F1TENTH MANUAL

UCSB Capstone 2025-2026

What is F1TENTH?

The F1TENTH community project began at UPenn in 2016 as a way to foster interest in autonomous systems by teaching students how to convert typical RC cars into self-driving racing robots. The name F1TENTH refers to how the RC cars used in the project are always one-tenth the size of Formula 1 race cars. Since 2016, F1TENTH has blossomed into an international community of researchers, engineers, and autonomous systems enthusiasts who meet annually to compete their cars and test their racing algorithms. By reading F1TENTH's educational website, anyone can learn the step-by-step process for converting a typical RC car into an autonomous racing robot and get involved in the F1TENTH community.

Your team's goal is to convert a one-tenth scale RC car into a robot that can autonomously drive around the inside of a walled track using an original racing algorithm. This document serves as an entry-level guide to help you initialize your hardware and software systems, learn some basic concepts in robotics, and understand the scope of the task at hand. Additionally, the "Learn" and "Research" pages on the F1TENTH website are valuable resources for learning the fundamentals of autonomous systems and gathering ideas for the design of your original algorithm.

In this project, your team must learn how to code in ROS. It is common for capstone students to have never worked with ROS before, so in this project you will likely spend most of your time troubleshooting and learning how to work with this coding style. Fortunately, ROS is an industry standard, so it is valuable to learn if you hope to pursue robotics down the road.

Hardware Parts List

The following table provides links to the needed hardware components for this project. Most components can be ordered through Amazon. Follow the F1TENTH build manual for assembly instructions.

Part	Cost	Link
Chassis Hardware		
1/10 Scale, 4x4 RC Car	$300-$500	Traxxas Slash VXL
Battery and Charger	$200	Two Lipos and charger
Platform Deck	$20	(Insert link to Kyle's design)
Antenna	$19	Wifi antenna and antenna mount
Kits and Fasteners	$70	M5 and M3. Fastener BOM
Adapters	$34	TRX to XT90 , battery charger adapter , bullet adapter
Powerboard	$80	Materials
Computing		
Nvidia Jetson Orin Nano*	$500	Developer Kit
SSD	$50	500GB
Sensors		
LiDAR	$200-$600	RPLiDAR
VESC	$260	VESC Make 6 and its VESC tool and PPM cable
Camera	$45	C270, C920,
Total	**~ $2100**	

*Previous years of F1TENTH projects have succeeded by using either a Raspberry Pi or a Jetson as the central processing unit for the car. We suggest that you use the Jetson, but if you already have proficiency with the Raspberry Pi you can use either option.

Software Setup (Preparing the Jetson)

In this project, you will be coding in ROS. The following sections offer a general overview of how you will be using ROS to control your car. However, you are highly encouraged to research ROS independently outside of what is presented in this document. Teams should expect to have a strong final product if they can allow for several group members to gain experience reading, writing, and troubleshooting ROS code by the end of winter quarter.

ROS is run on Ubuntu, which is a distribution of the Linux operating system. You will first need to install the Ubuntu desktop onto the Jetson. From there, you can install ROS using the desktop terminal. However, there are several distributions of ROS. Each distribution runs on a specific release of Ubuntu, so you need to decide which ROS distribution you will use before installing the desktop. We recommend you use ROS Noetic, which runs on Ubuntu 20.04. You can also use ROS Melodic, which runs on Ubuntu 18.04. These were the latest distributions of ROS before the release of ROS2, which is more modern but has less documentation in online discussions to aid with your troubleshooting. If your team has almost no experience with ROS, it is probably a smarter decision for your team to use ROS1 instead of a ROS2.

The YouTube channel JetsonHacks is a good general resource for learning how to work with the Jetson. It has a tutorial explaining how to install ROS Melodic onto the Jetson. It also has a tutorial explaining how to install Ubuntu 20.04 onto the Jetson using your SSD. Once you have Ubuntu 20.04 installed, you can install ROS Noetic by following this tutorial.

Before using the ROS environment, you need to source certain setup.sh files in the terminal. We recommend adding a command at the end of your terminal's startup script so that you do not need to "source" every time you want to work on your project.

Github and File System Structure

Github allows your team to keep your code in a repository, which is a convenient place to store and share your team's code online. Moreover, Github allows you to set up a folder on your local machine that can update its contents to match your team's online repository using commands from the terminal. By periodically updating this folder, the code on your local machine can stay up to date with your

teammates' recent work. Setting up a folder that connects to an online repository is called “cloning” a repository. The process of updating your local folder to match the contents of the repository is called “pulling” and the process of updating your repository to match the contents of your local folder is called “pushing.”

To get started, create a new blank Github repository with a README file and clone it onto the Jetson. This folder/directory is referred to as the “root” of your “catkin workspace” and will house all the code for your project. Let’s assume the name of your workspace/repository is “f1tenth_starter_code.” On the Jetson, add these four empty folders to your file system:

```
None
f1tenth_starter_code/
|    src/
|    |    pathing_package/
|    |    |
|    |    lidar_package/
|    |    |
|    |    vesc_packages/
|    |    |
|    README.md
```

Next, push your edits to your online repository by executing the following commands in the terminal:

1. git add .
2. git commit -m ‘message for the team’
3. git push

If git push asks for a username and password, then the password you type in must be a Personal Access Token. You can get a PAT by going to github.com and going to Settings (upper right) -> Developer Settings (scroll all the way down on the left tabs) -> Personal access tokens -> Tokens (classic). Once you have a PAT you can

use it as your password and edit your git credentials on the Jetson so that you do not need to find your PAT every time you want to push to the repo.

Now put some starter code into your repository. Copy the contents of this repository into the “pathing_package” folder of your catkin workspace. You can do this by cloning the repository to the Jetson, extracting all the files, and deleting the cloned folder. Then, compile the code by running the command “catkin_make” in the terminal from the root directory of your catkin workspace. Your file system should now look like this:

```
None
f1tenth_starter_code/ #root of the catkin_workspace
|    build/
|    devel/
|    src/ #contains all your packages
|    |    pathing_package/ #has "f1tenth_simulator" code
|    |    |    CMakeLists.txt
|    |    |    package.xml
|    |    |    # other files
|    |    lidar_package/
|    |    vesc_package/
|    |    CMakeLists.txt #helps compile your code
```

The catkin_make command automatically creates the “build” and “devel” folders if they are not already in the root directory. It also runs all the “CMakeLists.txt” files in the workspace. You can read more about catkin workspaces from the ROS wiki.

Packages in ROS always have a “CMakeLists.txt” file and a “package.xml” file. Packages contain completed, runnable ROS code—the f1tenth_simulator repository is an example of an open source package. Your team should work with the simulator on the Jetson to become comfortable with the F1TENTH ROS system while your car is still in the process of being built. The simulator might also be

useful later in the year if you want to speed up the testing process of your racing algorithm.

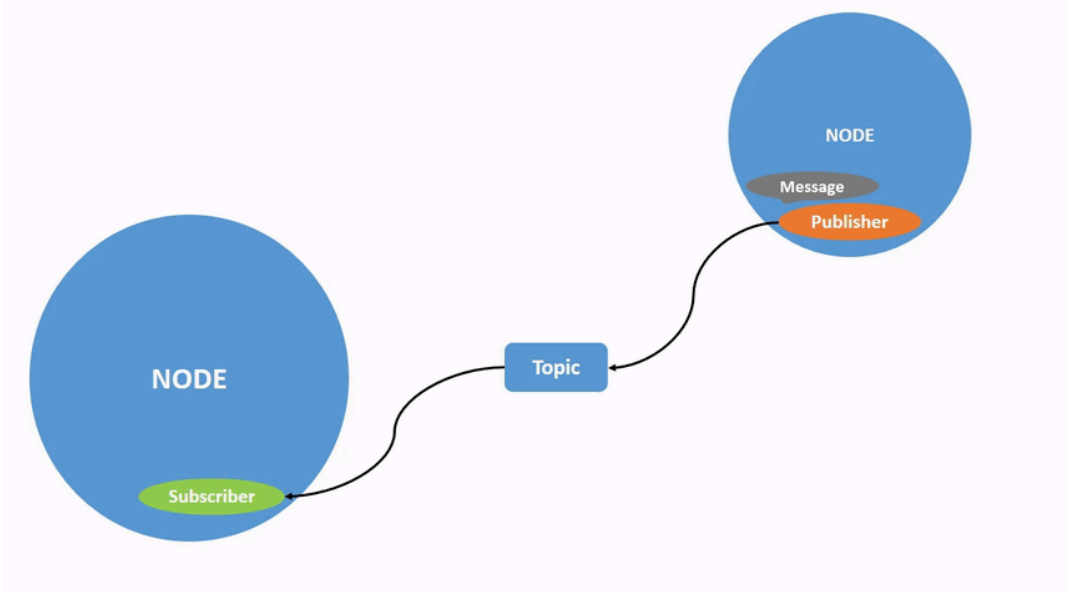
By the end of fall quarter, each member of your team should be comfortable with reading the files contained in the f1tenth_simulator package. Once the hardware assembly of the car is complete, your team can integrate the code onto the car by replacing the topics with simulated data with the real-time data taken from your physical sensors. Finally, your team will add nodes to the “pathing_package” in order to implement your own original racing algorithm.

Carrying out the steps of creating your own Github repository is an important skill to learn in this project. However, we offer a template github repository which should help you understand how your packages should be organized when using ROS.

The following section explains basic ROS concepts and how they are used in the F1TENTH project.

ROS Basics

ROS stands for Robot Operating System, but it is best understood as a software framework. Conceptually, think of ROS as an organizational tool that breaks down the many processes of your code into smaller distinct parts. We call these distinct parts **nodes**. Nodes send and receive **messages** to each other over pathways called **topics**.



A node is an executable piece of code that reads input messages and writes output messages. Your code in ROS is essentially a network of nodes connected by topics. When you create a node, you specify which topics it is "subscribed" to and which topics it is "publishing" to. By defining the subscriptions and publications of each node, you can control the flow of data in your network. This method of coding is useful for creating continuous loops, which is necessary for the design of autonomous systems.

The definition of a node is very general because you can write nodes to be used for different tasks. In this project, nodes can be used to either (1) collect sensor data, (2) send commands to the VESC, or (3) implement algorithms. In this document, we refer to these as **sensor nodes**, **controller nodes**, and the rest as part of the **network**. In any case, the basic input/output functionality of a node does not change.

Messages sent across topics must be formatted in some way. **Message types** define formats for handling data. When you create a topic, you must specify its name and its message type. Fortunately, ROS has several open source packages that define all kinds of useful message types. The simplest examples of message types are contained in the ROS std_msgs package.

In this project, your LiDAR and your VESC are your two main sensors. In ROS, LiDAR data is sent across topics using the LaserScan message type, which comes from the sensor_msgs package. The VESC acts as a sensor because it stores the history of your car's wheel movement. This **odometry** data can be used to help the car keep track of where it is oriented relative to its starting position. As a side note, the VESC also acts as a controller, as it can send electrical commands to the motors controlling the steering and acceleration of the car.

In general, before you order a hardware sensor for a given ROS project, you should always make sure there is an open source Github repository containing a ROS package associated with that sensor. A ROS package associated with a specific hardware sensor has three main purposes: (1) it ensures that there is a message type that can format the data taken from the sensor, (2) it provides a node that takes input from the device's USB port on the Jetson and publishes that data to a new topic, and (3) it gives the user a way to run the node from a terminal. See the rplidar_ros package as an example. To implement this code into your ROS network, all you need to do is clone the rplidar_ros package into the src folder of your catkin workspace. In the previous section, we suggested you to put the empty folder "lidar_package" into your src directory as a placeholder for the rplidar_ros package.

However, the folder containing the rplidar_ros package can be named anything. It is the “package.xml” file that specifies the name of a package–not the title of the directory. To be clear, when you see a package.xml file in your file system, it means that the current directory you are in is a ROS package.

The node run by the rplidar_ros package is an example of a **sensor node**. This type of node gathers input by reading the data received at a specific USB port on the Jetson. It then formats that data according to a specified message type and publishes the message to a specified ROS topic. Because many sensors take measurements at high rates, these nodes also publish messages at high rates. This means that topics containing sensor data are constantly being updated with new messages.

To understand how data is actually handled by nodes, you need to know the basics of how a node’s code is structured. In C++, a node is set up like a class. In the node’s constructor, you can create subscriptions and publications using built-in ROS functions. To create a subscription, you just need to specify (1) the name of the topic you want the node to subscribe to and (2) the name of a new function that will run every time a message is published to that topic. This means that as soon as a message is published to a topic, that message is instantly handled by any nodes subscribed to that topic using these functions. These functions are referred to as **callback functions**. To create a publication, you need to specify (1) the name of the topic you want the node to publish to and (2) the message type associated with that topic.

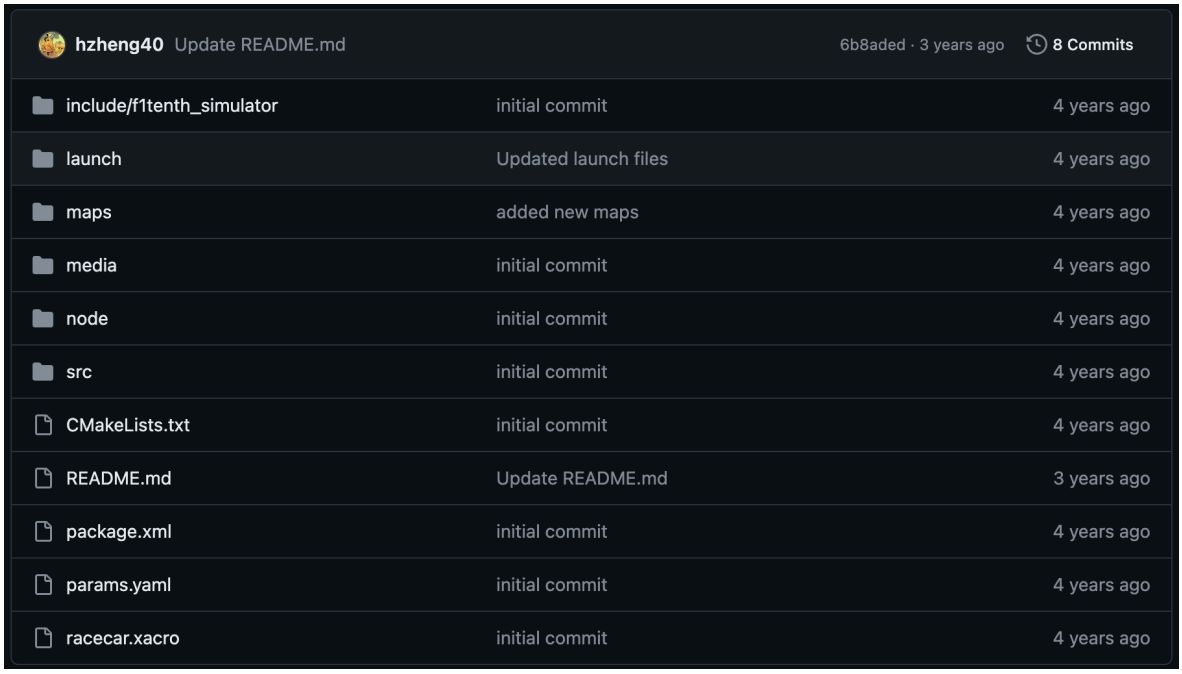
Callback functions are generally set up to create new messages from scratch and publish them to output topics using the node’s publications. This means that nodes generally publish messages as soon as they receive one through a subscription. As a result, data flows incredibly quickly through a ROS network. Think of your software system as a cyclical process. Your sensor nodes are at the start of the cycle and publish the sensors’ data to their topics at an extremely high rate. The nodes in your network subscribe to these sensor topics and try to process the data just as fast. This means that the rest of the topics in your network also update extremely quickly, as each node generally tries to publish new messages as fast as it receives them. This functionality allows your car to make quick decisions based on sudden changes in its environment. At the end of the cycle, your **controller node network** reads the message at the end of your network, converts it to a form the VESC can understand, and sends it to the VESC. You can implement this network by cloning the VESC packages into the src folder of your catkin workspace. This allows you to control the steering and acceleration of the car using ROS topics. As

the car moves, the sensor data is updated and the continuous cycle continues. F1TENTH autonomous algorithms are realized by writing nodes that can intelligently send commands to the VESC after processing the sensor data taken from the LiDAR and the VESC.

Finally, ROS is also useful because it comes with the visualization software RVIZ. RVIZ knows how to read messages across any topic and visualize the data in three dimensions. However, in order to visualize the data across multiple topics at the same time, the data in each message must have some reference coordinate system associated with it. In ROS, TF allows you to specify the position and orientation of each sensor relative to the coordinate system of the car. TF then creates new coordinate systems at these points and remembers how they are related to one another. In this way, TF helps you make sense of your sensor data geometrically. In other words, as long as you specify at the top of each message which coordinate system the data is coming from, TF can help RVIZ visualize the data properly. These coordinate systems are called **frames**, and you will find them specified in the headers of most messages.

ROS in F1TENTH

This section will go over the structure of the f1tenth_simulator package and explain how to run the code from the terminal.



As always, this package contains a CMakeLists.txt file and a package.xml file. It also contains a launch folder, which is common for ROS packages. Launch files are used to run code. An example launch file is shown below.

f1tenth_simulator / launch / simulator.launch

kimnluong Updated launch files

Code Blame 47 lines (37 loc) · 2.02 KB

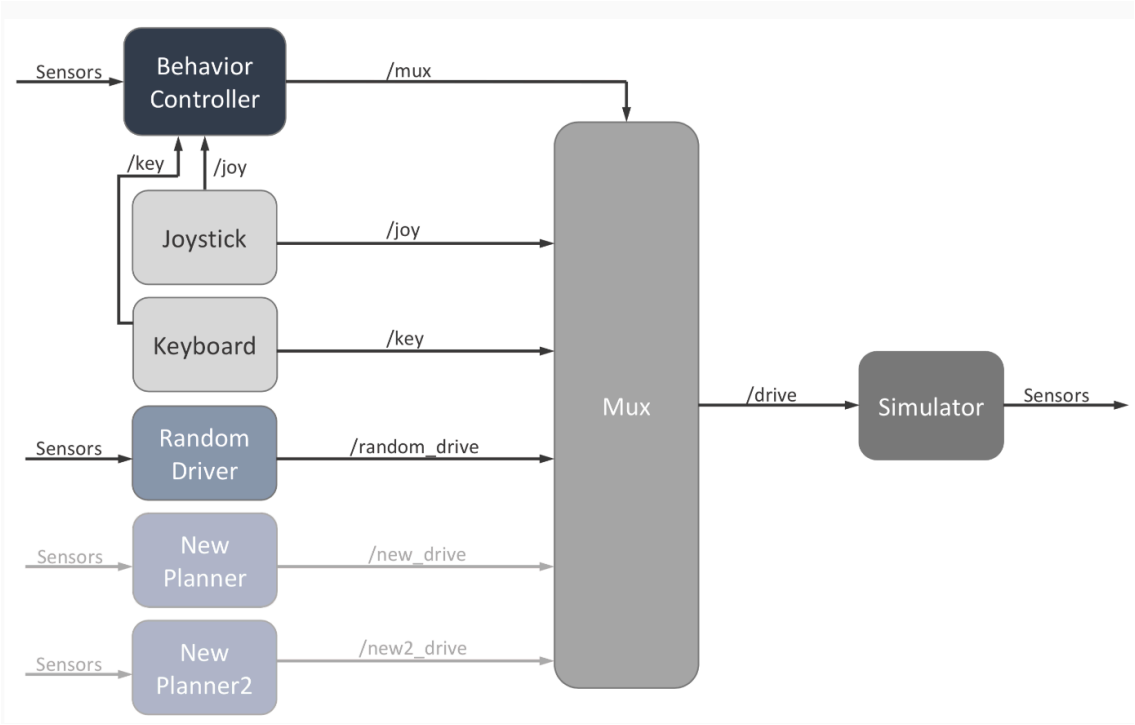
```
1  <?xml version="1.0"?>
2  <launch>
3    <!-- Listen to messages from joysicks -->
4    <node pkg="joy" name="joy_node" type="joy_node"/>
5
6    <!-- Launch a map from the maps folder-->
7    <arg name="map" default="$(find f1tenth_simulator)/maps/levine.yaml"/>
8    <node pkg="map_server" name="map_server" type="map_server" args="$(arg map)"/>
9
10   <!-- Launch the racecar model -->
11   <include file="$(find f1tenth_simulator)/launch/racecar_model.launch"/>
12
13   <!-- Begin the simulator with the parameters from params.yaml -->
14   <node pkg="f1tenth_simulator" name="f1tenth_simulator" type="simulator" output="screen">
15     <rosparam command="load" file="$(find f1tenth_simulator)/params.yaml"/>
16   </node>
17
18   <!-- Launch the mux node with the parameters from params.yaml -->
19   <node pkg="f1tenth_simulator" name="mux_controller" type="mux" output="screen">
20     <rosparam command="load" file="$(find f1tenth_simulator)/params.yaml"/>
21   </node>
```

From the command line, you can type “roslaunch <package_name> <file.launch>” to run a launch file. In this case, you would type “roslaunch f1tenth_simulator simulator.launch”.

The node tag is used to run specific node files. The *type* refers to the name of the executable created in the package’s CMakeLists.txt file. Typically the type is just the name of the file without its extension. If you read the CMakeLists.txt file, you will see that an executable is created in this way for each file in the node folder. The include tag imports an entire xml file and runs it. Include tags are useful if you want one launch file to launch multiple other launch files. The arg tag is used to explicitly pass parameters between launch files. When you use an arg tag, you can specify the argument’s value when you run the launch file from the command line. For example, you could execute “roslaunch f1tenth_simulator simulator.launch

map:=maps/berlin.yaml" from the f1tenth_simulator directory in the terminal to start the simulator with a different map setting. It should be noted that the command "$(find <package>)" returns the name of the directory containing that package. The name of a package is determined by its package.xml file.

The image below shows the layout of the simulator's node network. If you run the code from the f1tenth_simulator package, executing the command "rqt_graph" in a new terminal will bring up a similar graphic.



Each block represents a node and each word in front of a forward slash represents the name of a topic. The network is explained here.

In the simulator, a fake map is loaded into RVIZ using the map_server node and the simulator.cpp node publishes fake LiDAR, VESC, and TF data to sensor topics. Your group members should become comfortable reading and running the code for the simulator by the end of fall quarter.

Once the car is built, you will need to implement sensor nodes for the hardware components on your car. At this point, you should look into the f1tenth_system repository. Once you have successfully published the LiDAR, VESC, and TF data, you can stop launching the simulator file and essentially replace all the simulated data in the sensor topics with real data from the sensors.

You will find that there are other packages you need to have installed in your ROS environment in order to run the simulator package. These packages are called **dependencies**. The list of dependencies for a given package can be found in the package's package.xml file. Dependencies can usually be installed from the terminal. When you install a package by cloning the package's Github repository into your src folder, it is called "installing from source." Generally you do not need to install from source.

Moving Forward

There are plenty more ROS concepts to research that will help you design your racing algorithm. We have listed some of them below.

- The Wall Follow algorithm
- The Gap Follow algorithm
- The TF tree and associated commands
- The SLAM package
- The Pure Pursuit algorithm using waypoints
- The Navigation Stack package
- The F1TENTH research page
- The F1TENTH learn page
- The gpio_control package
- Packages relating to SLAM map analysis
- Image processing methods for identifying certain racing situations using the camera
- gitignore
- Ros graphs (the "rqt_graph" command and the "rqt &" command)

Throughout the year, teams will find that it is often hard to have every group member work on the code at the same time. If someone in your group has no immediate tasks, it is a good idea to have them read through some of these topics and present them to the group at the next weekly meeting.

Extra Comments

At some point, you will have multiple devices using the USB ports on the Jetson. In order for each of your sensor nodes to know which port to read data from, you will need to set up Udev rules on the Jetson.

In order to control the car remotely while it is driving, each team member will need to install some form of VNC onto their laptops. VNC allows you to control the Ubuntu desktop wirelessly, but it only works if the Jetson and your laptop are connected to the same Wifi network. Someone in your group will need to look into this.

Helpful Terminal Commands

Task	Command
General CLI Commands	
Update a Github repository	cd {repository}
	git add .
	git commit -m ‘message for team’
	git push
Update files to match a Github repository	git pull
Update	sudo apt update
Upgrade	sudo apt upgrade
Change into directory	cd {directory_name}
List available directories	ls
Make directory	mkdir {directory_name}
Move back a directory	cd ..
Remove file	rm (file}
Run as superuser	sudo
Edit file	sudo nano {file}
Kill process	Ctrl+C
List USB connections	lsusb
Get ip address	hostname -I
Get all available network interfaces	ip link
Reload udev rules	sudo udevadm control –reload-rules
	sudo udevadm service restart
	sudo udevadm trigger
Get info about a port name	udevadm info -q all -a -n /dev/<port>
Reboot	sudo reboot
Activate virtual environment for web app	source venv/bin/activate

Deactivate virtual environment for web app	deactivate
Run flask app	flask run
Git pull returns fatal error (loose object is corrupt)	cp -a .git .git-old
	cd .git/ && find . -type f -empty -delete -print
	git fsck --full && cd .. && git status
Stop needing to enter git credentials	git config --global credential.helper store
Clear current edits and reset to repo	git fetch origin
	git reset --hard origin/master
Error code 127 for python files	dos2unix PATH
Error code 11 (segmentation fault)	Find error by adding launch-prefix="xterm -e gdb --args"to <node> tag in launch file
Make all python files executable	find . -name “*.py” -exec chmod +x {} \;
General ROS CLI Commands	
Install package	sudo apt install {package}
List active nodes/topics	rosnode/rostopic list
Get info about a node/topic	rosnode/rostopic info {node/topic}
See data getting passed through a topic	rostopic echo {topic}
See active nodes and topics graphically	rqt_graph
Launch all nodes	roslaunch velma velma.launch
Launch lidar driver	roslaunch rplidar_ros view_rplidar_s2.launch
Launch pixy driver	roslaunch pixy2_node pixy2.launch
Launch simulator driver	roslaunch f1tenth_simulator simulator.launch
Start pigpio daemon to run gpio nodes	sudo pigpiod
Run gpio input driver	rosrun gpio_control gpio_control --device pi --input {pin_num}
Run gpio output driver	rosrun gpio_control gpio_control --device pi --output {pin_num}
Run gpio driver for multiple pins	rosrun gpio_control gpio_control_node --device pi --output {pin} {pin} {pin} --input {pin} {pin} {pin}
Launch vesc and lidar	roslaunch racecar teleop.launch

Three ways to view tf tree	rosrun tf view_frames && evince frames.pdf
	rosrun rqt_tf_tree rqt_tf_tree
	rqt &