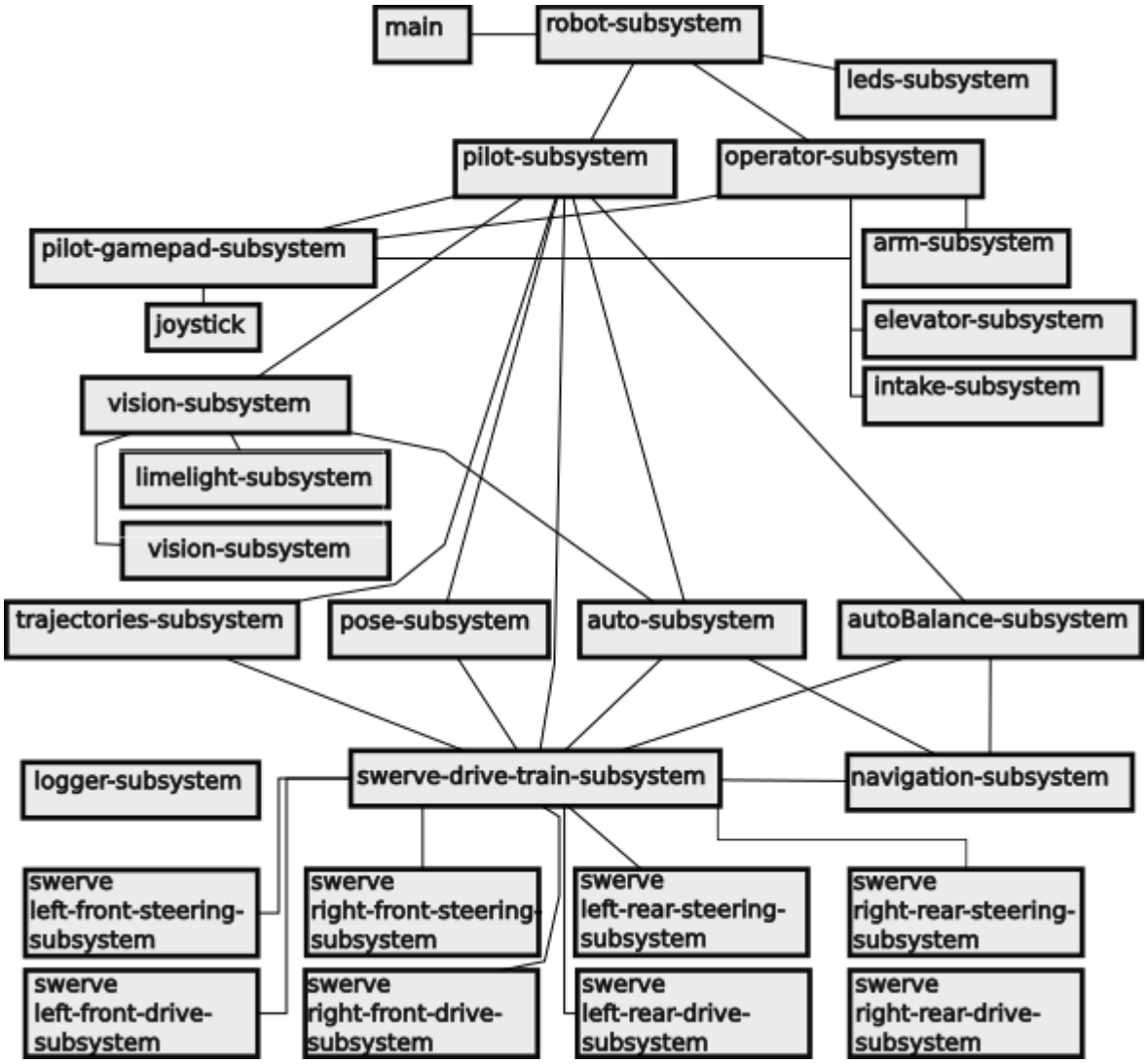
Path Planner

- Tool for laying out path segments for autonomous routines
- Can lay out dynamic paths to reach particular poses on field (e.g., pickup or scoring)
- uses spline curves like bezier curves to have smooth path
- also uses acceleration and jerk profiles to make motion smooth and controlled, like a PID routine on robot movement.
- Can use way points for more complex paths or to avoid fixed field elements.
- TODO Fix this with English and more detail

Background on Language Selection

FRC robots are supported by a software library called WPILIB. It was developed by the Worcester Polytechnic Institute (WPI) and it has libraries for most of the major components of a FRC robot. The WPILIB supports three languages: National Instruments LabView (a graphical programming language), Java (a modern object oriented language), and C++ (a version of the traditional C language that supports object oriented programming). FRC 4513 has chosen Java because it is a textual language and not as complex as C++.

There are many ways to learn Java programming. One way is to use the Team 20 Java video series at Team 20 Intro to Java Programming. W3C schools has a quick introductory Java course at https://www.w3schools.com/java/default.asp. This goes through the basic syntax and operations for Java, but light on the details and nuances. There are more complete courses on the Internet and in books.



Subsystem Software Architecture

Subsystem Software Architecture

Robot Module

The entry point for Java programs is a function called **main**. Java programs are compiled into a block of byte codes. Then the byte codes are executed by an interpreter on the target machine, control is transferred the entry point defined by the **main** function. In this architecture, the **main** module just passes control to the robot module, which stands at the top of the robot software architecture. It creates instances of the major subsystems.

It provides methods for major operation modes such as the **autonomous** mode where the robot is entirely controlled by the computer and the **teleop** mode where human pilots drive the robot with computer assist. The **autonomous** mode may have various options to fit particular scenarios so that the team can cooperate with other teams in its alliance during the autonomous period including selecting different starting positions. It also includes various modes for testing the robot and for simulating robot functions.

Subsystem Software Architecture

A subsystem controls a a device or set of devices within a robot as a single process running one command at a time. Each subsystem is managed with a common set of methods:

- **setupDefaultCommand** is a method to establish a default command within the subsystem.
- **periodic** is a method that is called periodically to poll the status of various switches and sensors pertaining to the particular subsystem.
- **commands** is a set of methods that control the subsystem. Each method does only one thing like start the motor, stop the motor, etc. Only one command can be run on a subsystem at a time.
- A **telemetry** module within a subsystem is used to convey information about the subsystem to a tab in the **shuffleboard** display at the drive station.

Subsystems

The following is an alphabetical list of the current subsystems used by the robot.

- **arm** controls and monitors the position of the arm.
- **auto** controls the robot during the autonomous period.
- **autoBalance**-controls the robot to balance the robot on the charge platform without human intervention. *This is a game specific subsystem.*
- **elevator** controls and monitors the position of the elevator.
- **intake** controls and monitors the intake mechanism to load, process and unload game pieces.
- **leds** controls LEDs on the robot.
- **limelight** controls and monitors a vision system capable of reporting certain location and game piece information. *(This does not appear to be hooked up in the current 4513 code.)*
- **logger** log various events for debugging and performance evaluation. Most subsystem generate events for logging.
- **joystick controller** interfaces a joystick by normalizing its inputs and applying those inputs to a set of commands passed to the robot.
- **operator** provides control and feedback interfaces to a human co-pilot operating a joystick controller.
- **pilot** provides control and feedback interfaces to a human driver operating a joystick controller.
- **pose** provides an estimated x and y coordinate of the robot as well as the angle of the robot with respect to the field. *(This is only used by the swerve subsystem in the current 4513 code.)*
- **drive train** controls the motion of the robot by specifying a desired rotational position, x velocity and y velocity, all with respect to the field. The underlying drive train may be anything, e.g., a swerve drive, tank drive, or west coast drive.
- **swerve module** controls the drive velocity and the steering angle of a single swerve module. It is controlled by the drive train subsstem.
- **trajectories** methods for driving the robot over paths determined on the fly to minimize unnecessary movement to maximize speed and agility.
- **climber** is a subsystem used in games with a climbing challenge to lift the robot off the floor.
- **intake** is a set of methods used to autonomously intake a game piece from a loading station or off the floor.
- **score** is a set of methods used to autonomously place or shoot a game piece. *Ficticious!

Drive Train Subsystem

It is useful to have a drive train abstraction as a subsystem. The job of this abstraction is to translate desired robot movements into individual module commands. For instance you want to the robot to move forward and the abstraction figures out the commands to send to the individual motors to move forward. This is simple enough to understand, but this gets a bit more complex when you want to turn and how tight you want to turn because the speeds of individual motors is controlled to bring on the turn. In swerve drives the swerve modules will be given steering commands as well. You want to hide the details of how individual motors are being commanded, so you can worry about the overall

movements of the robot. In the WPILib, the drive train abstraction is given a simple move commands with three parameters relative to the field frame of reference:

- desired x velocity,
- desired y velocity, and
- desired facing angle.

The vector resulting from the desired x and y velocities is the heading or direction in which the robot will move. In tank and west coast drives the heading and facing angle are the same, except when no velocity is requested and the robot just turns on its own axis.

The robot's pose on the field is a Pose2D object which reflects the x and y field coordinates in meters and the robot facing angle relative to the field coordinates. These coordinates are estimated by the odometry subsystem and may not be accurate due to the accumulation of errors. The angle for this pose comes from the robot's gyroscope. This too may have some error as the gyroscope will drift over the course of a match. Both are accurate enough for basic navigation, although there are techniques to reset these values during a match to make them more accurate.

Most of the time the robot is driven from a driver frame of reference. Pushing a joystick forward sends the robot down field, pulling on it moves the robot back toward the drive station, pushing right sends the robot to the driver's right, and pushing left send the robot to the driver's left. Note that the driver's frame of reference may be different than the field frame of reference when the team is on the red alliance.

There may be some instances where the robot is controlled from the perspective of the robot. The driver has to imagine what commands are necessary for the desired movement.

Drive Motor Subsystem

The subsystem for the drive motor is similar for all drive trains other than the swerve drive. Drive motors are given commands to request changes to their velocity. With modern drive motors is a controller that actually controls the motor, so this does not have to be managed directly by robot software. The motor control algorithms are supplied with a set of parameters that control the motor with the desired performance.

Odometry Subsystem

The odometry subsystem manages the system gyroscope to determing the robot facing or pose angle and to maintain an estimate of where the robot is located on the field. Motion requests to the drive subystem command the robot to turn to a specific angle. The odometry synsystem provides that angle to know when the robot is on target.

Tank Drive Train Subsystem

Translate drive train commands into motor speed commands for the left and right side motors.

The drive equations are something like

$$v_{request} = \sqrt{v_{xReguest}^2 + v_{yReguest}^2}$$

$$\theta_{target} = \arctan\left(\frac{v_{yRequest}}{v_{xRequest}}\right)$$

$$\theta_{error} = \theta_{target} - \theta_{current}$$

$$v_{leftSide} = v_{reguest} + f(\theta_{target}, \theta_{error})$$

$$v_{rightSide} = v_{reguest} - f(\theta_{target}, \theta_{error})$$

Where the function f supplies a correction factor to change the heading using something like a PID routine so that it decreases to zero when Θ_{error} drops to zero°.

With tank drive you have to make a decision to optimize movement and just reverse or to cause the robot to turn 180°. In the former case, the driver would have to deliberately make a 180° turn. This decision is a bit more complicated in that the it is really a decision about driving to field orientation or from the perspective of the robot

West Coast Drive Train Subsystem

Translate drive train commands into motor speed commands for the left and right side motors in a similar manner to the tank drive subsystem. Odometry is complicated by the robot shifting to a front or back center of gravity.

Mecanum Drive Train Subsystem

Translate drive train commands into motor speed commands for the four motors to achieve the desired robot motion.

$$v_{request} = \sqrt{v_{xReguest}^2 + v_{yReguest}^2}$$

$$\theta = \arctan\left(\frac{v_{yRequest}}{v_{x\,Request}}\right)$$

$$v_{frontLeft} = v_{reguest}\cos(\theta) + v_{reguest}\sin(\theta)$$

$$v_{frontRight} = v_{reguest}\cos(\theta) - v_{reguest}\sin(\theta)$$

$$v_{backLeft} = v_{reguest}\cos(\theta) - v_{reguest}\sin(\theta)$$

$$v_{backRight} = v_{reguest}\cos(\theta) + v_{reguest}\sin(\theta)$$

This is not accounting for spinning the robot while moving, which is possible.

TODO. Work out the stationary case first.

To spin about the robot center, the command comes with requested x and y velocities of zero and a desired angle. The subsystem decides which direction to turn and continues to do so until the desired angle is reached.

To turn clockwise about the motors are commanded:

$$v_{frontLeft} = v_{reguest}$$

$$v_{frontRight} = -v_{reguest}$$

$$v_{backLeft} = v_{reguest}$$

$$v_{backRight} = -v_{reguest}$$

To turn counter clockwise the motors are commanded.

$$v_{frontLeft} = -v_{reguest}$$

$$v_{frontRight} = v_{reguest}$$

$$v_{backLeft} = -v_{reguest}$$

$$v_{backRight} = v_{reguest}$$

Swerve Drive Train Subsystem

To understand how swerve drive trains work requires a little math background. Vectors are a way of expressing a direction and a magnitude. This could be something like go west at 30 MPH or the house is ¾ of a mile southwest of the town center. For the swerve drive train, these vector are a direction and a velocity. This vector starts as a simple joystick command where the driver pushes the joystick which produces an x and a y value. These two values are combined as a vector to express in what direction the driver want the robot to move and how fast. That's simple enough and the swerve drive train library has a move command with the parameters for the x and y components of movement with a boolean to select whether the coordinates are in the field frame of reference or the robot frame of reference. Normally the driver uses the field frame of reference because that is what he or she sees. The driver can select to drive using the robot frame of reference if the robot is driven with a video feed from the robot and the driver wants to move relative to what is shown on the video. The movement commands are sent as velocities rather than as coordinates, because that is more intuitive to the driver. There may be some adjustments to the raw joystick values to the values used by the robot because the joystick gives a normalized value between -1 and 1 and the drive train uses a velocity expressed in units of meters per second. Also the raw joystick values may be altered to fit a non-linear curve to give the driver more control over lower speeds than high speed.

Swerve Module Subsystem

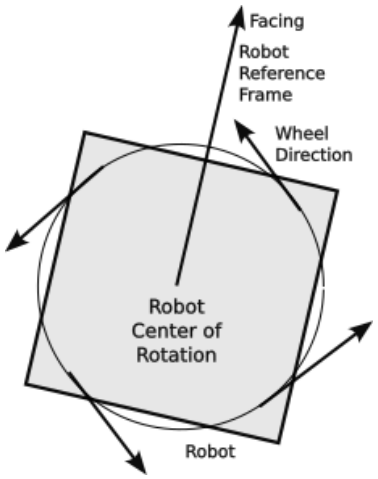
The swerve module subsystem is really just two drive motor subsystems, one for steering and one for the drive power. The drive train for a swerve drive must control all eight motors to achieve the desired motion. Basic movement is by steering all modules in the same direction and applying the same power to all drive motors. A spin in place maneuver can be performed by pointing the swerve wheels tangent to the center of rotation which is simplest if the center is the center of the robot so the same velocity can be applied to the drive motor of each module.

A minor problem is that each swerve drive has its own frame of reference for the rotation of its wheel. The angle of the wheel for rotation of the robot is dependent on the x and y distances of the center of the swerve module to the center of the robot. Robots need not be a shaped like a square, nor do the swerve modules need to be positioned symmetrically in the robot. These adjustments are made in the configuration constants used as swerve drive attributes.

A wheel alignment procedure is performed from time to time. First the wheels are turned so that they all point in the same direction with the drive gear on the same side of the wheel. Then the wheels are aligned with a straight edge for further refinement. The turn angle of each wheel is noted at the offset to zero angle needed to drive in that direction.

Spinning While Moving a Swerve Drive Robot

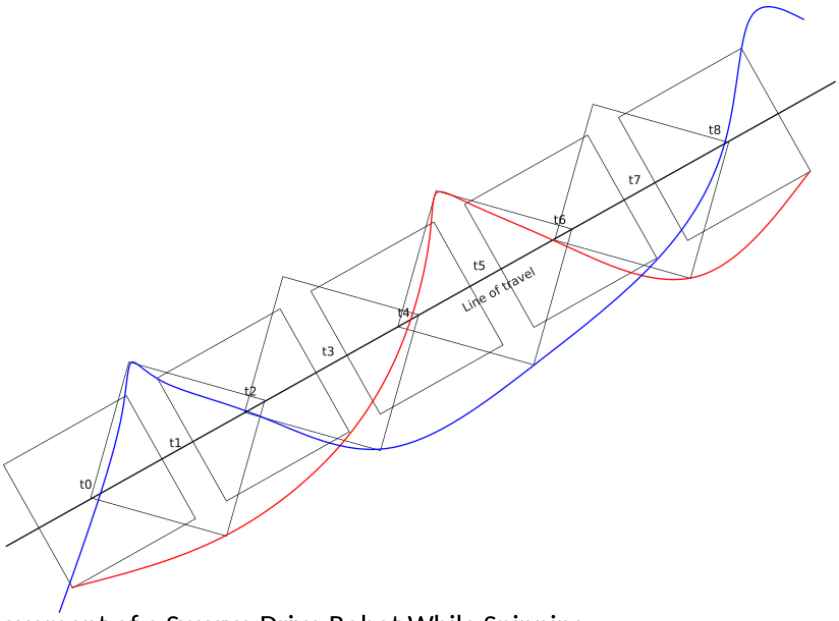
So simple enough. Push the joystick in the direction you want to move and the robot moves in that direction. What about rotation or spinning? Normally the robot rotates about its center and that is what we will discuss for the moment. The angle of rotation for each swerve module is tangential relative to a circle centered on the robot and passing through the center of the swerve module. This is illustrated in the figure to the right.



Rotating a Swerve Drive Robot

To spin the robot while moving requires the angle and speed of each swerve module be coordinated by the drive train to have a smooth and efficient movement. To do this, the software does a vector sum of the commanded movement vector and a rotational vector for each swerve module. It also has to translate the resulting vector relative to which ever movement reference frame is used to the reference frame used by each individual swerve module.

The swerve drive train interface uses four parameters: X velocity, Y velocity, rotation velocity, and a boolean to select either the field frame of reference or the robot frame of reference. The rotation velocity is the same for either the robot frame of reference or the field frame of reference. However, internally it must convert the angular velocity to the same units as the X and Y velocities, i.e. to meters per second. Multiply the angular velocity in radian per second by the distance (radius) of the swerve drive to the robot center. The angular velocity is perpendicular to the radial between the robot center and the swerve drive. The X velocity and Y velocity together define a robot velocity vector which is an angle and a magnitude.



Movement of a Swerve Drive Robot While Spinning
(red line traces one wheel and the blue traces the wheel of the opposite corner. Only two wheel traces are shown.)

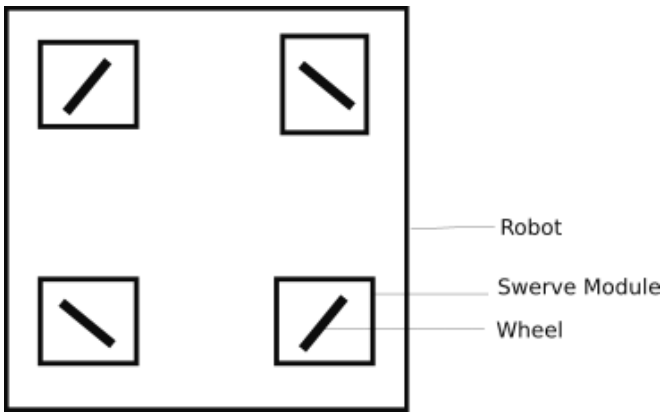
The figure to the right shows a robot spinning as it moves forward. This shows two things: as the robot spins its apparent width changes and the speed of the robot is diminished to allow one side of the robot to spin. This is the default way to spin the robot while moving forward. The path of two corners is shown, one in red and one in blue. Time is marked along the line of travel. Because one side of the robot must slow down for the robot to spin, the overall speed of the robot is reduced. Spinning in general is not efficient as it slows the forward velocity of the robot.

Note that the arc tangent is a little more complicated than pure math. The normal math function will return an angle between -π/2 radians (-90°) and π/2 radians (+90°). We want the angle between 0 and

2π radians (360°), so the signs of the x and y determine the quadrant of the Cartesian coordinates which is used to extend the range of the returned angle in the Java robot math (Rmath) class.

Locking the Wheels on a Swerve Drive

At times it may be desirable to make a robot harder to move or push. This is done by locking the wheels by turning all wheels to be “tangentially aligned” (i.e.. aligned perpendicular to the radial connecting a wheel to the center of the robot.) as in the figure to the right:



Position of Wheels When Locked in a Swerve Drive

Joystick Subsystem

The joystick subsystem maps the joystick controls to particular subsystem commands. The following modifiers may be used:

- **onPress** to only apply the command when the button is pressed.
- **onRelease** to only apply the command when the button is released.
- **whilePressed** to apply the command while the button is pressed.
- **and** to apply a button to another command to work like a shift key for another button.

The joysticks have two separate axis that produce an analog value: one for the x axis and one for the y-axis. TODO list the range.

The triggers also produce an analog value. TODO list the range.

The hat can be pushed in 8 different directions each producing one or two button presses (up, down, right, left, up-left, up-right, down-left, down-right).

list alll of the control and their output ranges or conditions.

A_BUTTON = 0

B_BUTTON = 1

X_BUTTON = 2

Y_BUTTON = 3

Left bumper LEFT_BUTTON = 4 # above left trigger

right bumper RIGHT_BUTTON = 5 # above right trigger

VIEW_BUTTON = 6 # overlapped squares

MENU_BUTTON = 7 # hamburger strips

XBOX_BUTTON = 8 # X-Box logo

LEFT_JOY_BUTTON = 9

RIGHT_JOY_BUTTON = 10

right joystick

x

y

left joystick

x

y

right trigger

left trigger

hat (f, b, r, l, fr, fl, br, bl)

should also talk about non-linear mapping to analog controls

LED Subsystem

The LED subsystem is used to control the LEDs on the robot. Normally this would be a single string of serially accessable LEDs. The string can be addressed as a whole or broken into to sections. Commands are created to do particular displays like display a solid alliance color, display a flashing alliance color, display a rainbow, display a rainbow wave, etc. Each command applies to a non-overlapping set of sections. The overall LED command clears the buffer, updates the display for a set of sections, and finally writes all sections out the the LED string.

The WPI LIB provides a library for using a PWM channel of the roboRIO for communicating with LEDs in this fashion.

TODO need to verify the above

TODO need to describe the LED protocol.

- each LED has three colors red, blue and green.
- each color of each is controlled by 24-bits, 8 for each color.
- The 24 bits for an LED are sent in some serial fashion then a gap of __ ms and then more bits sent in the same fashion until all of the bits for the string have been sent
- The first LED in the string takes the first 24 bits to control its LED and passes the remaining bits down the string
- the second LED in the string takes the first 24 bits that it sees (really the second set sent) to control its LED and passes the remaining bits down the string.
- And so on for all LEDs in the string
- A gap of 50us or longer signals that a new set of bits is being sent
- formula for the length of time to transmit all bits based on the number of LEDs (30us * number LEDs) + 50us.
- warning about power consumption
    - only display one color at a time
    - do not use full brightness

TODO Remove overlap with basic electronic section...

Use of library...

set up sections. Each section is to be a group of lights affected by commands. Commands are sent to change the color or mode of the lights in a segment. Multiple commands for multiple segments can be enforce at a time. Not sure if a FSM approach here is justified, as it just reflects would be going on with commands.

Improve section definition.

Improve modulo for light control. Each LED command can be executed at a slower rate than the norm 20 ms loop. Part of problem is how long it takes to send out the bits. Part is delay for the eyes.

Commands with repeat should use system time as a basis. Take the modulo of the time using the period of the repeat as the modulo. Within the time you can do things like turn on or off for flashing, advance a color pattern, advance a marquee pattern, advance any other desired pattern.

Just like other commands.

- Init to set up what you want to do (modulo, lamp state, etc.)
- execute to advance the lamp state
- end to release any resources used by the command.

Software Development Process

Software Development Steps

- Study: understand how the existing code works
- Plan: on the changes to be made (see next section on strategies)
- Design: figure out the major design changes to subsystems, commands, etc.
- Write the code
    - edit
    - compile cleanly
    - deploy
    - debug: find bugsor things that do now work as expected. This may involve adding some temporary code to print out values or progress. Studying the results to infer why things are working they way they are and develop a fix to make things work as expected.
    - loop back to writing
- accept the code when it works as expected (and the documentation is done)

Software Development Strategies

Try to work incrementally.

- Do small steps with minimal changes to the existing code base
- Small changes are easier to test and debug
- Small changes are easier to understand
- Small changes have smaller risk
- Quicker cycles can be more rewarding and less frustrating
- Complete each step before moving to the next step.
- Large steps inherently have a lot of problems and risk both the base and the schedule

Thius is also know as agile or cascaded software development.

Software Development Tool Chain

A tool chain is the set of tools used to develop software programs. It can be as simple as a run time environment that allows entry and direct running of a programming. Such an environment limits the complexity of the code that can be developed. Robotics code requires a more complex environment.

Integrated Development System

TODO Fill out this outline

- Visual Studio Code is an “Integrated Development Environment” or IDE which includes access to several software tools from a single window.

    - An editor for entering the text of the program. This is flexible allowing various key maps to simulate may editors, including a Linux favorite for 50 years, vi.
    - A syntax checker to point out syntax errors in near real time as you edit the program.
    - An easy interface to compile all of the modules of a program to build the Java .jar file for deployment
    - An easy interface to deploy the compiled code to a target robot.
- WPI integration to keep different versions separate. The WPI libraries change each season. The libraries and source code for a robot for a year are kept together so that they will compile and simultaneously are kept in separate trees so that the current code accesses the current library.
- Compiler
- Gradle… whatever it does
    - puts together all of the files to download to the target robot
        - robot.jar file with all of the Java code
        - path planning .json files
        - any othe files needed by the robot...what are they??
- deploy.. download to the target robot

Source Code Editing

Source code is written on an editor or integrated development system. The later includes many tools so that a developer can see the file structure, code and other tools at the same time and use the same interface to do many tasks.

Source Code Management (git)

A source code management system keeps track of the changes to the software over time. It allows multiple people to work on the same code base at the same time and provides tools for resolving conflicts when two people change the same code module. It allows an unsuccessful or experimental code change to be backed out. If used properly it can keep track of who made what changes and why the changes were made. Saving code to a remote repository is also a way to maintain a backup of the source code during development.

One such code management system is **git**. You can use git on you development system locally or you can use it with an external code depository such as github.com or gitlab.com to allow others to access or, if trusted, to modify the code depository.

Basic code management has three phases:

- staging where modified files are **add**ed to a group of files to be included in a committed change. The s**tatus** command an be used to list the files which have been modified as well as the files staged for commit.
- The **commit** command saves a copy of the staged files so that the set of files can be reproduced should future changes do not achieve their objective or to retrieve the current version of files. It. also changes the state of the staged files to unchanged for future staging. In general, commits should cover as small of a set of changes as possible to give granularity to the change and to implement a particular function.

- The **push** command is to upload one or more commits to a remote repository.
- (A **pull** request is a change outside of the normal source code control change. This lets outside developers propose changes to fix a particular problem or problem. This probably will not be used in robot code development, except to propose changes to the external code libraries.)

Source Code Management (git)

TODO Fill out this outline and merge with previous

- normally one can use git to manage different versions of software code.
- The basic idea is to write code in fairly short segments and check point or commit that code. These commit points allow the program code to be rolled back to a particular check point, even if multiple files are involved.
- Once code is committed locally, it can be pushed to an on-line repository, like github.com or gitlab.com. Code on the on-line repository can be accessed by other members of the team or even be made public to the world. Changes to the same repository can be coordinated even when changes are made to the same file.
- Steps:
  - write
  - stage – add files to be committed
  - commit – mark the files as part of a general code checkpoint. The commit should be commented so that the commit is significant: what is being committed and why. Just saying “fixed bugs” is not descriptive. Saying “Fixed problem 3429” lets you go back and find out what problem was addressed. Of course tested and verified code is an important milestone that should be committed.
  - push – committed changes should be pushed to the repository when the code is tested and ready for use by others.
  - Pull request – sometime code changes need to be coordinated. A pull request is code that has been tested and verified, but is on another branch of the code. The code is pulled in if it has no difference with the main branch. Conflicts have to be resolved, by accepting code differences line by line or by re-writing the conflicted module.

- Support tools
  - Drive station to access the roboRIO log files in real time
    - Once the drive station has access, they can be accessed in a window of Visual Studio Code.
  - program and maintain CAN bus controllers
    - CTRE Tuner X for accessing CAN FD Bus devices and CTR devices like the Pigeon. It is also used to upgrade the software on devices with CTR controllers.
    - CTRE Phoenix Tuner
    - REV Hardware Client

- git and gitHub or gitLab

Debugging

All code that is writted must be tested to ensure that it works as the programmer intended. In the course of that testing, it is inevitable that anomalies or bugs will be found. These bugs need to be corrected so that the program works as it is intended to do. Never assume that a change will work the way that you think that it will. Test it and make sure the code is correct.

Simulation

Simulation is a way to run the code without having to load it on the physical robot hardware. In simulation mode the various physical subsystems are replaced by software that acts the same way as the hardware that it represents. While simulation my not be perfect, it does offer a way to test software before hardware is ready or in use by other team members.

Deploying Code to the Robot

The WPILIB/VS Code environment has a method for deploying code to the robot. This looks for an attached robot and interacts with it to download the robot control code. After the download is complete, the new program is immediately executed.

Log Analysis

Logs kept by the robot allow for analysis of the performance of the robot after a practice session, match or competition. The goal of this analysis is to uncover the underlying problem and to help formulate the corrections to alleviate such problems in the future.

- Logging
  - log shuffleboard data
    - Display data on the drive station
    - Mostly used for debugging
    - some could be used during competions, but remember the drive team has a limited bandwidth.
    - may log activity on the network tables
- Advantage view
  - visualization of logged data
  - read odometry data to display on a two-dimensional or three-dimensional display
  - show data in real time
  - show current usage by the robot overall or individual motors
  - show temperature readings
- advantage kit logging
  - log “everything”
  - all inputs to processes pass through an abstract “interface”
  - possible to replay a match using the real inputs

- keep track of the battery used
  - each battery has a QR or bar code which is read when the battery is loaded (or robot is started).
  - allows analysis of the amp hours supplied by a battery per charge, high current spikes on the battery and the history of battery usage.
- Keep track of the software version
  - know the github version
  - know the versions of code that has changed but is not staged.

Language Selection… basically a footnote

Differences in language types

- machine language is the codes used within the processing unit. This code is the closest you can get to the machine and can be the fastest possible code. This code is dependent on the processing unit on which it runs.
- assembly language is a human readable form of machine language using mnemonic codes and simple arguments. Like machine language, assembly language is dependent on the processing unit on which it runs.
- compiled language is a human readable language that is compiled into a machine readable language, usually machine language. An example of a compiled language is C or C++.
- interpreted language is a human readable language which can be indistinguishable from a compiled language (or a compiled form of the interpreted language). The difference is that the interpreter language is interpreted at execution time. This language form is typically the slowest of all language forms. An example of an interpreted language is the JavaScript code that runs in most browsers.
- tokenized language is a human readable langage which is compiled into an intermediate machine readable form which in turn is interpreted at run time. This is normally quite a bit faster than pure interpreted language because the language elements do not have to be interpreted and analyzed. There must be a run time environment on the target machine to interpret the tokenized code. An example of a tokenized language is Java.

Object Oriented Languages

An object oriented language uses classes to extend data types by defining attributes of the class and methods to perform operations on the class and its attributes. A class my be instantiated as an object any number of times. Java, C++, Python are object oriented languages. JavaScript can do object oriented like code or use procedural forms.

Other language type include procedural where code is written to process inputs (e.g., C, COBOL, BASIC) or functional where the code is written as functions (FORTH, Clojure, LISP, Erlang). Of course purists love to debate about the type of langages.

Computer Science Topics

Java programming uses a lot of Computer Science jargon and concepts. This section briefly explains some of those concepts.

Declaration, Initialization, and Assignment

In a "typed" language like Java all variables and attributes must be declared before they are used. This declaration allocates space for the value and assigns it a specific type. When the variable or attribute is used, it type is passed along so that it us used according the rules of its type. For example, integers can be added together to form a new integer value with the plus operator. The same plus operation can be used with Strings to concatenate them into a new String.

Variables and attributes must be assigned a value before they are used. Java will issued a "Null Pointer Exception" if this rule is broken.

A variable or attribute may be initialized, that is, declared and assigned an initial value at the same time.

Methods, Functions, Procedures and Subroutines

In general methods, functions, procedures, and subroutines are vary similar. All can be invoked or called to perform a task, usually a task that is performed repeatedly. All may be passed zero or more arguments (also called parameters) as input. Not all tasks require an argument. A function returns a value. It might be mathematical, like return the square root of input argument *x*. Procedures and subroutines usually do not return any values. A method is similar in that it performs a task, but it operates within the scope of an object. It can use and manipulate attributes of that object. It can return one or more values to the invoking code like a function.

Variables, Constants, Attributes, Parameters, Arguments

A variable is a declared value whose value can be changed by the program.

An attribute is a variable tied to the scope of an object or its defining class. In Java almost all variables are really attributes of an object.

A parameter is a value passed to a method, function, procedure or subroutine.

An argument is a defined value which can be passed. For example, lets say you have a function that returns the square root of the value *x*. The argument of the function is x. The value of the argument could be anything, but it is 4 when you want the square root of 4. You could have passed the value of a variable or constant or attribute to the function and that value is assigned to the argument of the function. Arguments in Java are positional, meaning that a value in a particular position within the method parentheses is treated a particular way. Java does not have a way to name arguments.

Arguments for a Java method are defined when the method is defined.

Parameters for a Java method are used when the method is invoked. The value of the parameter is passed to the method as an argument and it must be the same type as the declared argument.

Pass By Value

Pass by Value is when a passing an argument to a method, only the value is passed. Changes to the argument internally do not affect the value of the original argument. For example:

```
int fum (int fumIn) {
     fumIn = fumIn + 1;
     return FumIn
}
```

```
int foo = 2;
fe = fum (foo);
// fe is 3
// foo still is 2
// fumIn no longer exists, the scope of fum no longer exists
```

When an object is passed as an argument, it is passed by value. Its value is copied, not a pointer or reference to the object.

Pass By Reference

Pass by Reference is when passing an argument to a method, the a pointer or reference is passed. If the called method modifies the passed argument, the value of the original argument is modified.

```
String fum (String fumIn) {
     fumIn = fumIn + “!!”;
     return fumIn
}

String foo = “this is a string”;
fe = fum (foo);
// fe is “this is a string!!”
// foo is also “this is a string!!”
// fumIn no longer exists, the scope of fumIn exists only
// within the execution of fum
```

Encapsulation

Encapsulation allows for a variable name to be reused in different parts of a program without conflict. This is normally provided by a couple of mechanisms. Most things can be accessed only within the scope in which they are defined without special modifiers.

Modularization

Modularization allows a program to be broken up into several files for ease of maintenance. A program that needs the code of another file may **import** that file.

Inheritance

Inheritance allows the methods and attributes of a class to be inherited by a subordinate class. This allows a class to use the methods and attributes of its parent or superclass. There is a mechanism to override those classes or methods that do not apply the subordinate class.

Polymorphism

Polymorphism allows a method to be overridden. For example a class defining math operations may overlay a method (such as square) with different return types and argument types. Each tuple of return types and argument types is called a signature. For example a math class with methods for squaring integers, floats and doubles:

```
class MyClass {
     int square (int in) {
          return in*in;
     }
```

```
      float square (float in) {
            return in*in;
      }

      double square (double in) {
            return in*in;
(     }
}
```

The method square is overlayed with three signatures:

- int, int
- float, float
- double, double

Signature

A signature is the tuple of the return values and input arguments to a method. For polymophism multiple methods can share the same name (and basic functionality as long as they have distinct signatures.

Basic Java

The code for the FRC 4523 robot is written in Java which is a modern Object Oriented Language. Java was developed by Sun Microsystems in the 1980s to be a portable language. In general computer languages fall into two major types: compiled and interpreted. Compiled languages translate source code into the native machine codes for the target computer. Interpreted languages are parsed at run time to perform the actions directed by the programming. Java is a hybrid type as it is compiled into a set of “byte codes” which are then stored in a “.jar” file. The “.jar” file is interpreted on a target machine by a Java runtime. Since the Java runtime is written specifically for the target machine, the .jar code is portable between different machines. The Java runtime is a subset of a normal Java runtime as it contains only the features needed by robot systems.

Learning your first computer language is a journey of exciting new concepts and understandings. There are many courses available in books or online to teach the basics of many languages including Java. Many of the concepts apply across many languages, so once you know one language it is relatively easy to move to other languages. The intent of this handbook is not to teach you everything you need to know about Java, but it does have a few things to remind you of some concepts that are important to understand when reading or writing FRC Java based robot control code.

A list of the key words (the words used by Java language and should not be used for the names of anything defined by the user) can be found with a short definition at https://www.w3schools.com/java/java_ref_keywords.asp. This does not include the use of “decorators”, words preceded with an at ‘@’ character.

Much of the syntax of Java is similar to C, JavaScript, Rust, and a few other languages in that it:

- blocks of code are fenced off with braces ‘{‘ and ‘}’. These blocks also control the scope, although the scope rules vary between languages.
- expressions in a conditional are enclosed in parenthesis ‘(‘ and ‘)’.

- members of an array are accessed with an index enclosed in brackets '[' and ']'. For example to access the second element the array 'cars' one can write 'cars[1]'. Note that array indexes are zero-based and not one-based.
- Long comments are set off by slash-asterisk '/*' and asterisk-slash '*/'. Comments at the end of the line are set off by double-slash '//' and extend to the end of the line. Comments are ignored by the compiler and are used to provide information about the program to human readers.

```
int robot1 = 1; // this comment goes to the end of the line
int robot2 = 2;
/* this is a multi line comment
int robot3 = 3;
the above line is commented out, as is this one /*
```

- A statement must end with a mandatory semicolon ';'.
- Conditionals use a common form:

```
int robot2 = 2;
if (condition) {
     /*then block*/}
else {
     /* else block */
};
```

- For loop statements:

```
for (/*initialization*/; /*done condition*/; /*loop increment*/) {
     /* for block */
};
```

These similarities allow one to move easily between languages. However there are some differences and these differences sometimes cause confusion and difficulties because what works in one language may not work in another. This is especially true when using libraries as they have different usage.

Java uses types extensively. They are part of the declaration of variable, attributes, arguments, and methods. They are checked when these are used in other parts of the program. There are some studies that conclude that highly typed languages are more reliable because the right things are being used and acted upon. So just what is a type. Java has four primitive types:

- **int** or integer for positive and negative whole numbers.
- **float** for positive and negative fractional numbers.
- **double** for more precise fractional numbers. This is the defacto type
- **char** for single characters. Characters also use single quotes like 'a'. Single quotes cannot be used to delimit a string as in other languages.
- **boolean** for values that are either true or false.

In the robot code, there are many uses of **int**, **double**, and **boolean** types. **float** and **char** are not used often, if at all.

Types are extended by classes. **String** is a built-in class to handle arrays of characters. User defined classes also extend type. Robot code uses many of these classes, e.g., Pose2D, Pose2D, Rotation, and Rotation3D. Many of these object classes provide conversion methods. For example the Rotation class provides methods to convert to degrees or to radians.

An object oriented language allows the software to be written as a set of reusable components or modules. An object can be thought of as a "thing." An object may be physical thing like a robot or a logical or abstract thing like an address. An object has a set of attributes which are used to define its characteristics. An object has a set of methods that manipulate the object in some way, like

- setting the value of an attribute.
- getting the value of an attribute.
- manipulating the attributes of the object.
- controlling the physical thing represented by the object. (e.g., control the motor(s) assoicated with the elevator subsystem)

Objects are defined by a "class." There can be many objects defined with the same class. All objects in a class have the same set of attributes, although the values for those attributes will likely be different. Likewise all objects of a class have the same set of methods to manipulate the individual object.

Different objects may have similar methods with the same name. This allow for treating objects similarly even if they are in different classes. For example, many objects have an initialization method called 'init.'

Basic Java Syntax

TODO This whole section needs to be re-worked. It is desirable to have examples of each topic and possibly graphics to highlight different things.

End of Line Comment

A comment is way to provide information about code that is not executed by the program. End of line comments are used to explain the intent of a particular line of code.

```
foo = 16; // this is an end of line comment
```

Multiple Line Comment

Multiple line comments are well suited to explanations of the code that go beyond a particular line of code. It is sometime used to "comment out" a block of code that is used only for testing or that is no longer needed.

```
/* this is a multi-line comment
in a general form.
And another line */

/*
 * This is a format used to set off a block of code
 * and it can have as
 * many lines as it
 * wants
 * or needs.
 */
```

Partial Line Comment or Comment Within a Line

A comment is way to provide information about code that is not executed by the program. A partial line comment is a way to insert a comment within the line of code itself. This usage is rare because it breaks the normal flow of the code.

```
foo /* a comment within a statement */ = 19;
```

Variable or Attribute Declaration

The declaration of a variable or attribute within a method defines it name and data type and is used by the program to allocate space for it.

```
int foo; // foo is declared as an integer, but not initialized
```

Variable or Attribute Assignment

Assignment of a variable or attribute is to give it a specific value. The variable or attribute must be declared before it is assigned. A variable or attribute may be reassigned at any time, unless the variable or attribute has the **final** property.

```
foo = 19; // foo is assigned the value of 19
```

Variable or Attribute Reference Within the Scope of the Attribute

An attribute may be referenced or accessed within the scope of the attribute by just using its name in an expression.

```
fi = foo;  // foo is referenced in the assignment of fi
```

Variable or Attribute Reference Outside of the Scope of the Attribute

An attribute may be referenced or accessed outside of the scope of the attribute by just using its name in an expression.

```
fi = remote.foo;  // foo in remote module is referenced
```

Class Constructor or Code Executed with Instantiating an Object

Explanation of syntax and usage

```
class Fee {
     fee(/*may have arguments*/) { /*constructor*/
           /* constructor code block */
      {
}
```

Object Instantiation (Class Reference)

Explanation of syntax and usage

```
Fee fum = fee(); /* class fee is instantiated as fum */
```

Method Declaration

Explanation of syntax and usage

```
class MyMath { // the containing class
      double square (double in) { // square is method of type double
            return in * in;
      }
}
```

Method Reference Within Scope of Its Containing Class or Object

Explanation of syntax and usage

```
class MyMath { // the containing class
      double square (double in) { // square is method of type double
            return in * in;
      }

      double cube (double in) {
            return square( in) * in; // square is method in same class
      }
}
```

Method Reference Outside of its Containing Class or Object

Explanation of syntax and usage

```
class MyMath { // the containing class
      public double square (double in) {
            return in * in;
      }
}

MyMath math = MyMath(); // object of type MyMath

double a = math.square( 4.2); // uses method square of object math
```

Declaration of a Class Attribute

Explanation of syntax and usage

```
class Fee {
      double tax; // tax is an attribute of class Fee
}
```

Use of a "Getter" Method

Normally the attributes of a method are private to prevent them from being accesses or modified externally. A "getter" method is used to return the value of a private attribute externally.

Define it as a method within a class definition

Use it outside of the class with a object reference.

```
foo /* an int */ = 19;
```

Use of a “Setter” Method

Normally the attributes of a method are private to prevent them from being accessed or modified externally. A “setter” method is used to modify the value of a private attribute externally.

Define it

Use it

```
foo /* an int */ = 19;
```

Use of the “private” Attribute Modifier

Explanation of syntax and usage

```
foo /* an int */ = 19;
```

Use of the “public” Attribute Modifier

Explanation of syntax and usage

```
foo /* an int */ = 19;
foo /* an int */ = 19;
```

Use of the “static” Attribute Modifier

Explanation of syntax and usage

```
foo /* an int */ = 19;
```

Use of the “final” Attribute Modifier

Explanation of syntax and usage

```
foo /* an int */ = 19;
```

Use of Lambda Functions

A lambda function is a form of anonymous function or a function that can be used without formally declaring it and its name. A common use of lambda functions is to pass a function as an argument to a method. The method would used the passed function in its internal processing. This can be to access a value that would change over the life of the method, like a execute function of a command. It can also be used to perform an evaluation like the sort order for particular data types of a general sorting function.

```
foo /* an int */ = 19;
```

Use of Supplier Function

A supplier function is an abstract function that returns a value of a specified type. A method may specify a supplier functions as one of its arguments and it would execute that function when it wants a particular value for evaluation. When the method is invoked, the a function that returns the desired type is passed as an argument. This might be to read the analog value of a particular sensor.

```
foo /* an int */ = 19;
```

Use of Consumer Function

Need an example of a consumer. This might be something like a logging method for a particular type, but that doesn't seem right.

Explanation of syntax and usage

```
foo /* an int */ = 19;
```

Use of Diamond Operator for <type> Casting

Explanation of syntax and usage

- <type> type casting

```
foo /* an int */ = 19;
```

Example of a Class Definition

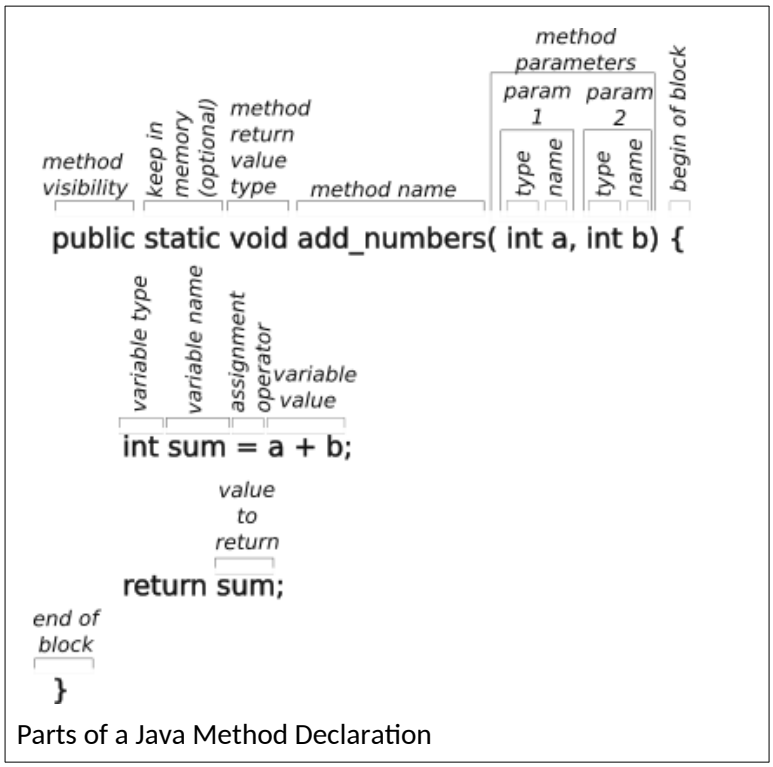
Let's work through a simple example for a sample street address object declaration.

```
Class Address (){
	// list of class attributes
	private String streetNumber; // allow numbers like '234 1/2'
	private String streetName;
	private String city;
	private String stateAbbreviation;
	private int zipCode;

	Address () { // constructor with no parameters
		streetNumber = 0;
		streetAddress = "";
		city = "";
		stateAbbreviation = ""
	}

	Address ( String number, String street, String city, String state, int zip) {
		// constructor with parameters to initialize all attributes
		street = number;
		streetName = street;
		this.city = city;
		stateAbbreviation = state;
		zipCode = zip;
	}

	public void setStreetName( String street) { // a "setter" for street name
		streetName = street;
	}

	public void getFormattedAddress() { // a "getter" for formatted address
		String zipStr = "00000" + zipCode; //pad zipCode to have lead zeros
		zipStr = zipStr.substring(zipStr.length() -5); // limit to 5 digits
		String line1 = streetNumber + " " + streetName;
		String line2 = city + " " + stateAbbreviation + " " + zipStr;
		return (line1 +"\n" + line2);
	}
}
```

Java Method

A method is a block of code that is used to manipulate an object.

An example of a method declaration is as shown to the right. All methods start with a return value type. If nothing is be returned this type is "void." If can return any type, like a boolean for a condtion that is true or false, an integer, float or double for methods that return a primative value.



Parts of a Java Method Declaration

The method return type is followed by the method name and a set of parameters enclosed in parenthesis. A method need not have parameters, which is indicated with just the open and close parenthesis. Each paramter is declared with a type and name. The scope of the name is only within the method which means that the parameter cannot be accessed outside of the method.

After the paramters is a block of code set off with an open and close brace. To be useful there are one or more statements which do something with the parameters and may or may not return a value of the type declared with the method declaration.

Mismatching the type of any of the input parameters or the return value will result in a compilation error.

Java Naming Conventions

Many programmers use naming conventions to help identify the use of an identifier by its appearance. There are a few conventions which apply to the robot code as follows:

- Class names are capitalized to set them off from object names that are lowercase, sometimes using the same name as the class. The file containing the class definition is name for its class. For example the Robot class is defined in the Robot.java file
- Files containing Java code should end with the extension '.java'
- Files containing documentation for the github.com site should use markdown formatting and have the '.md' extension.
- Directories use lower case names.
- Names for variables, methods, attributes, classes, objects are usually descriptive and use camel case (start with lower case (except class names) and begin each new word with a capital letter. Names cannot contain spaces. Underscores are rarely used except to separate words that are capitalized, like WPILIB really should be WPI_LIB to separate the two words, but this was their design choice and it is pervasive.

Coding or Style Conventions

OK, now for the controversial side of coding. Everyone has a style that they like and they basically hate everyone else's style. If you are modifying existing code, it is generally a good idea to mimic the style that is there. The purpose of style conventions is to make your code easier to read.

Here are some rules that I like (that may or may not be reflected in the robot code):

- Use meaningful names so that you don't have to guess or remember its use.
- Similarly longer names that refect their usage is better than short names like 'a' or 'b'.
- Add a space where you would pause while reading the statement out loud.
  - e.g., j=a+b should be written j = a + b.
- Add a space after an opening parenthesis when the parenthesis is really part of a name.
  - e.g., function(a) should be function(a), the parenthesis is part of 'function'
  - e.g., 'if(a<b)' should be 'if (a < b)', the parenthesis is used for grouping and is not part of a name.
- Conditionals should ALWAYS be written the same way with the blocks indented, so you can quickly recognize the design pattern and don't have to think.

  For example, the following overly terse code:

```
if (foo) return true;return false;
```

  should be written in the more regular design pattern as:

```
if (foo) {
	return true;
} else {
	return false;
}
```

- Do not use the ? : operator as also breaks the design pattern which may help to hide bugs.

Don't waste a lot of time with style wars. It is somewhat personal, but it is important to adapt to the organization so that it is easy to read each others code. Some organizations use 'lint' utilities to enforce their local coding style guide. Messy code shows a lack of thought and care and increases the probability of hiding problems. Code that is straight forward and easy to read is usually more reliable and is much easier to maintain.

Robot Procedures

Swerve Drive Alignment Procedure

Follow the following steps:

1. Manually turn each drive module so that the gear is on the right hand side.
2. Align the two wheels on each side using a bar or piece of lumber.
3. Note the angle of each wheel
4. Put the angles found in step 3 in the swerve configuration file.

Commissioning Procedure

Fill out the basic steps of commissioning a robot TODO

- CTRE Tuner X for accessing CAN FD Bus devices and CTR devices like the Pigeon. It is also used to upgrade the software on devices with CTR controllers.
- CTRE Phoenix Tuner
- REV Hardware Client

set the address of devices on the CAN bus

roboRIO CAN bus ID should be 0

roboRIO CAN bus ID should be 1

resolve any conflicts

record the can bus address in the appropriate robot config file(s)

Match Procedure(s)

Before every match there is a list of things that need to be checked and re-checked. Different members of the team have different roles and different things that they are responsible for. Each team should develop its own checklist for its own team members.

- Drive coach
  - work with alliance partners to learn their strategies and your team's roles to implement those strategies. The plural form is used to allow a strategy to change in mid match if necessary.
  - make sure other drive team members know their roles and match strategy.
  - Don't forget defense. The team with the best defense is often the winning team.
- Driver
- Co Driver
- Technician
  - Make sure robot has a fresh battery and that it is connected and strapped in.
- Human Player

- Scouts
- Cheer section
- Mentors
    - Do not lose kids as parents may become quite upset.
    - Make sure kids are fed.

Ideas

Collision Detection

The navigation subsystem will detect collisions. You can take the impact vector and subtract the last known robot momentum ($E = mv^2$) vector to find the direction of the colliding robot. For automatic bump and run…check to see if the robot is clear of game structures, if it is, roll away from the colliding robot.

Defensive Rolling

The bump and run roll holds a corner of the robot against the other robot. The center of the robot will trace a scallop curve with the radius of the center of the robot rotating around a corner until another corner makes contact with the opposing robot. If the movement is to the left of the other robot, the spin is clockwise, otherwise it is counter clockwise. The rate of the spin is to keep the corners of the robot stationary. The swerve drive chain provides for the translation of the center of rotation to allow this.

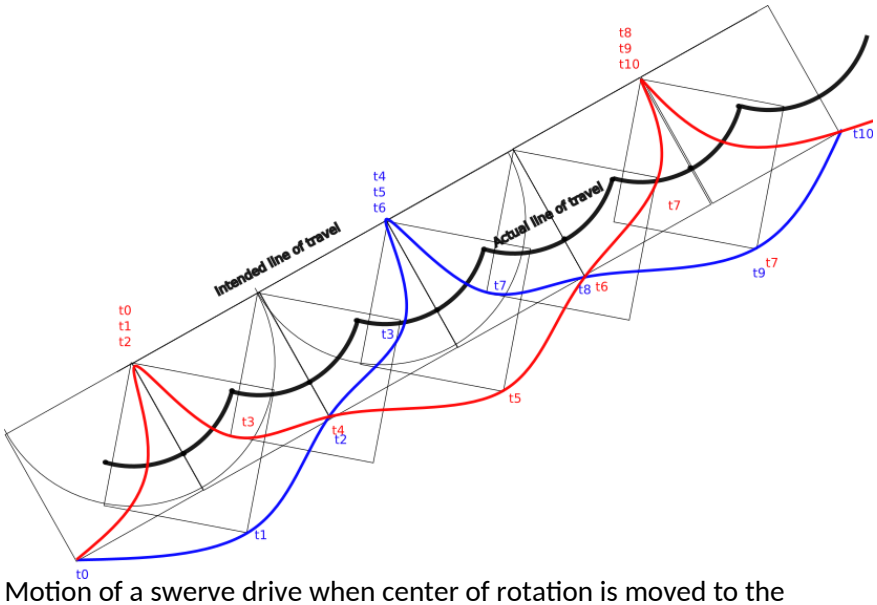
Logging Data for Analysis

Should turn on the logger code. This should be used to output timestamped pose estimates which can be used to create a heat map to point out where too much time is being spent. It can also more accurately report instantaneous velocities. It can also point to wheel slippage and other problems. It can help identify total cycle times and components of cycle times to point out areas for improvement.

Defensive Spinning

In the figure to the fight. A robot is spun while moving forward, but the point of rotation is moved to the upper forward corner of the robot. This allows the robot to move in a straight line with respect to its side. This maneuver may be useful for defense as it does not “give up” any territory while spinning and moving forward. Again speed is sacrificed to spin while moving.

The path of two corners is shown, one in red and one in blue. The times for each corner is shown independently one corner will stop and the robot rotates about that corner.



Motion of a swerve drive when center of rotation is moved to the forward right corner and swung through 90 degrees for each corner of the robot. (Heavy black line traces the movement of the robot center. The blue line traces one wheel and the red line traces the wheel on the opposite corner. The light black lines show the robot positions at various times.

Practice Fencing

Use the odometry data to disable the robot if it goes out of bounds of the practice field to prevent injury to students, robot, furniture, walls, etc. It also gets this subsystem more of a workout to make sure that it is more reliable and more useful.

Vocabulary

argument a positional parameter passed to a method, function or subroutine

attribute a variable associated with the instance of an object.

block a group of statements bound by curly braces '{...}'

CAN bus is the Controller Area Network used for automobiles and FIRST robots.

controller a device that controls something. For example, this can be the software that controls a motor.

constructor a method of a class that is used to declare and initialize the attributes of an object of that class.

co-pilot is the human who controls accessories on the robot. Also called the **operator**.

declare define the name and type of a variable or attribute.

decorator modifies the behaviours of object instances of a class at runtime without affecting other objects of the same class.

driver the human driving the robot and the same as a pilot. The low level code used to control a piece of hardware.

encapsulation allows names to be reused without conflict in different modules. Part of this is controlled with the limited scope of an attribute or method, but these can still be accessed by specifying the containing object name, a separating dot or period, and the attribute or method name.

FMS is the Field Management System which allows the field referees to control robots for simultaneous starts and safety.

function is a routine tha retucorns one or more values.

heading is the angle that the robot is moving toward with respect to the field.

inheritance a class can inherit the attributes and methods of a parent by extending a class. The new class can add additional attributes and methods. This is done with the Java keyword **extends**.

instantiation the creation of an object from a class template.

initialize set the initial value of a variable or attribute.

Lime Light is the brand name of a FIRST robotics vision co-processor.

LimeLightOS is the brand name of a proprietary vision processing system for detecting and processing April Tags and other objects.

operator is the human who controls accessories on the robot. Also called the co-pilot.

method a function or subroutine that performs an operation on the containing object. Method names are signaled by as set of parentheses which may be empty or contain a set of arguments.

multithreading is a service offered by some operating systems, like Linux, that allow a program to run as separate threads in separate processes either on the same processor or on a different processor or core. The roboRIO has two cores, so it can do processing in both cores simultaneously.

navX of **navX2** is the brand name of a FIRST robotics navigation co-processor.

operator is the human who controls accessories on the robot. Also called the co-pilot.

overloading is using the same method name with different signatures to do essentially the same thing but with different arguments.

parameter see argument

PID proportional-integrated-differential control, a mathematical procedure to control a system to smoothly and quickly reach a desired target position or velocity.

pilot is the human driver of the robot.

pitch is the (front-to-vertical) angle of rotation of the robot about the x-axis with respect to initial inertial vertical angle. (see yaw.)

pointing or **pose** is the angle of the robot with respect to the field.

polymorphism the ability to overload a method with different signatures to do essentially the same thing with different data types.

private accessible only inside of the object

public accessible outside of the object

PWM is Pulse Width Modulation which uses a timed wave form to dynamically change the amount of power delivered to a device.

roboRIO is the primary robot controller required by FIRST.

roll is the (side-to-vertical) angle of rotation of the robot about the y-axis with respect to the initial inertial vertical angle. (See yaw.)

RSL is the Robot Status Light which communicates the Field Management Status of the robot.

scope is where an object, variable, or parameter may be accessed. In Java scope is normally confined to the containing **block** set off by a pair of curly braces ("{" and "}"). For example this can be an object, a method within an object, an attribute within an object, a variable within a "for loop" a method.

semaphore is a mechanism that is locked and unlocked by a thread to temporily prevent read access to a variable or block of memory.

shuffleboard is the display of telemetry data on the drive station.

signature is the returned primitive data types or classes and the primitive data types or classes of the passed arguments. For example:

> int foo(int a, int b) is the same signature as int foo (int c, int d)
>
> (int, int, int) = (int, int, int)
>
> int foo(float a, float b) is a different signature than int foo(int a, int b)
>
> (int, float, float) != (int, int, int)

static retained in memory. It is not necessary to create an object for a static class.

subroutine a routine that does not return a value.

thread safe is a technique to guarantee that a thread of a process will not read certain values that has been partially written by another thread of the same process. This uses semaphores to lock and unlock read access to such values when they are being written.

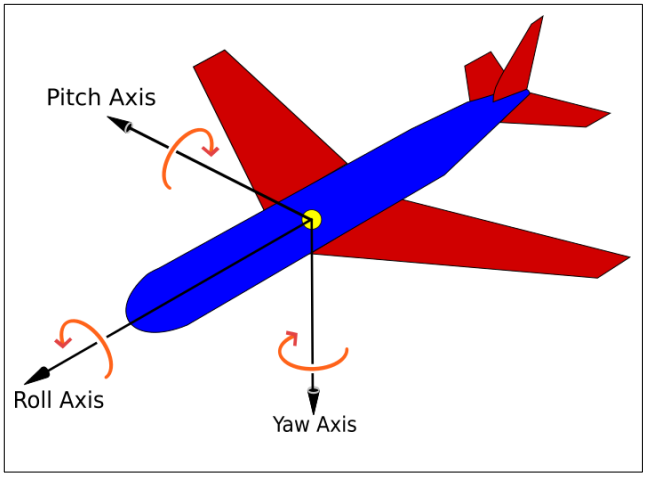
type the primitive type (integer (int), float, double, boolean, char) or class (String or a locally defined class)

variable a named value with a specified type

WPILIB is the Wooster Polytechnic Institute (WPI) library for FIRST robotics

yaw is the angle of the robot (really inertial measurement device) about the z-axis and the initial inertial angle.

TODO Stuff to work in



Command Chaining

WPILIB allows for command chaining. This allows single command to actually run multiple commands and may command different subsystems. For instance a command to prepare intake may:

- Run the elevator to a predetermined height
- Move the arm to a predetermined angle
- Start running the intake motor

These commands may be executed sequentially or in parallel based on the constraints of the robot. Team 2910 has had command chains that defy understanding, but work well in practice.

Path and Trajectory Planning

path: a sequence of poses for moving the robot. In general make this as short and smooth as possible.

trajectory: a timed sequence of posing for moving the robot at differing velocities.

Init and follow: a path that also initializes the robot pose to a known field location and orientation before following the path where it goes.

follow: a path that defines where the robot is to go.

Use with autonomous routines. It is tied to a sequence of commands something like:

- Move to a position smoothly
- Aim shooter
- Bring shooter up to speed
- shoot
- End shooting
- Move to next piece
- pickup piece
- stop
- Move to shooting position
- bring shooter up to speed
- shooter

- end shooting

use with teleop: This is more like move to load station, move to shooting position. It is used to assist driving to a specific location and orientation. The driver may override the path to avoid other robots. To be useful in teleop, the robot must have good odometry so that its knows its location precisely to get the destination quickly and accurately without hitting obstacles. Paths may require way points to get around fixed obstacles.

General Strategies

- Know your competitors and alliance team mates
    - Qualitative measures
        - Consistency
        - ways to score
        - reliability
        - ability to work well as an alliance member
        - leader/follower/not a team player
        - play innovation
    - Quantitative measures
        - scores of each time
        - cycle time
    - Scout
        - do by hand
        - use a scouting tool
        - Make results easy to use as alliance captain
- Score
    - raw score counts and helps gain ranking points (usually 2 for winning)
    - reduce cycle time
        - reduce unnecessary movement
        - reduce waiting
        - score at a distance
        - pass game pieces to alliance members
        - have a lean mean machine
        - work for faster builds and repairs to be quicker in the lab and in the pits
        - work for faster software development for more improvements
- Defense
    - always be ready and willing to play defense

    - interfere with others ability to move
    - interfere with others ability to score
    - know the rules!
- Speed
    - use path planning where possible to increase robot speed
    - Do not spin unnecessarily
    - Do not collide unnecessarily
- Match Negotiations
    - Don’t over look cooperation to increase scoring in competitions
    - Establish lanes and roles to avoid interfering with partners
    - Assign and keep task assignments
    - take direction from alliance teams
- Constant Incremental Improvement
- Compete at the team’s level of competence
- What ever it takes
    - do grunt work even if distasteful
    - get the job done
    - do it quickly and without hesitation
    - applies to everything… build, clean up, driving, defense, loading in and out, being there
    - watch other teams to find out what they are doing (strategies, design, etc.)
- Fail Fast—Fail Often
    - Write code so that it does not mask problems, better to let it fail and be corrected.
    - If something isn’t quite righat, fail, don’t limp along.

Alphabetical Index