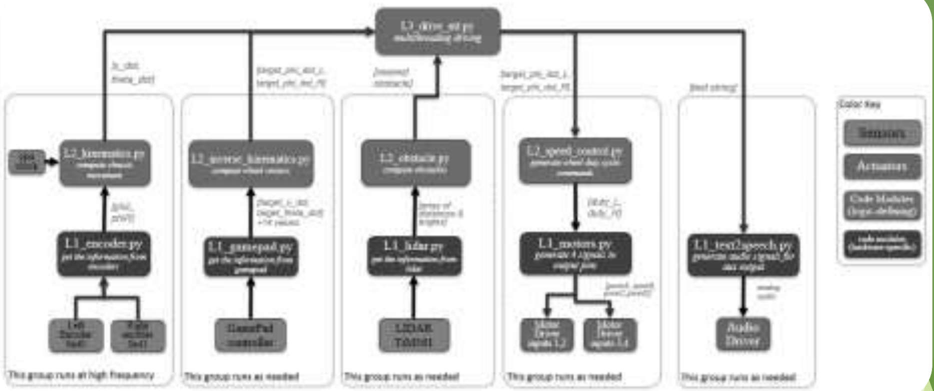
Scuttle Software Guide, Kinematics & Obstacle Avoidance

revised 2020.09.03

Software Architecture

This guide covers:

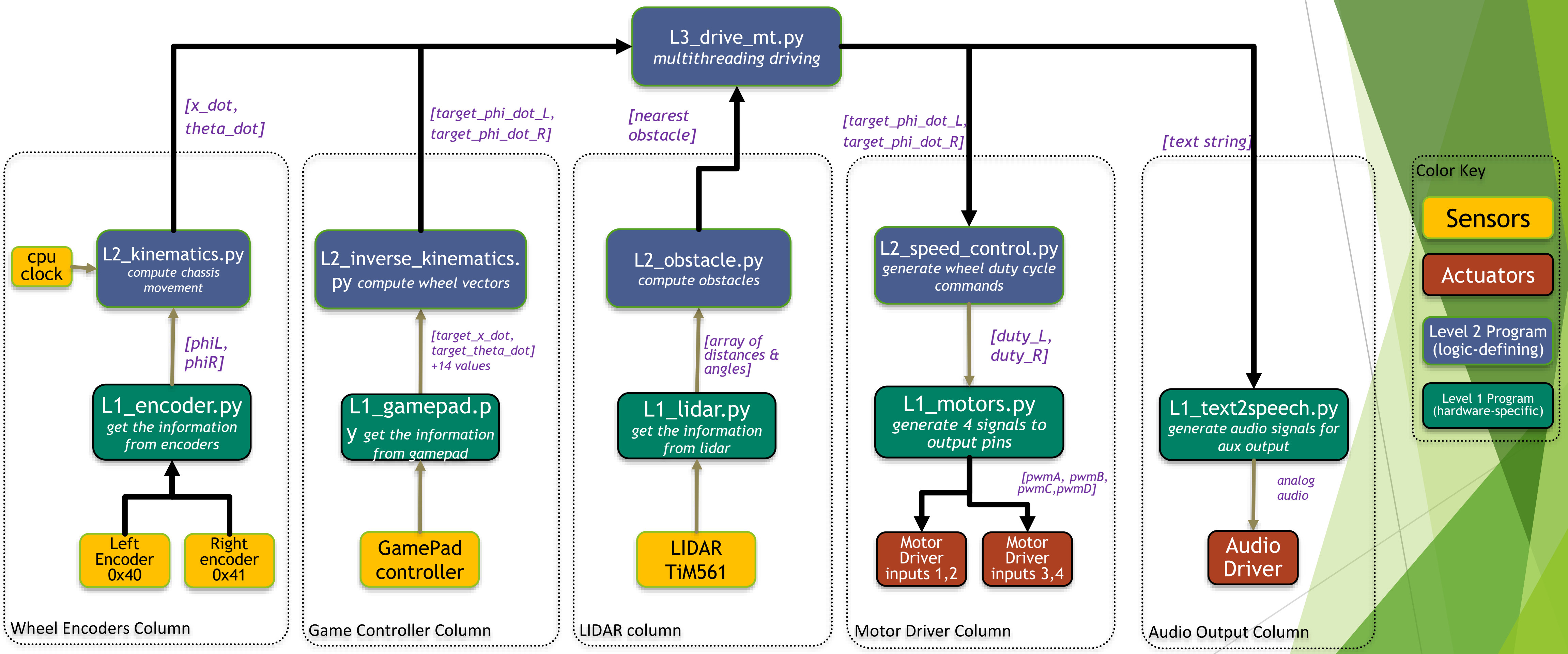
- The parts of each software file
- How the programs interact with each other
- How the programs interact with hardware
- Sensor software vs actuator software



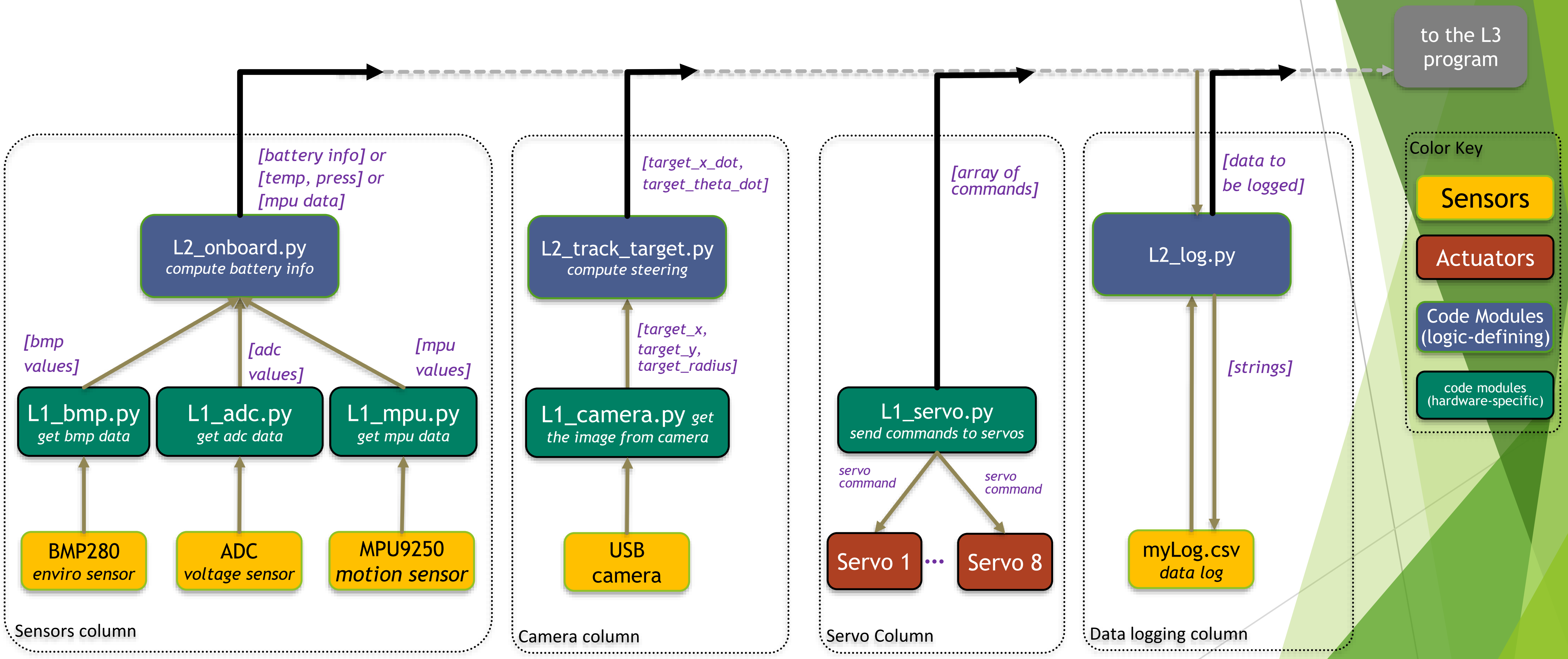
The SCUTTLE robot software has been programmed in Python3 on an embedded Linux platform. Both Beaglebone Blue and Raspberry Pi have been tested successfully. The software has been architected to make a robust starting point for students to create their own autonomous missions. These slides detail the software architecture.

Next Steps: ROS2 is quickly becoming a reliable, versatile software platform for mobile robots. During 2020 and 2021 the SCUTTLE team will aim to create a new ROS2 version of the software.

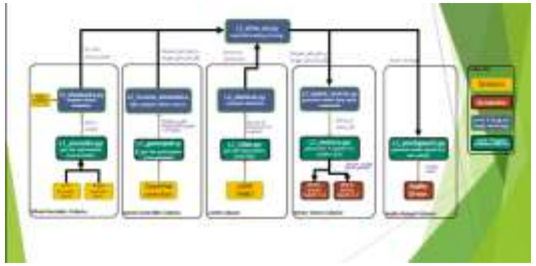
Software Architecture - Overview



Software Architecture – Overview (continued)



Software Architecture – Explanation

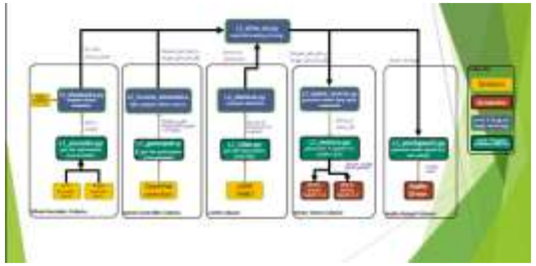


The blocks in yellow are sensors, and the items in orange are actuators or other outputs. The level-2 blocks in teal are specific to the hardware platform (beagle, pi, etc) and perform communication with the low level devices. The blocks above level 2 are non-hardware specific.

Each block aside from sensors and actuators represent an individual python program. The purple text indicates what important information is passed between programs and the black arrows indicate (for the most part) what direction the data is flowing. If a level 3 program needs information from another, it must receive the information from the top-level program, in order to maintain the structure of independence in program functions.

This software structure is preferred in order to perform subsystem testing. The data flowing through the top level is minimal and can be replaced with artificial data in the even that a sensor is unavailable.

Software Architecture – Libraries



Python importing guidelines:

1. Each file should import the files below it in hierarchy, and not the files above it.
2. Each file should not import files more than 1 level below itself.
3. Each file may import non-scuttle libraries as needed (import NumPy, import time, etc.)

Libraries Utilized:

BeagleBone Blue Integration:

- RCPY for communicating with MPU9250 & commanding motor drivers
- Adafruit GPIO for I2C Communication
- BMP280 for communicating with the onboard bmp280 sensor.

Raspberry Pi integration:

- pysicktim for accessing LIDAR data
- gpiozero for controlling GPIO pins.

Common Libraries

- os for making shell commands via python code.
- time for keeping track of time
- threading for performing multithreading
- NumPy for performing math operations
- pygame for accessing gamepad controller data
- cayenne.client for sending MQTT messages
- smbus for accessing i2c bus through python commands

Software Architecture - Level1 Program Example

Explanation of what the code does

Import External programs, then internal (if applicable)

Define constants and
take initialization actions

Define functions.
In some cases, make functions that
combine other functions in sequence.

Offer a simplified, minimal
loop for testing the code

```
# This example drives the right and left motors.
# Intended for Beagle hardware

import rcpy
import rcpy.motor as motor
import time # only necessary if running this program as a loop

motor_r = 2     # Right Motor
motor_l = 1     # Left Motor
rcpy.set_state(rcpy.RUNNING)

#channel refers to left(0) or right(1)
def MotorL(speed):
    motor.set(motor_l, speed)

def MotorR(speed):
    motor.set(motor_r, speed)

# Uncomment this section to run this program as a standalone loop
# while rcpy.get_state() != rcpy.EXITING:
#
#     if rcpy.get_state() == rcpy.RUNNING:
#
#         MotorL(0.5)  # gentle speed for testing program. 0.3 PWM may not spin wheels.
#         MotorR(0.5)
#         time.sleep(4) # run fwd for 4 seconds
#         MotorL(-0.5)
#         MotorR(-0.5)
#         time.sleep(2) # run reverse for 2 seconds
```

Software Architecture: SCUTTLE Threading Purpose

- Threading offers better control over **timing of code execution**.
- Each thread should contain **actions that are related** and that should be executed within a specific time window.
- The user should avoid passing data between threads because it reduces robustness. Instead, **call the level 2 program as needed in each thread**, even if you need to communicate with the same device (ie, retrieve gamepad commands for driving and retrieve in parallel for speaking commands)

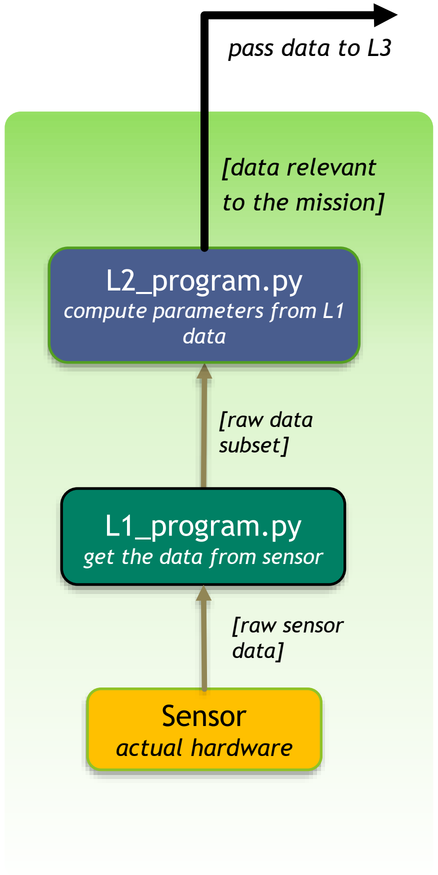
	Thread 1 (driving)			Thread 2 (obstacle detect)	Thread 3 (speaking)
Do What?	Drive the robot	AND	Log the speed	Indicate obstacles using speaker	speak via speaker
When?	After sampling the controller		After sending drive command	when obstacle is detected by lidar	user presses button
	20ms cylcle			10ms cylcle	5ms cylcle

Software Architecture: Tiers Guidelines

Importing
Files & Libraries

Only import L1 programs directly below. Do not import L3 program. As needed, import external libraries.

Do not import L2 or L3 programs. As needed, import external libraries.



Testing
Code and Hardware

This file has a loop to test the whole group (L1 + L2 + sensor).

This file has a loop to test sensor function.

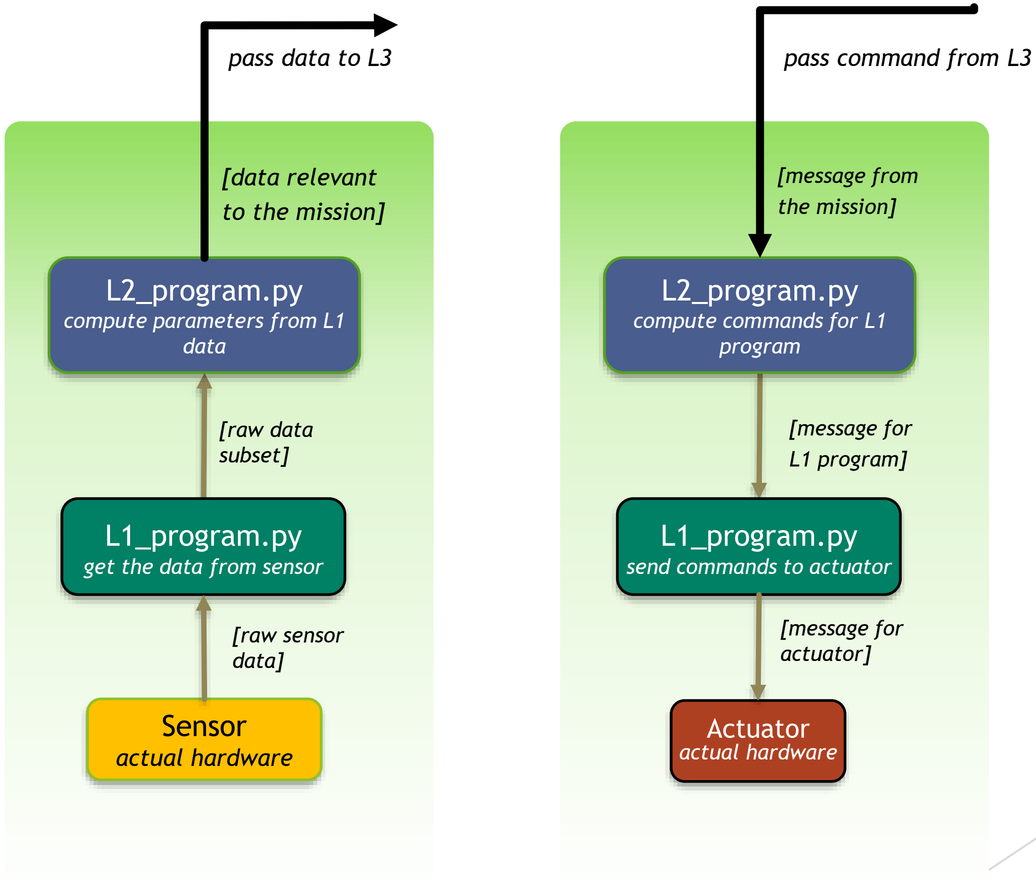
Software Architecture: Sensors vs Actuators

Sensor and Actuators have the same architecture except for **data direction**.

For **sensors**, the data is generated at the hardware and sent UP.

For **actuators**, the data is generated at the top and sent DOWN to hardware.

Some sensors and actuators have feedback and preset commands, so data may flow both ways.



Software Architecture: Modularity & Robustness

Send and receives data in the format that is “universal” or standardized.

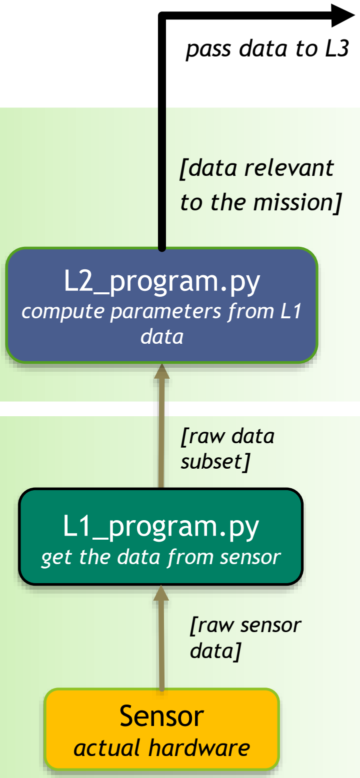
If there is a hardware change, this program does not need to change

If the driving strategy changes, this program may change

The job of L1 program is to convert the data into the desired format.

If there is a hardware change, only these items need to be updated.

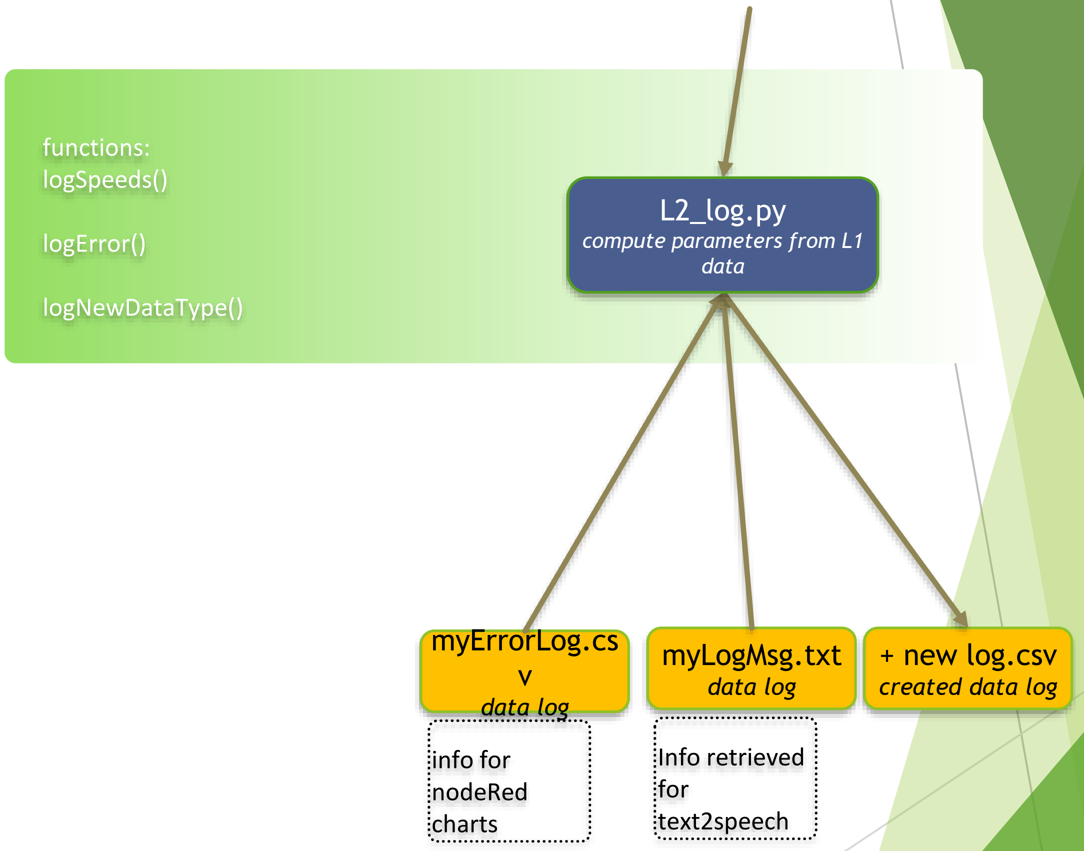
If the driving strategy changes, these items do not change.



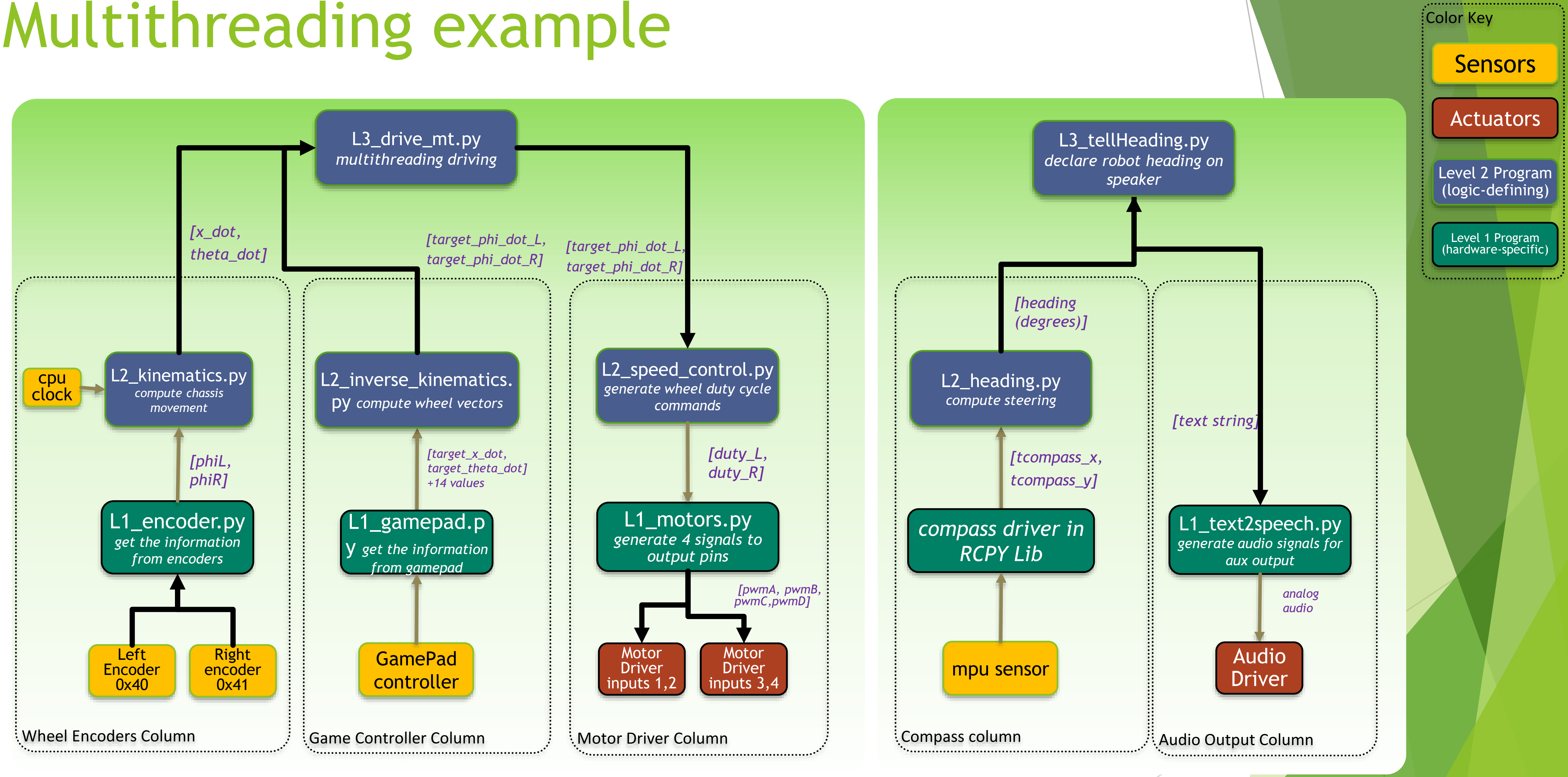
Software Architecture: Editing Log Files

Rather than interacting with hardware, the L2_log program will interact with files on the linux machine. It **acts as a sensor** in that it retrieves recorded data and it **acts as an actuator** in that it can receive data and perform an action with it (store in a file).

As of 2019.08, the L2_log.py program is not written but the intended functionality is shown. There will be ability to access different files and to write new files, by calling the functions from L2_log.py.



Multithreading example



Python3 Libraries in Use

Common Libraries

- os for making shell commands via python code.
- time for keeping track of time
- threading for peforming multithreading
- numpy for performing math operations
- pygame for accessing gamepad controller data
- cayenne.client for sending MQTT messages

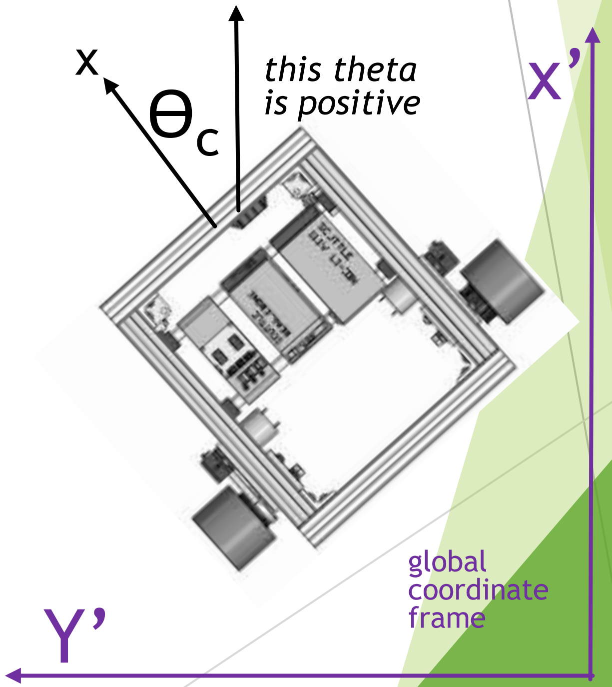
BeagleBone Blue

- RCPY
  - for communicating with MPU9250
  - for commanding motor drivers
- Adafruit GPIO
  - I2C Communication

Absolute Orientation

- SCUTTLE has a compass for orientation
  - The compass is nothing but a 3-axis magnetometer
  - Encoders can provide *relative* orientation
  - Compass is required for *global* orientation
- The compass is embedded in the IMU (MPU-9250)
  - It has 3 sensors oriented in the indicated directions
  - L1_mpu.py accesses the magnetometer
  - Each magnetometer requires calibration

Remember: Theta is defined as scuttle's x-vector minus the global x-vector

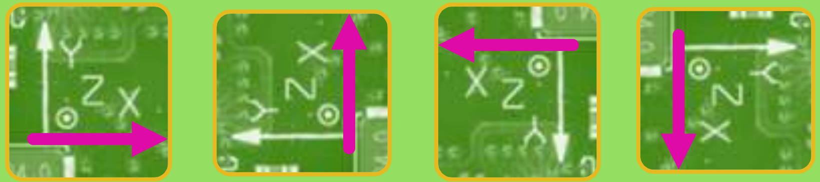


Magnetometer Behavior

- An axis is at its MAXIMUM when it is **aligned** NORTH
- The axis is at its MINIMUM when it is **opposing** NORTH
- After calibration, we can achieve the behavior below

Values Desired by Direction

Using X axis for example:



$x_c = 0$ | $x_c = 1$ | $x_c = 0$ | $x_c = -1$

(magnetic north)

1) Discover the maximum and minimum values by rotating sensor in a full circle.

Permanent magnets influence the sensor, so calibration must be done on the robot, in position near the motors.

Before Calibration	Min (microtesla)	max (microtesla)
X	-15	38
Y	-22	20

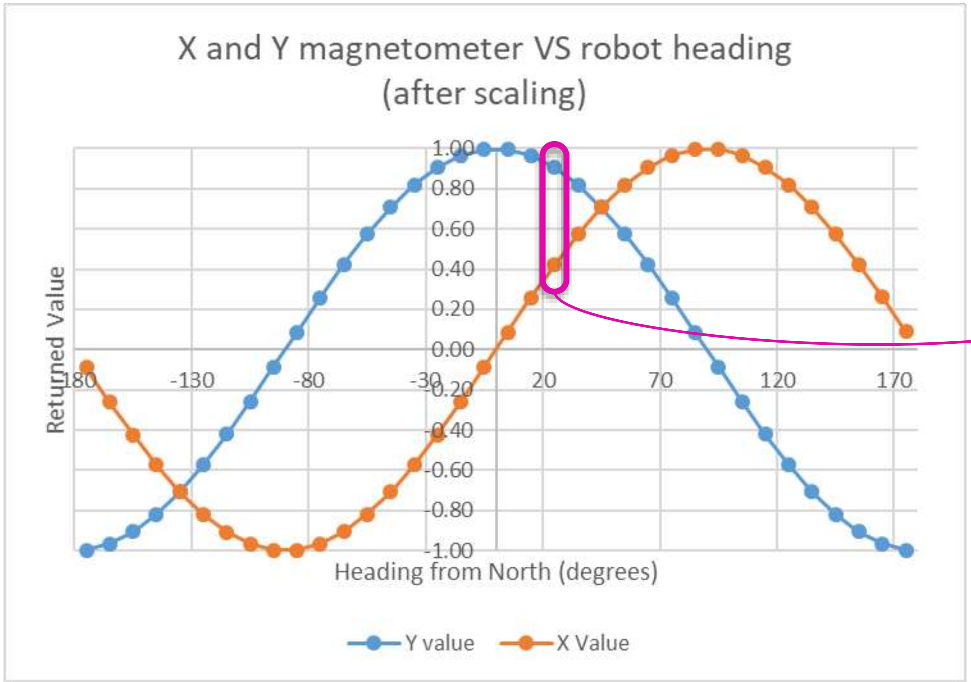
2) Using the following equation, re-scale each axis

$$x_{\text{scaled}} = \frac{2(x - x_{min})}{(x_{max} - x_{min})} - (1)$$

AfterCalibration	Min (ratio to max)	max (ratio to max)
X	-1	1
Y	-1	1

Determining Absolute Orientation

- X and Y axes are sufficient information to give heading.
  - Z axis returns zero if scuttle sits flat
- Theta is defined as rotation of SCUTTLE from the global coordinate frame, or y-prime
  - positive theta means SCUTTLE is turned left from north
  - We can define NORTH as the y-axis of the global coordinate frame



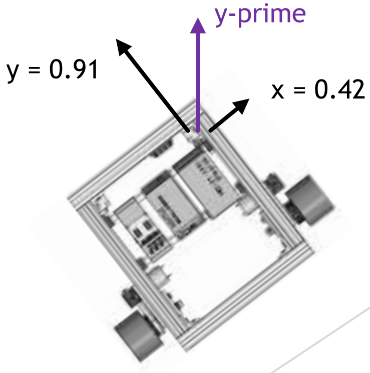
Theta is positive when scuttle points west
Theta is negative when scuttle points east

Use arctan2(y, x) to return a heading

arctan2 is the "element-wise arc tangent of y/x choosing the quadrant correctly."

Example:

ATAN2(0.91, 0.42) returns 25 degrees



y is pointed strongly north
X is pointed weakly north
both axes return positive values

Speeds Tuning

- V_{max} = 0.4m/s (measured by wheelspeed)

$$v = \omega * r$$

- $\omega_{max,\ motor\ pulley} = 19.5$ rad/s
- $\omega_{max,\ wheel} = 9.75$ rad/s
- With 1 wheel stopped and 1 wheel moving:

$$\dot{\Theta} = \frac{v}{L}$$

(where L = half of wheelbase)

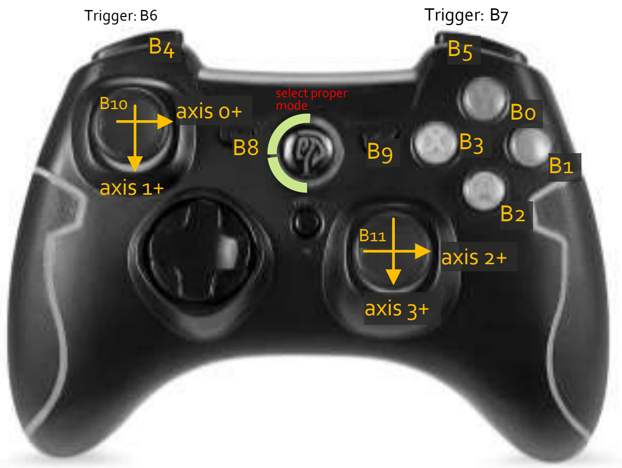
SCUTTLE Driving

- The left joystick operates the robot wheels
- The forward/backward axis will request a speed
  - (A.K.A movement in x)
- The left/right axis will request an angular velocity
  - (A.K.A movement in theta)

Given a measured max forward velocity of 0.4m/s, the other maximums are calculated.

Move	Theta_dot (rad/s)	X_dot (m/s)	Phi_dot
max	1.99	0.4	9.75

Gamepad Controls Mapping




```
axes_status = np.array([axis_0, axis_1, axis_2, axis_3])
button_status = np.array([B0, B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11])
```

SCUTTLE Movement Example

USER SAYS: right turn on joystick

GAMEPAD SAYS: outputs positive value on axis_1

KINEMATICS SAYS: interpret as negative theta movement

KINEMATICS SAYS: performs POSTIVE phi_dot_l and NEGATIVE phi_dot_R

MOTOR PROGRAM SAYS: performs POSTIVE differential on NEGATIVE phi_dot_R

Move	Theta_dot	X_dot	Duty_r	Duty_l
Fwd	0	1	1	1
Rev	0	-1	-1	-1
right	-1	0	-1	1
left	+1	0	1	-1

LIDAR Concept of Operation

ANATOMY
Lidar systems have a rotating sensor collecting multiple measurements to measure in a 2D plane. (Some have 3D, by other methods).

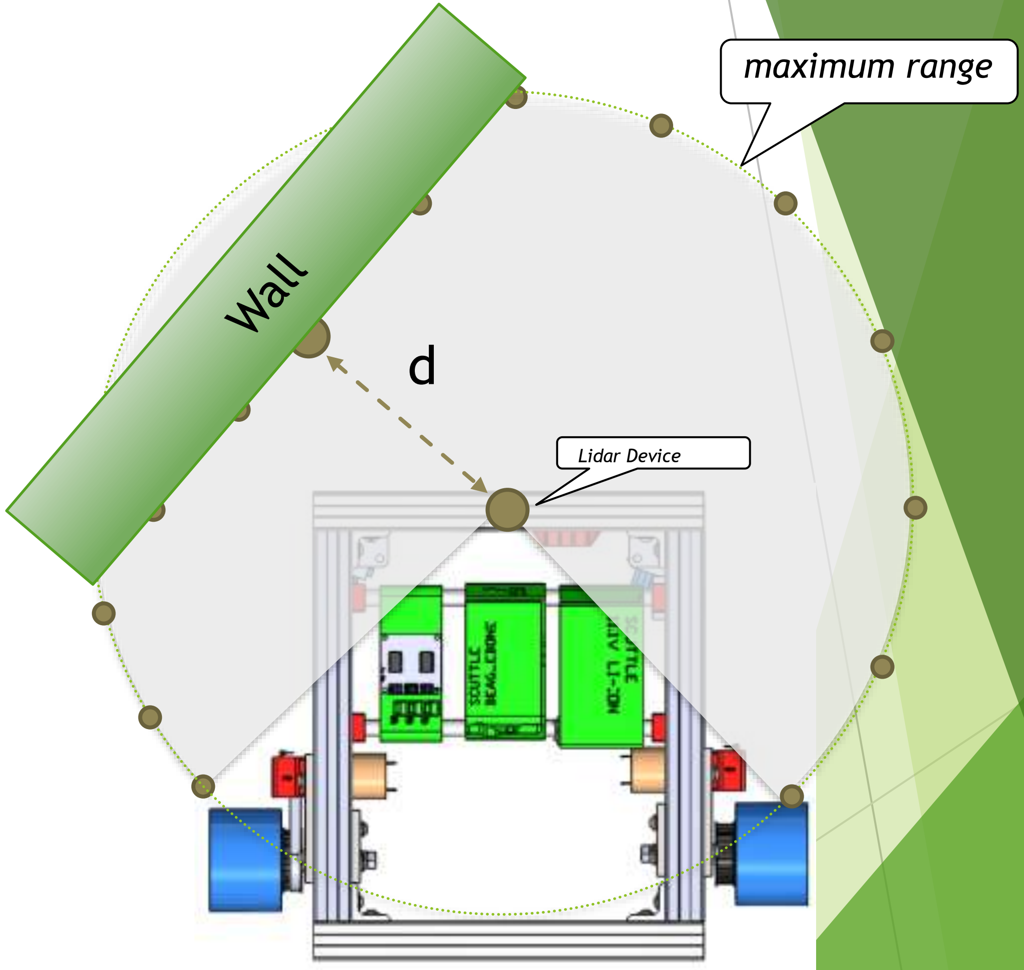
METHOD
Lidar emits a beam of light and receives the reflection. distance is based on Time of Flight concept.

POWER
TiM561 uses about 2.1 watts during operation, mainly due to driving the motor and driving a strong IR emitter diode.

FAILURE MODES
Just like light, a Lidar beam can be absorbed by very dark objects and can be mis-directed by highly reflective objects which are non-perpendicular to the beam.

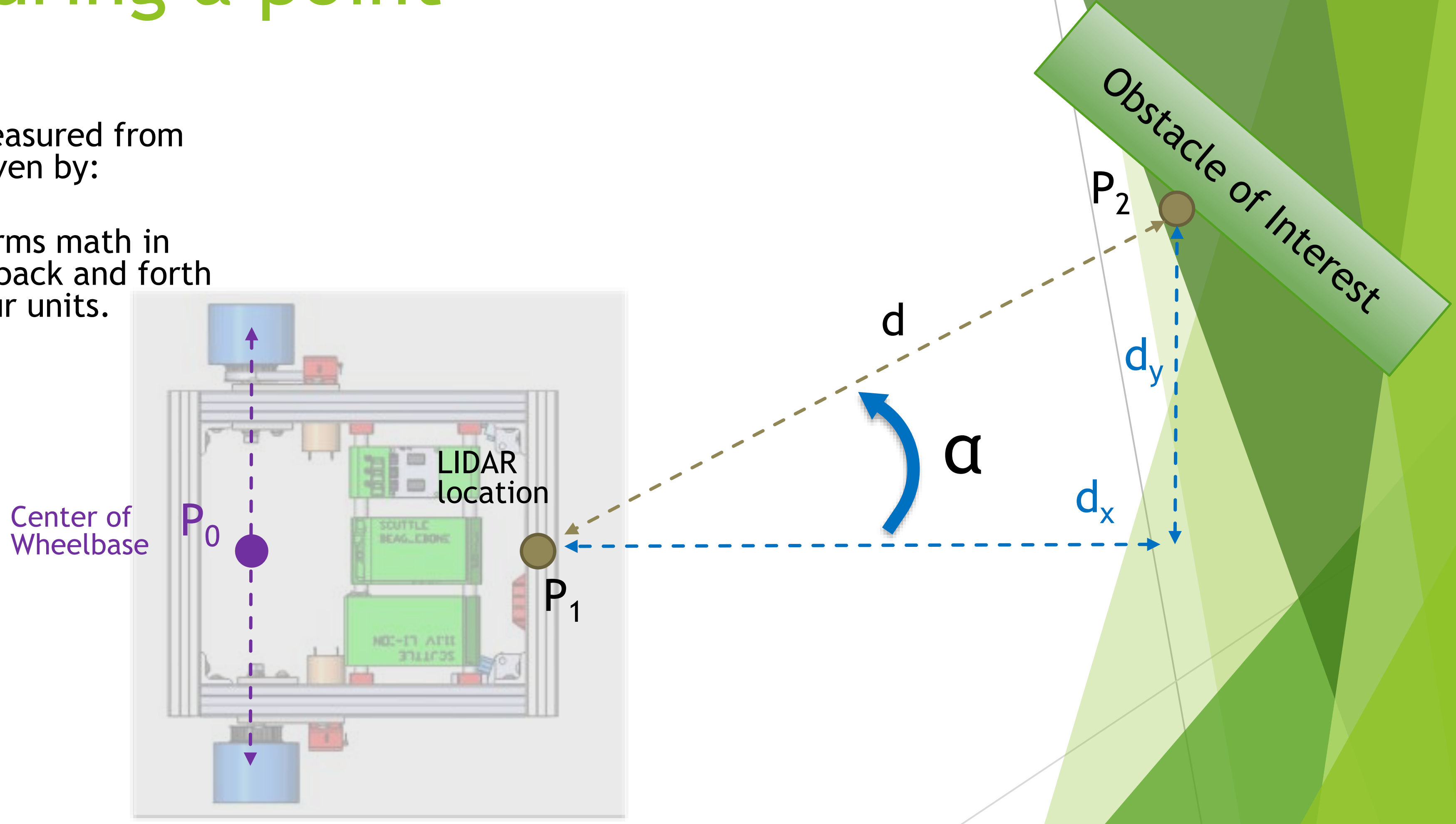
DATA QUALITY
The lidar has *variable resolution* in a sense! 0.33 degrees offers 5mm point spacing at a 1m distance, and at 10 meters, 50mm point spacing.

APPROPRIATE USE
To be successful in using the device, you need to see the datasheet to understand the parameters of your device.



LIDAR – measuring a point

- P_1 is the location of the lidar.
- The points will be initially measured from lidar and returned as pairs given by:
- [d (mm), α (degrees)]
- Python's numpy library performs math in radians. It is easy to convert back and forth but you must be aware of your units.



Software For LIDAR

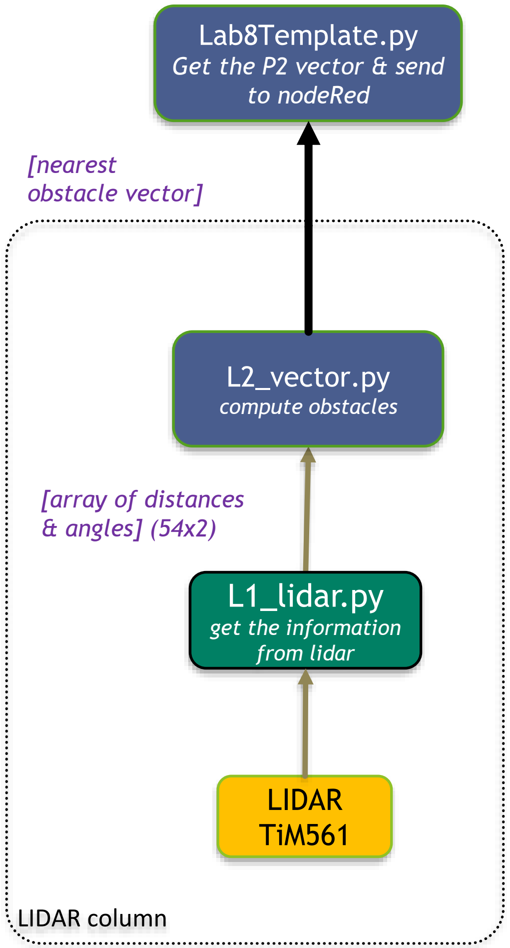
<u>Key Points:</u>
Software is using the numPy library to handle vectors and matrices. numPy computation is faster than raw python and requires proper syntax.

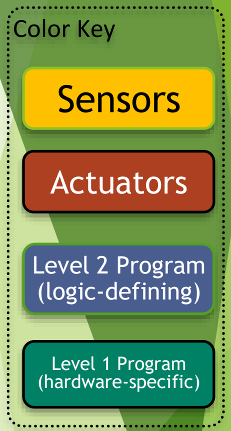
Lidar scan frequency: 15hz, so you cannot get new measurements faster than 66ms.

L1_lidar.py returns 54 measurements by default and can return over 800 single points if desired, for more resolution.

TiM561 LIDAR returns distances in meters. Distances under 16mm are returned as error codes in case of poor reflection or other problem for a given measurement.

L2_vector.py can manipulate measurements, with functions such as returning the nearest point, combining cartesian vectors, and converting vectors from polar to Cartesian coordinates.



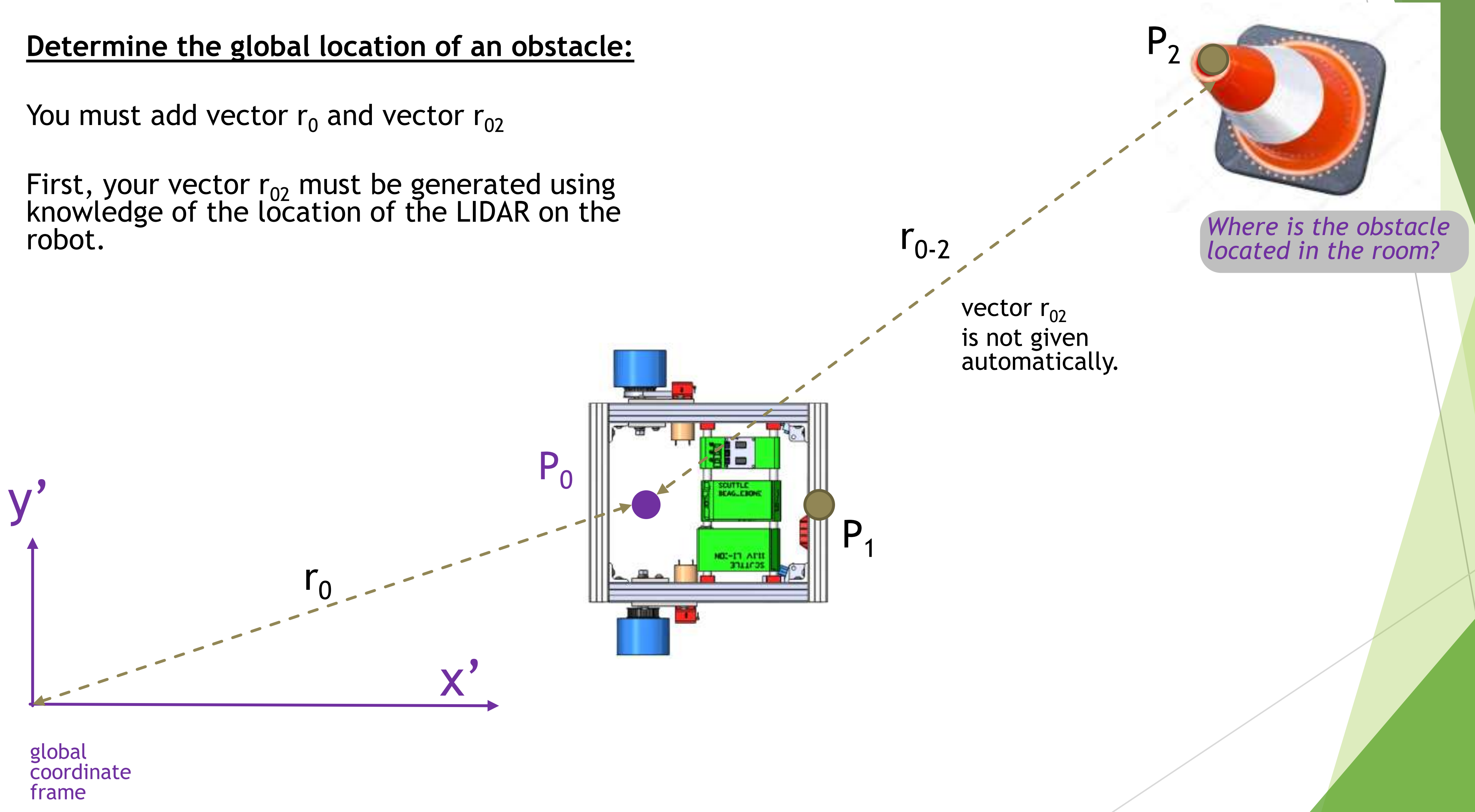


Global Location of Obstacle

Determine the global location of an obstacle:

You must add vector r_0 and vector r_{02}

First, your vector r_{02} must be generated using knowledge of the location of the LIDAR on the robot.

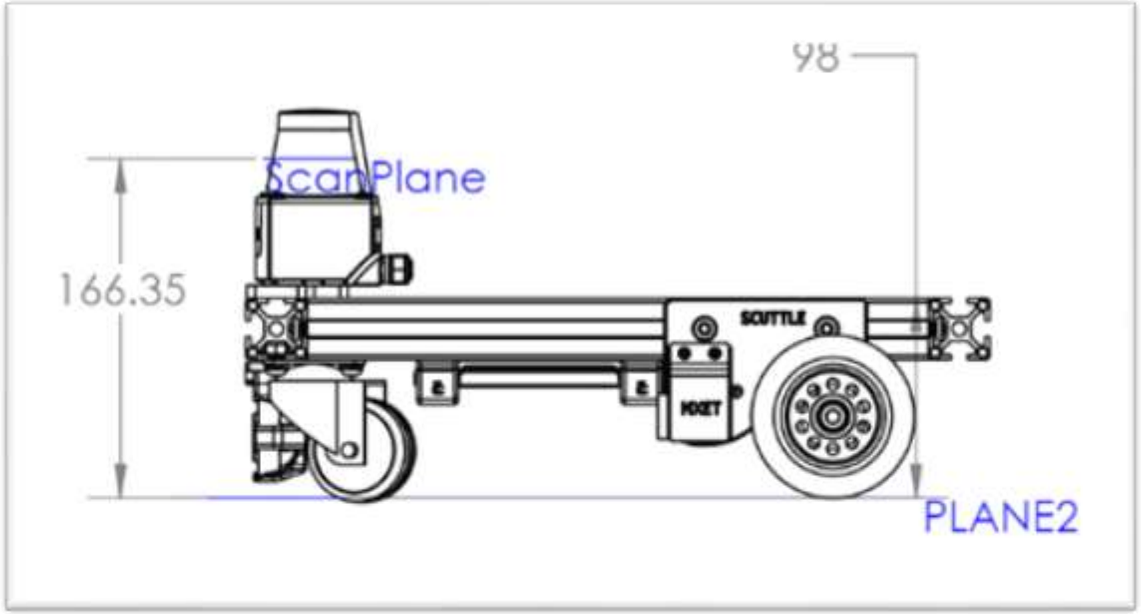


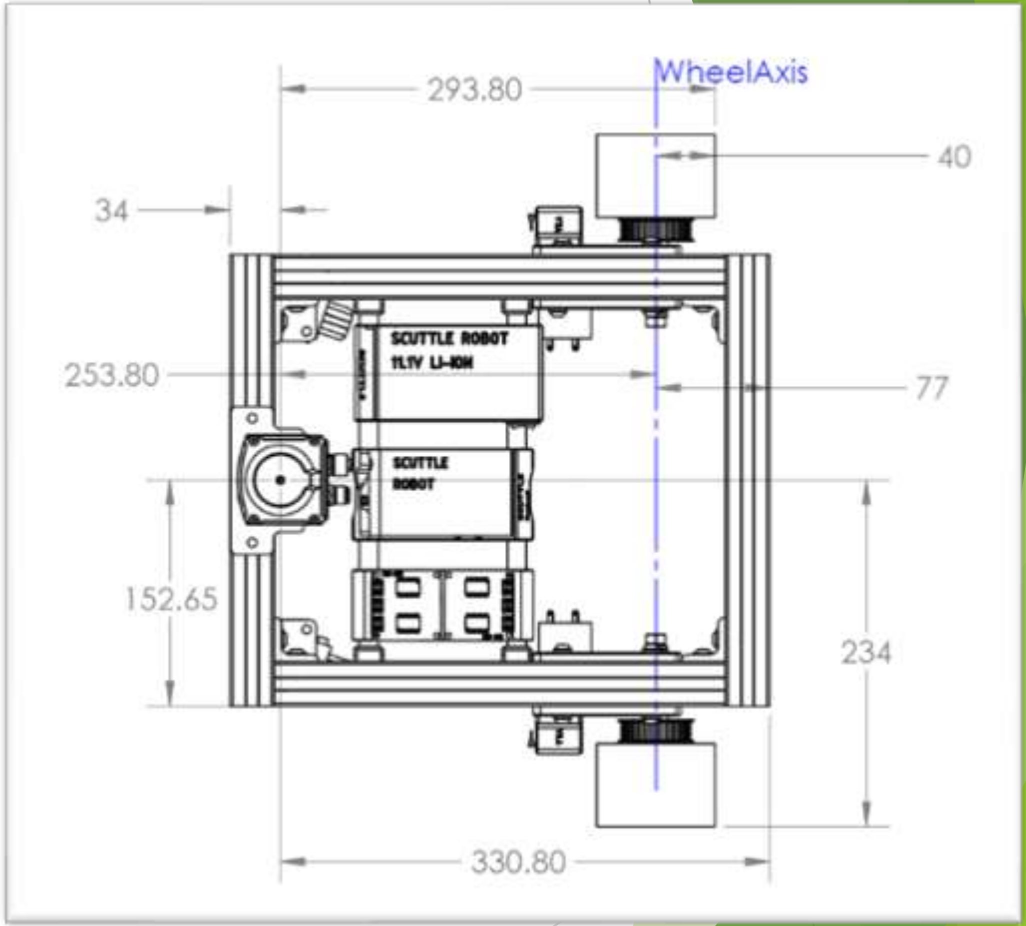
Global Location of Obstacle

<u>Determine the global location of an obstacle:</u>

Lidar is located at positive 254mm in the x-direction on the robot.

The lidar beam is 166 mm above the floor.



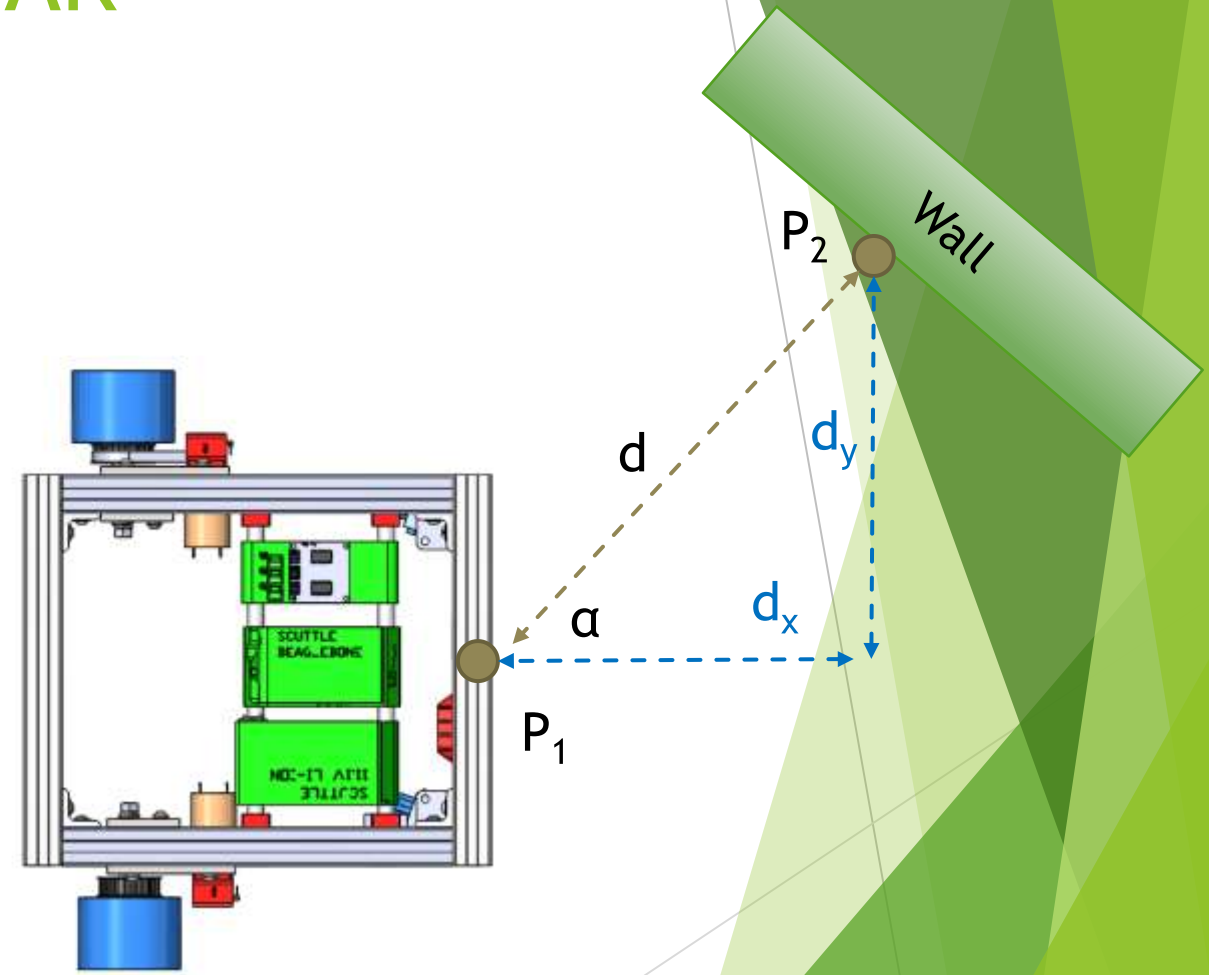


Obstacle Avoidance by LIDAR

One method to avoid obstacles is to generate an imaginary spring which pushes on your robot and depends on the nearest obstacle.

D_y is the y-component of distance d

D_x is the x-component of distance d

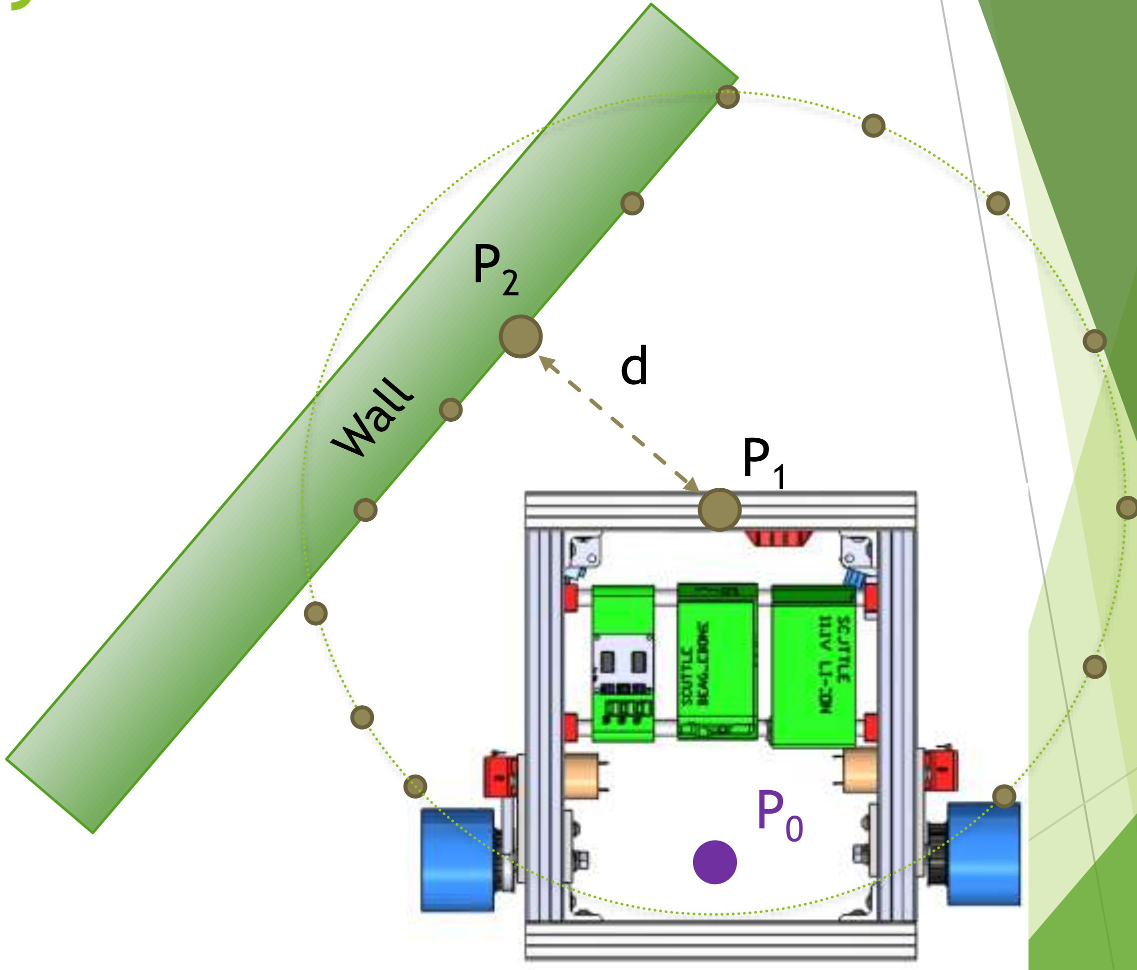


Obstacle Avoidance by LIDAR

Strategy:
The obstacle avoidance feature will try to detect the nearest objects to the robot and apply an "invisible force" to prevent the robot from crashing. The force is intended to act like a spring which is anchored to the nearest obstacle and pushes the robot at a point on the body, referred to as P1.

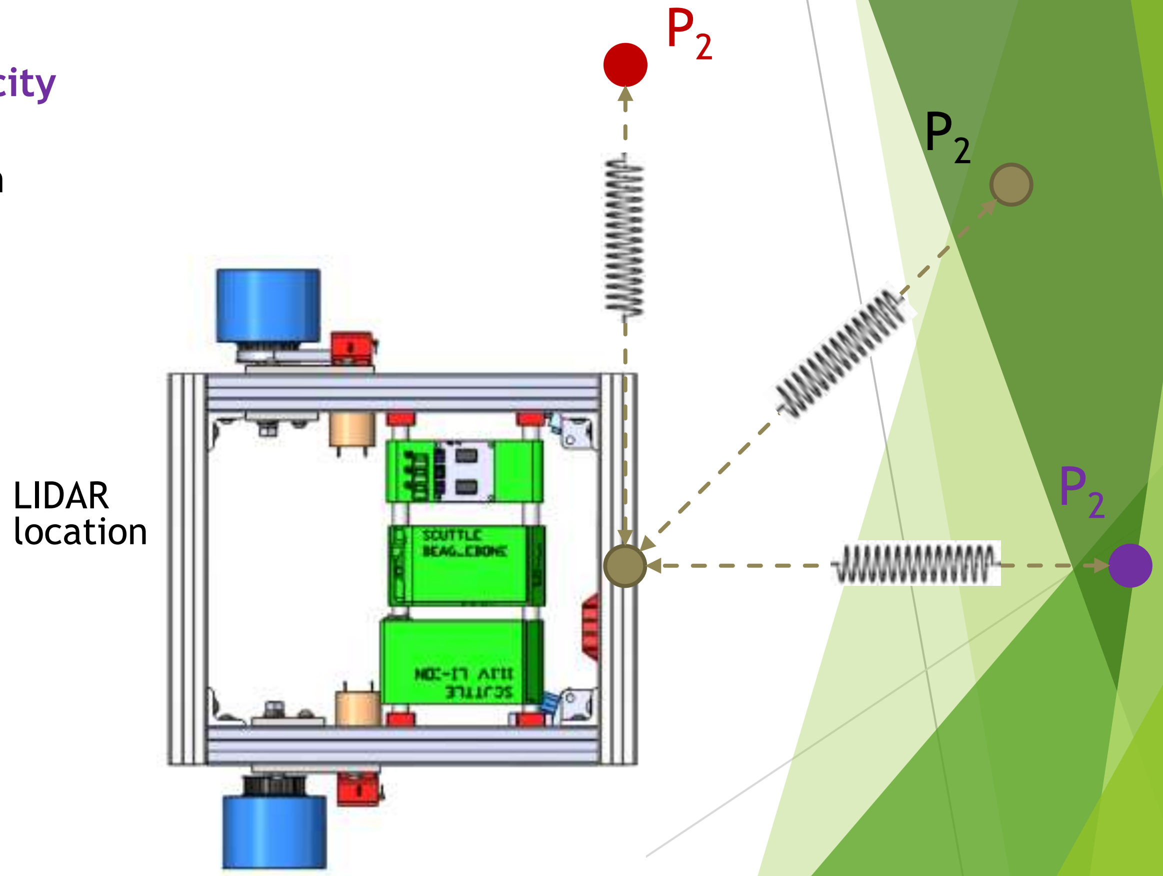
The obstacle avoidance only deals with the body-fixed frame

- Define P_1 as a point of interest on our robot.
- P_2 is assigned to the nearest point detected by the LIDAR scan.
- d is the distance between point 1 and point 2
- We would like to handle all of these variables in:
  - body-fixed frame
  - Cartesian coordinates



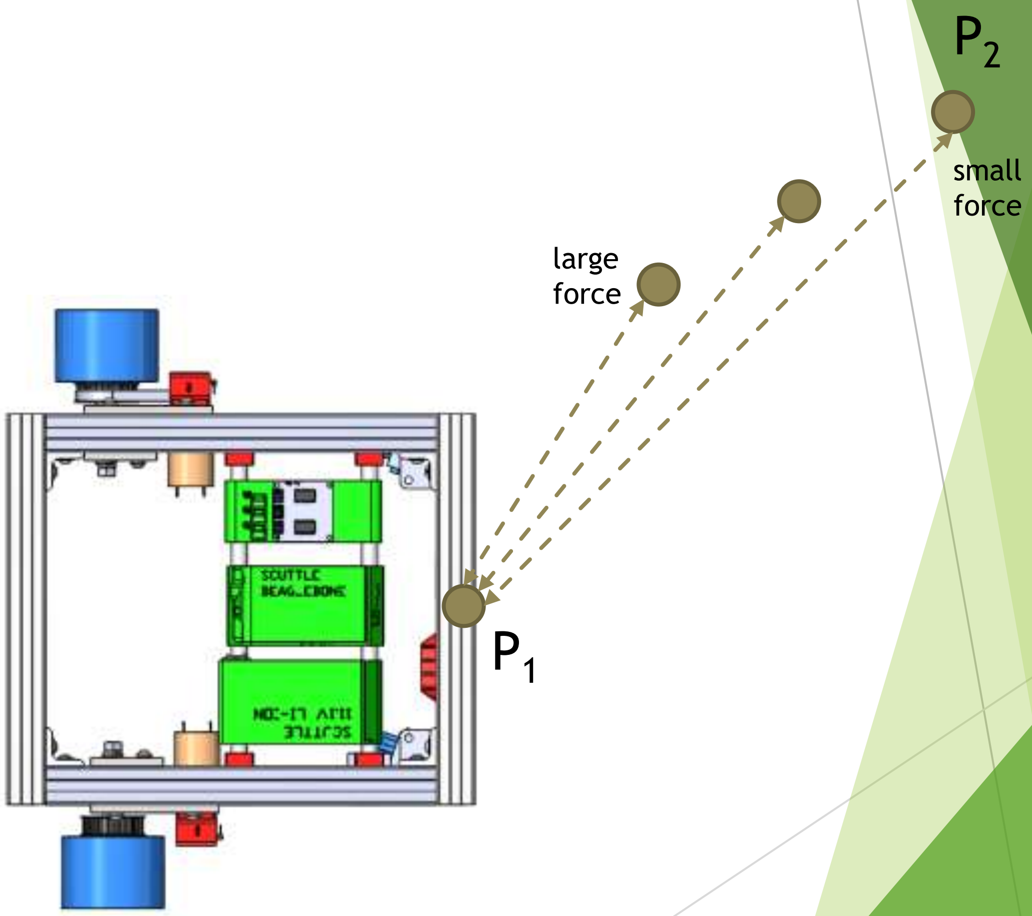
Obstacle Avoidance – influence on velocity (translational and angular)

- If P2 is detected straight ahead, the force influences velocity
- If P2 is detected at the side, the force influences steering
- If P2 is detected at an angle, the force will **influence both**



Obstacle Avoidance – Variable Force

- If d is large, the force is low
- If d is small, the force is high
- If d is larger than d_{max}, the force is absent



Quick Dive – Barometric Pressure

COLLEGE STATION 4-DAY CHANGE

delta in pressure is 30.09-29.58 “Hg ➔ 0.51” Hg
delta pressure = 1.73kPa

STANDARD PRESSURE CALCULATIONS

sea level std pressure: 101.3kPa
pressure at 1000ft: 97.7kPa
delta pressure = 3.6kPa
elevation change represented by 1kPa = 278ft

What the Barometric pressure will tell you:

1.73kPa change in pressure will represent 480ft altitude change.

High & Low Weather Summary for the Past Weeks

	Temperature	Humidity	Pressure
High	93 °F (May 28, 2:53 pm)	97% (May 17, 5:53 am)	30.09 "Hg (May 17, 5:53 am)
Low	65 °F (May 16, 4:53 am)	39% (May 15, 3:53 pm)	29.58 "Hg (May 21, 2:53 am)
Average	80 °F	76%	29.85 "Hg

* Reported May 15 10:53 am — May 30 10:53 am, Bryan – College Station. Weather by CustomWeather, © 2019

Note: Actual official high and low records may vary slightly from our data, if they occured in-between our weather recording intervals... More about our weather records

https://www.timeanddate.com/weather/usa/bryan-college-station/historic

Further Reading

- https://en.wikipedia.org/wiki/Holonomic_(robotics)

- Connector types
- http://dangerousprototypes.com/blog/2017/06/22/dirty-cables-whats-in-that-pile/

End