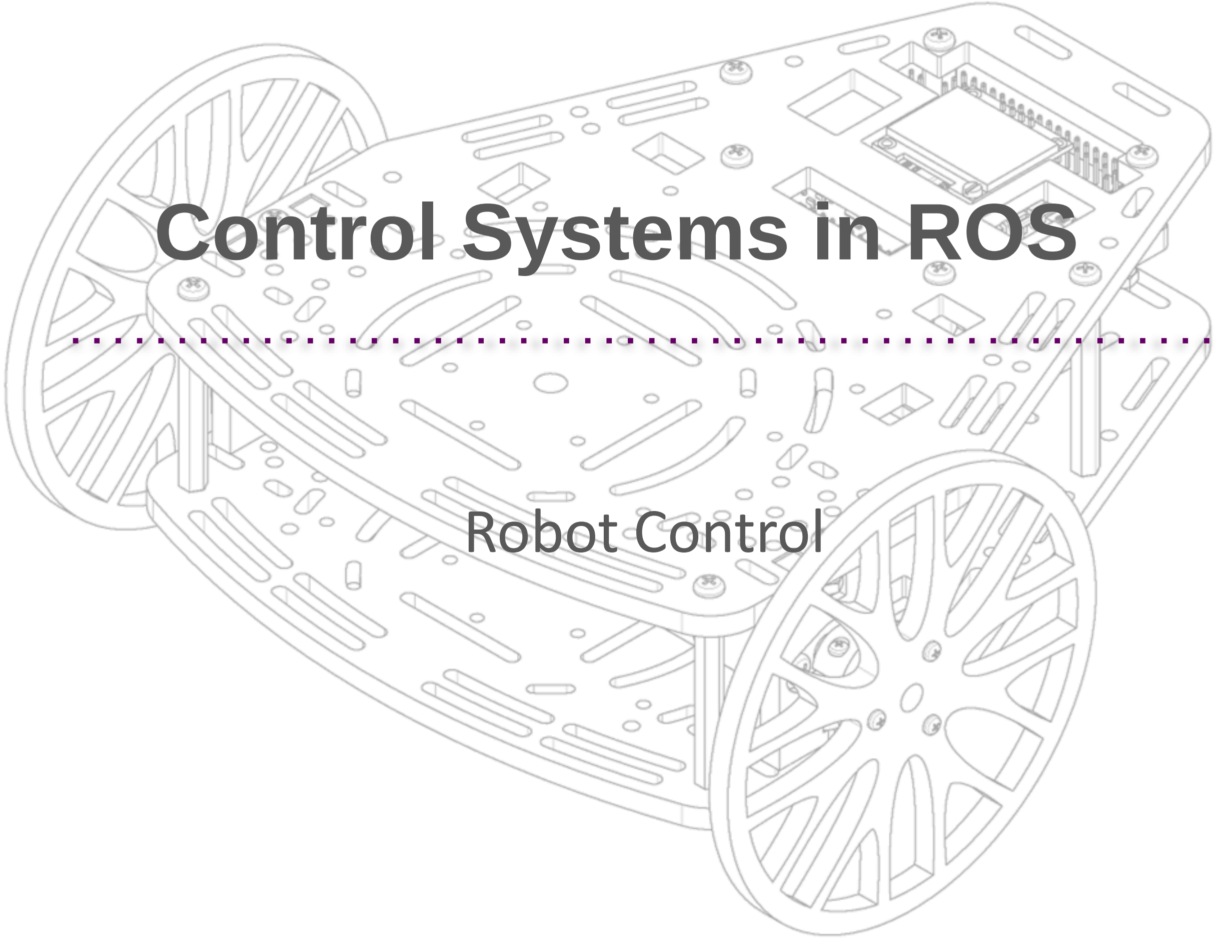
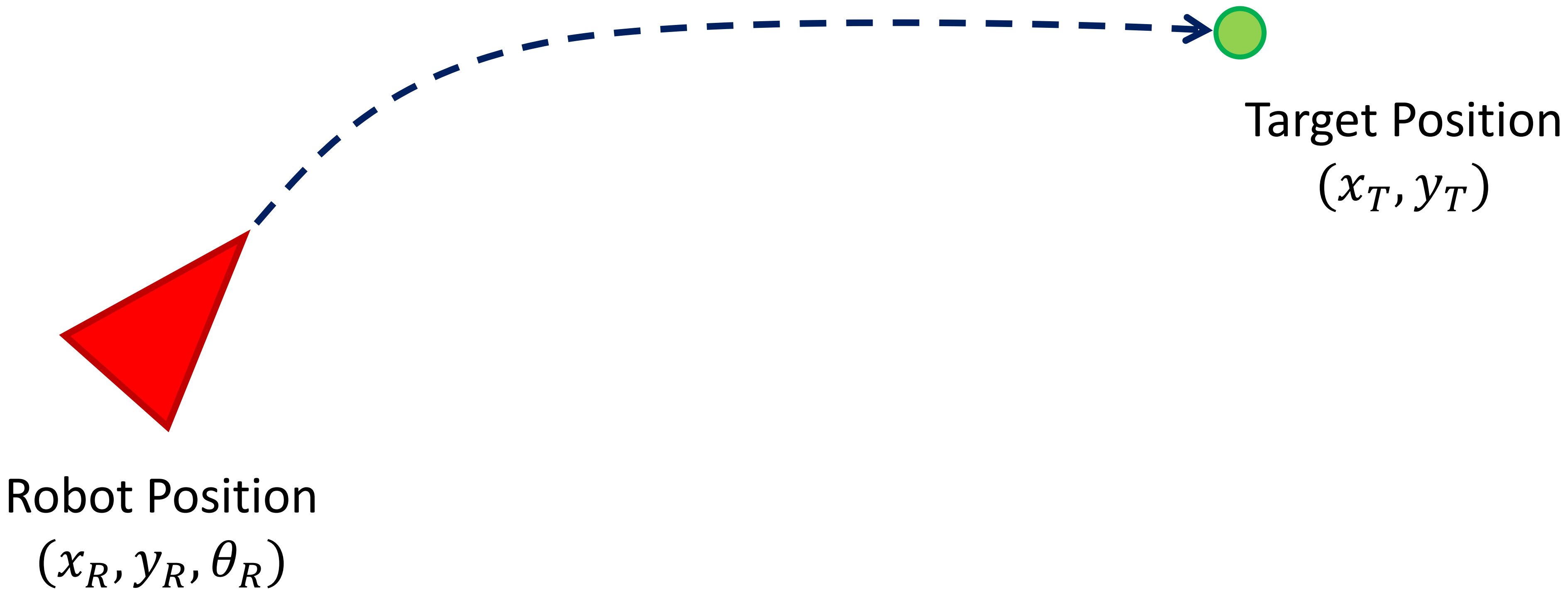
# Control Systems in ROS

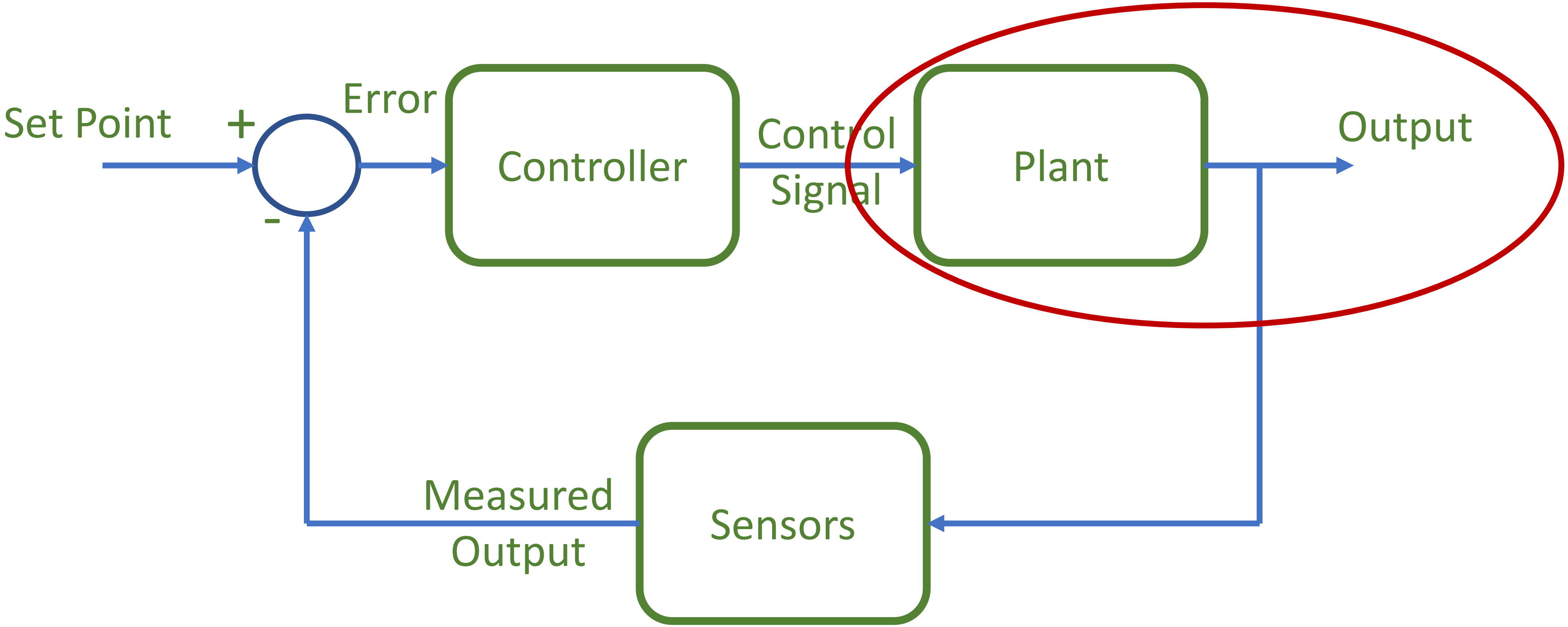


Robot Control

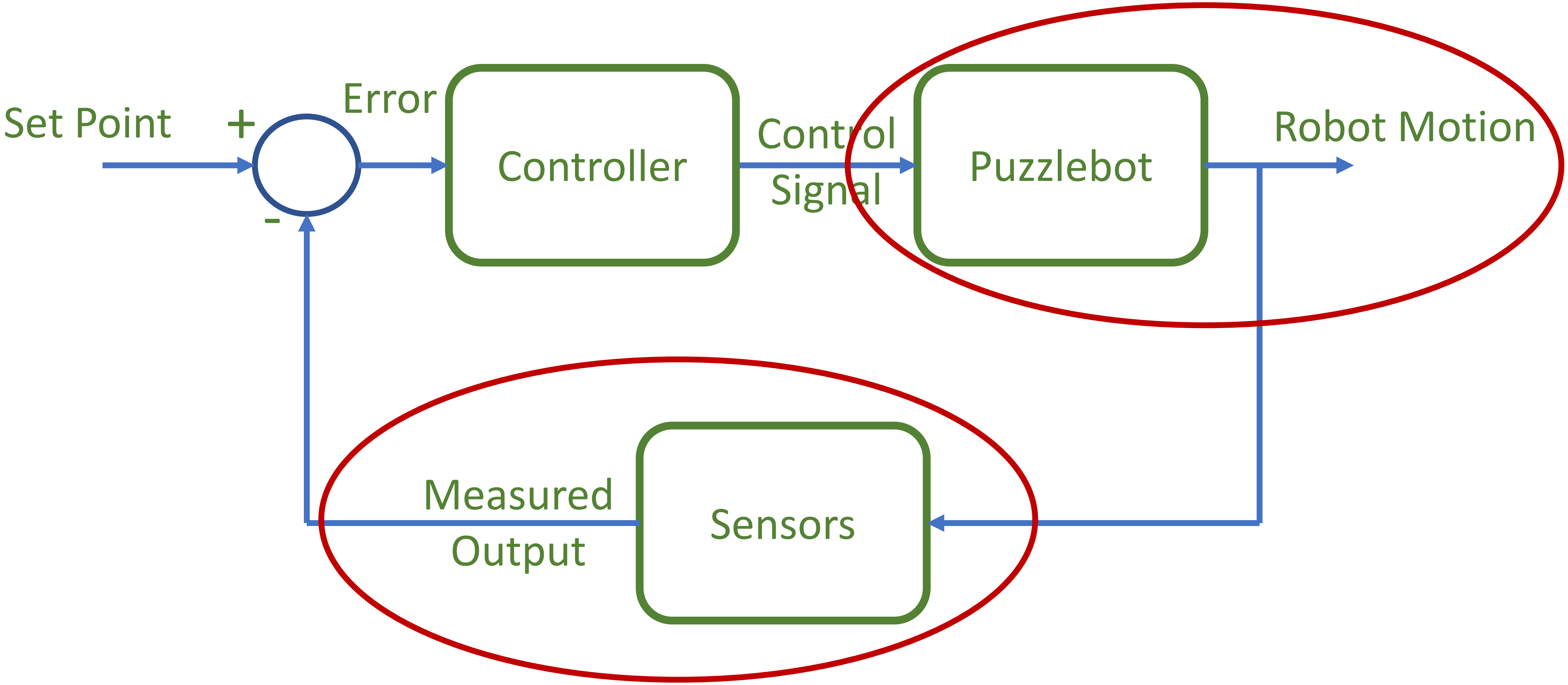
# The Task



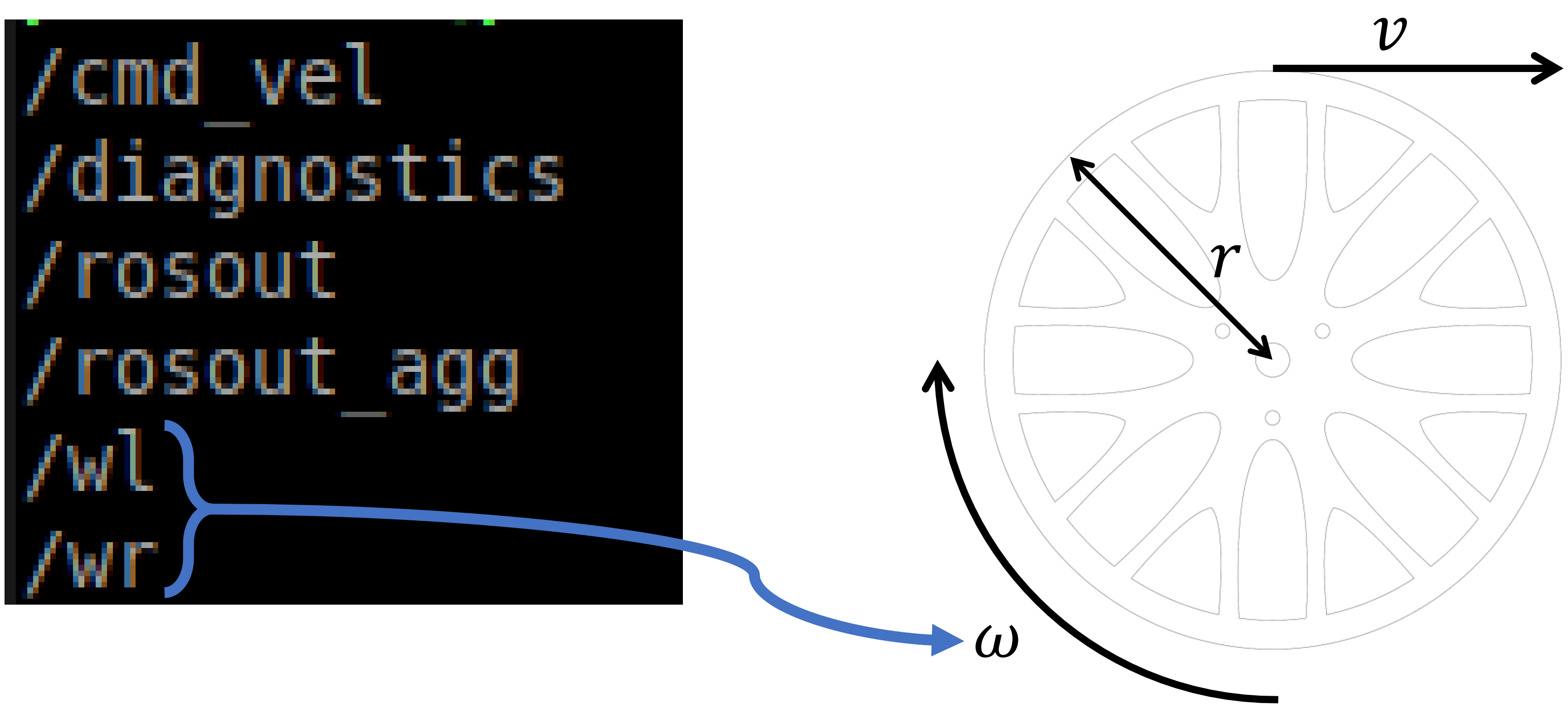
# The Control System



# The Control System

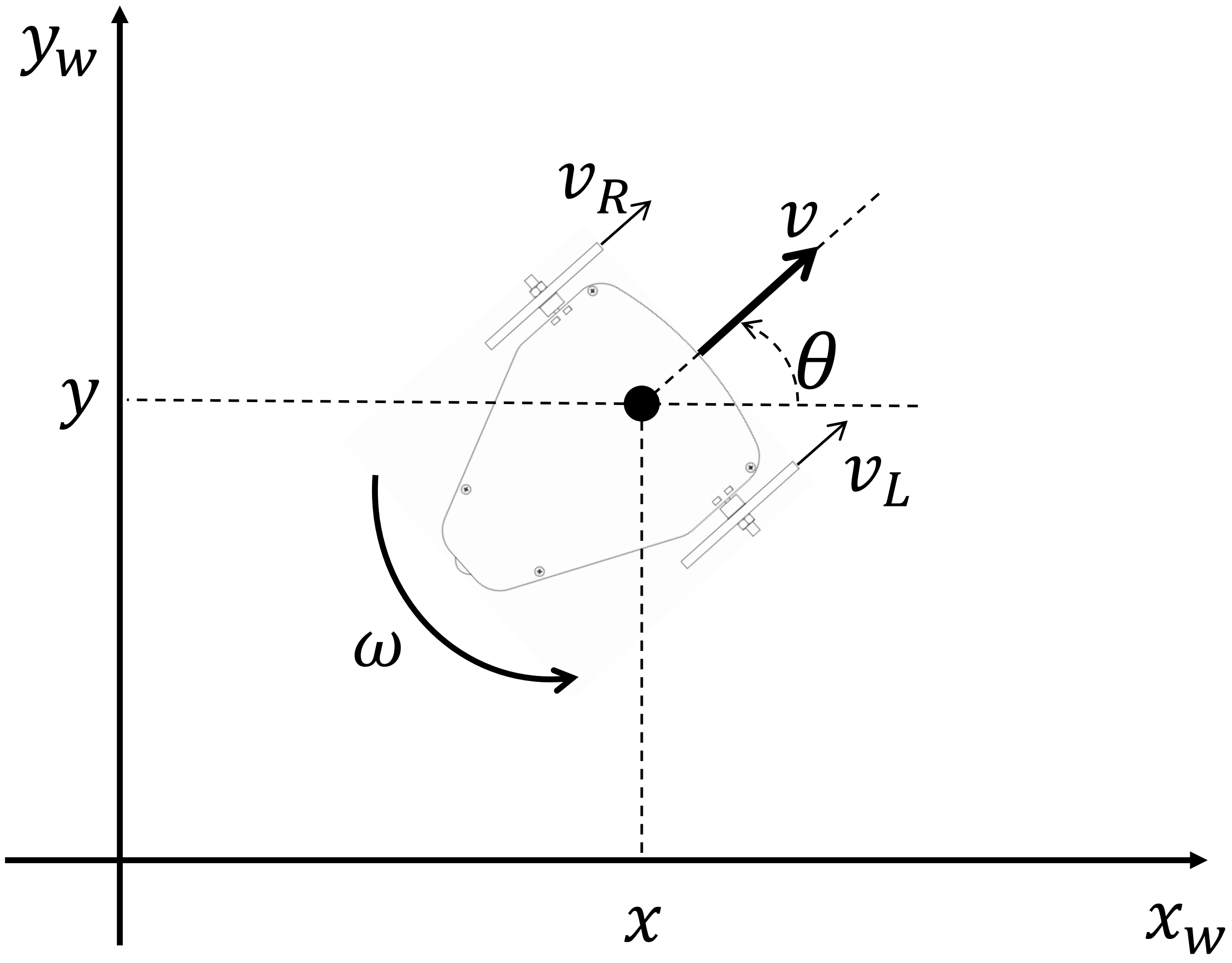


# Determining the Robot Position




$$v_{Wheel} = r\omega_{Wheel}$$

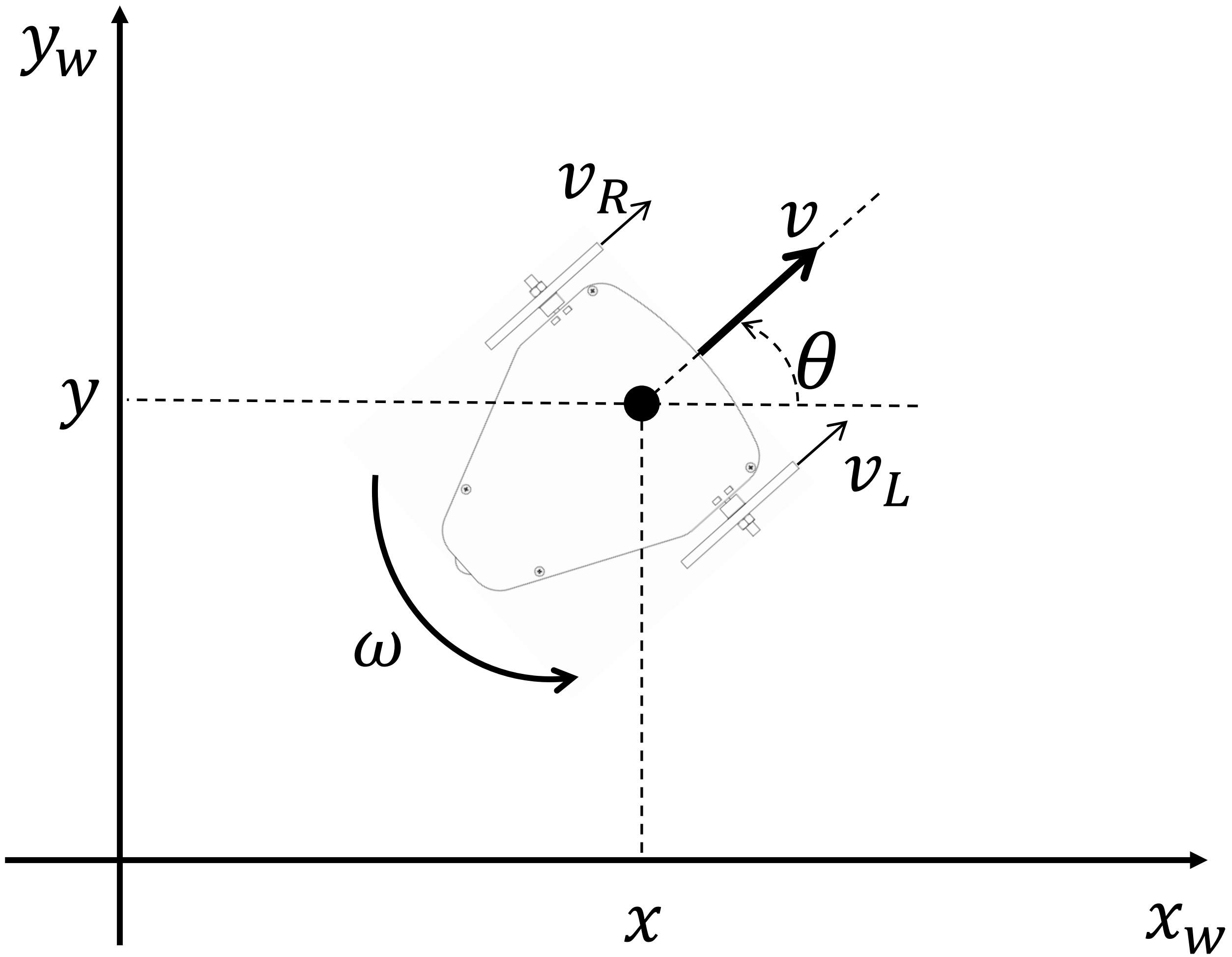
# Determining the Robot Position




$$v_{Robot} = \frac{v_R + v_L}{2} = r\frac{\omega_R + \omega_L}{2}$$

$$\omega_{Robot} = \frac{v_R - v_L}{l} = r\frac{\omega_R - \omega_L}{l}$$

# Determining the Robot Position



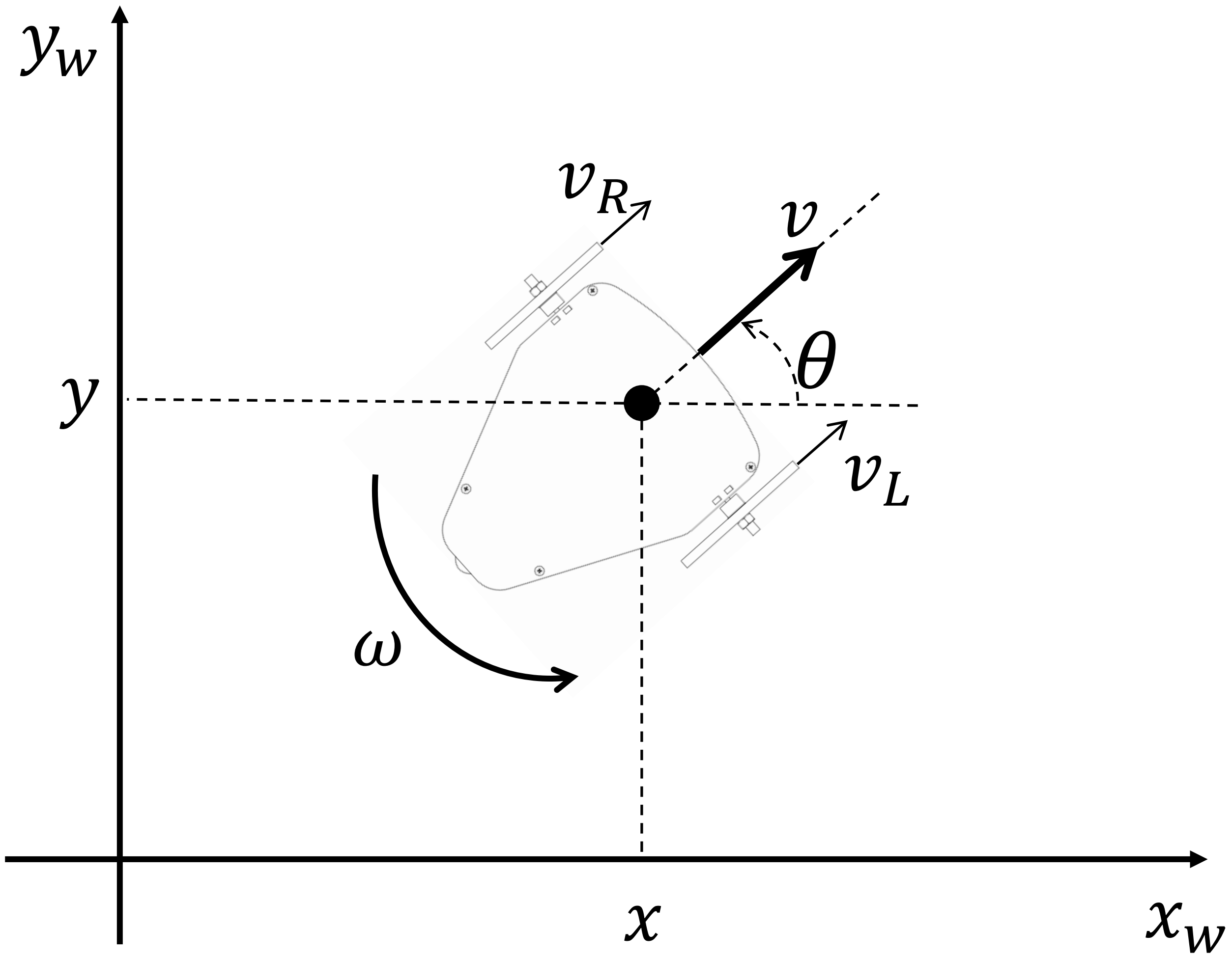

$$v_{Robot} = \frac{v_R + v_L}{2} = r\frac{\omega_R + \omega_L}{2}$$

$$\omega_{Robot} = \frac{v_R - v_L}{l} = r\frac{\omega_R - \omega_L}{l}$$

$$\dot{x} = v\cos\theta = r\frac{\omega_R + \omega_L}{2}\cos\theta$$

$$\dot{y} = v\sin\theta = r\frac{\omega_R + \omega_L}{2}\sin\theta$$

# Determining the Robot Position




$$\theta_{k+1} = \theta_k + r\frac{\omega_R - \omega_L}{l}dt$$

$$x_{k+1} = x_k + r\frac{\omega_R + \omega_L}{2}dt\cos\theta$$

$$y_{k+1} = y_k + r\frac{\omega_R + \omega_L}{2}dt\ \sin\theta$$

# Determining the Robot Position

$$\theta_{k+1} = \theta_k + r\frac{\omega_R - \omega_L}{l}dt$$

$$x_{k+1} = x_k + r\frac{\omega_R + \omega_L}{2}dt\cos\theta_k$$

$$y_{k+1} = y_k + r\frac{\omega_R + \omega_L}{2}dt\sin\theta_k$$

Robot Location:
$(x_k, y_k, \theta_k)$: Pose of the robot at timestep $k$ (m, m, rad).
Stored in memory, initial value 0

Robot Constants:
$r$: Wheel radius = 0.05 m
$l$: Distance between robot wheels = 0.19 m

Measured variables
$(\omega_R, \omega_L)$: Wheel velocity (rad/s)
$dt$: Time between samples (s)

Values of $\theta$ must be contained within a single circle:
Either:

$$-\pi \leq \theta < \pi$$

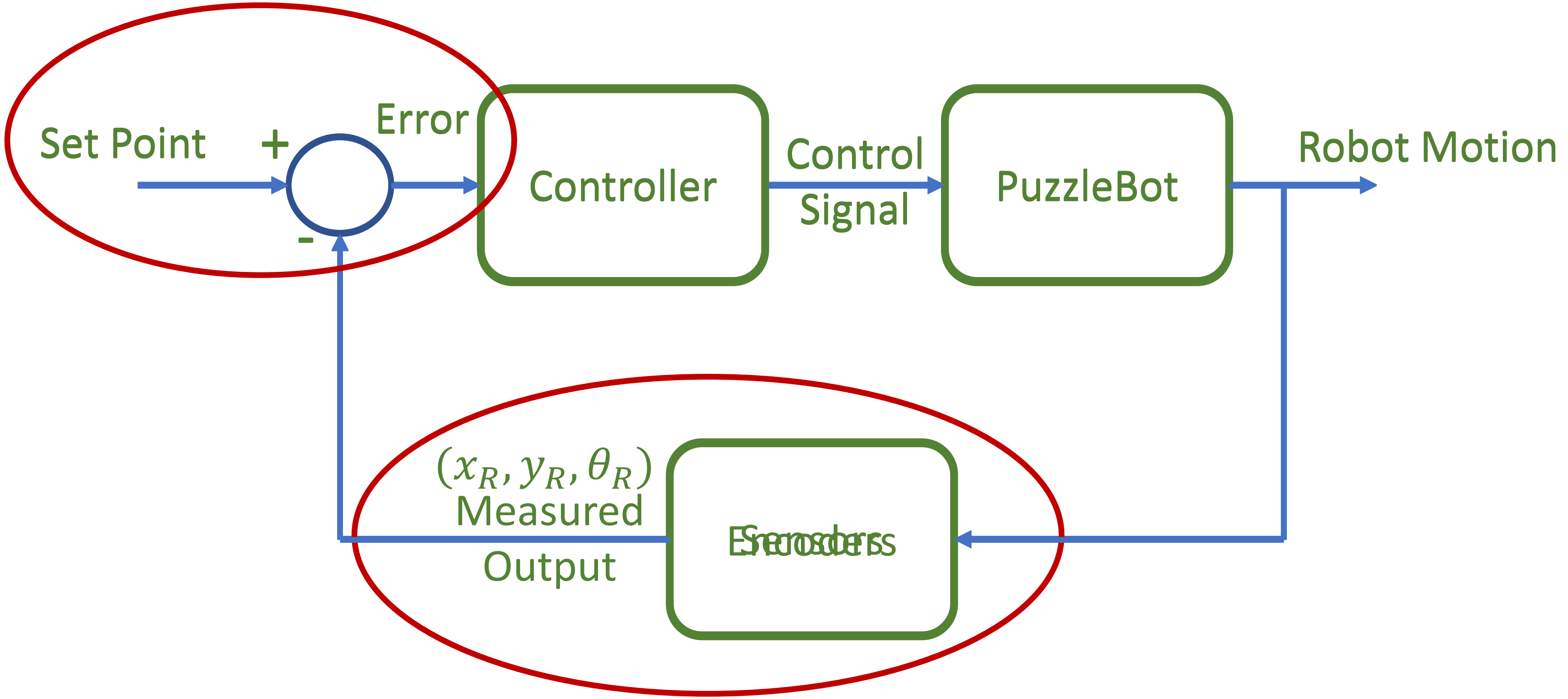
Or:

$$0 \leq \theta < 2\pi$$

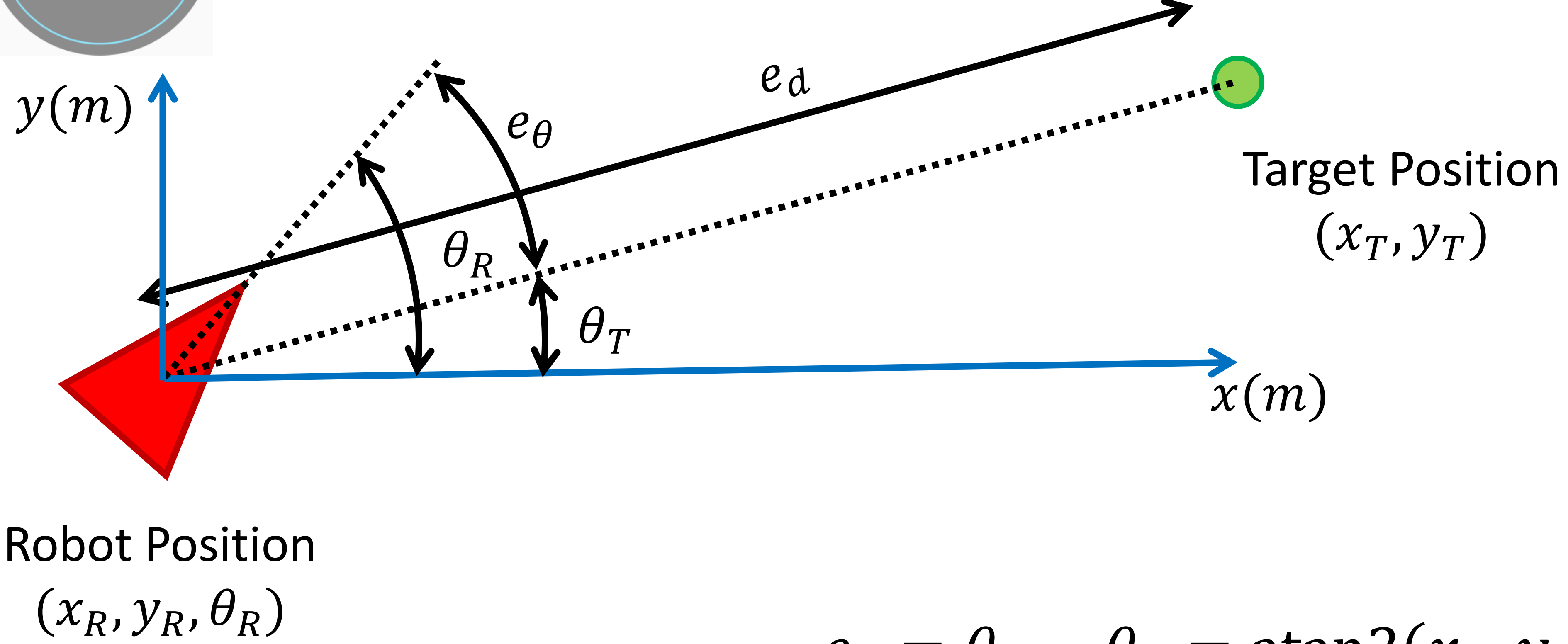
# Activity

- Implement a ROS node that computes the robot location using the encoder data
  - It should subscribe to `/wl` and `/wr`, and publish the data to a suitable set of topics
  - The published messages could be a set floats, or you can combine them in any way you see fit

# The Control System



# The Error

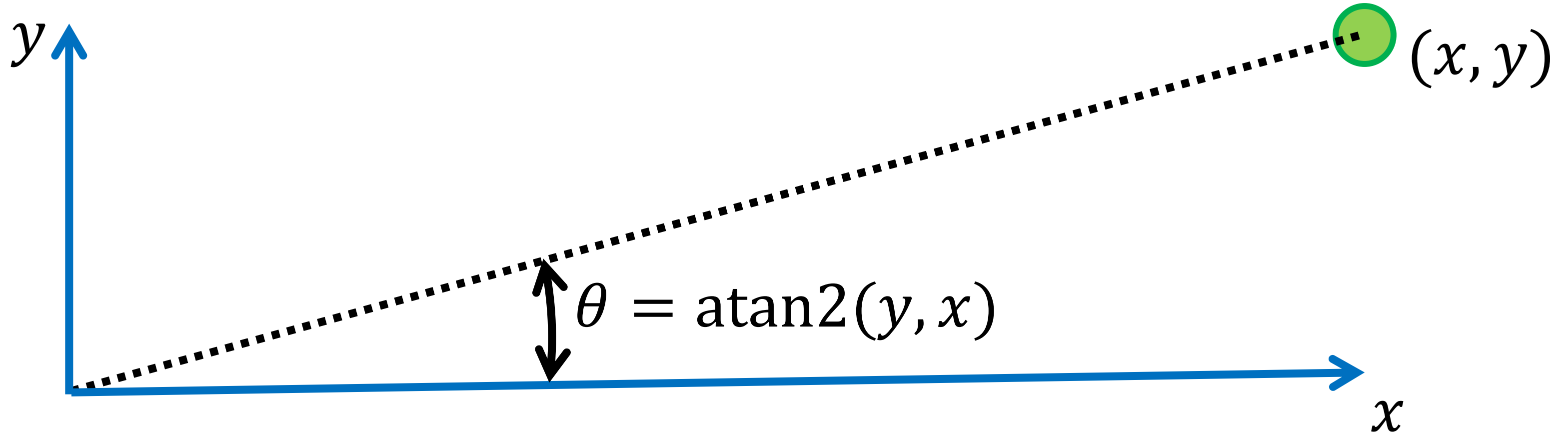



$$e_\theta = \theta_T - \theta_R = \mathrm{atan2}(x_T, y_T) - \theta_R$$

$$e_d = \sqrt{(x_T - x_R)^2 + (y_T - y_R)^2}$$

# atan2

- The atan2 function is a special form of arctan or $\tan^{-1}$.
- It takes two arguments, $y$ and $x$, and returns the angle to the $x$ axis:



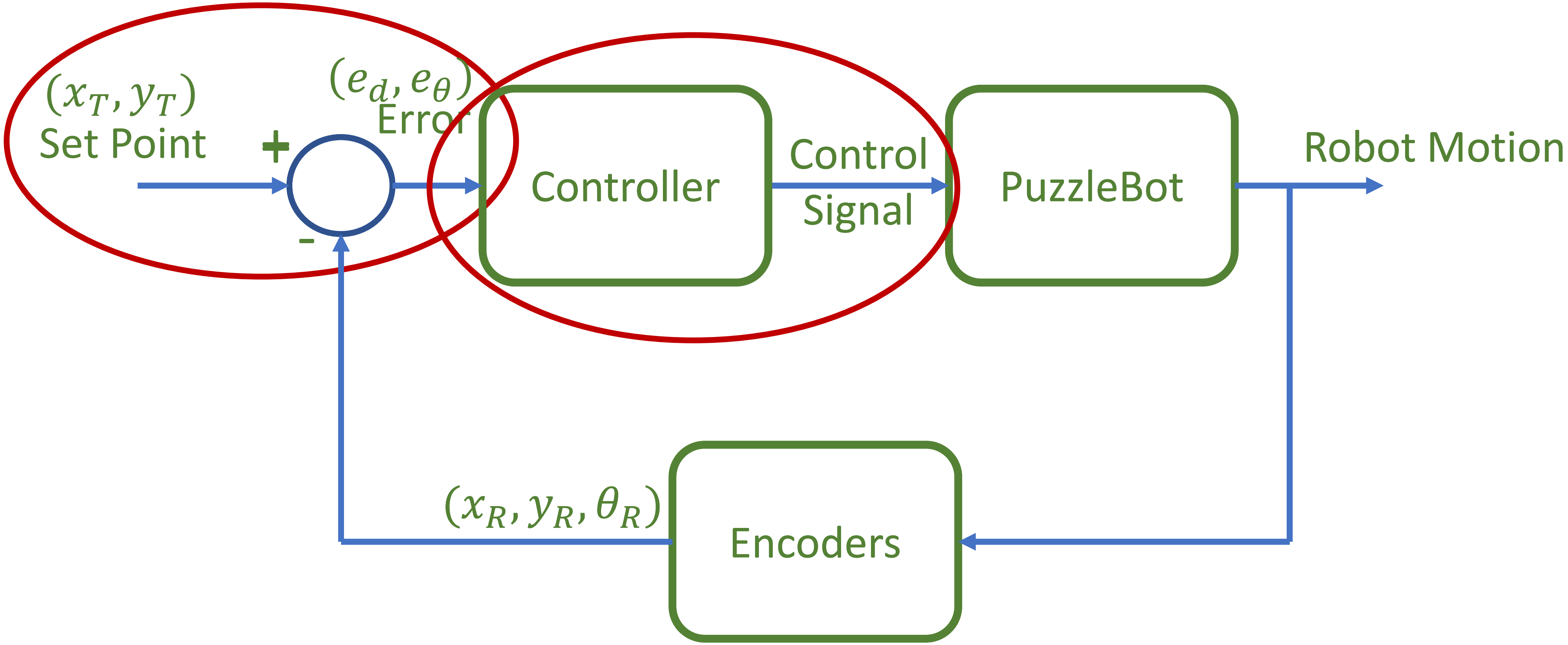

- It is included in most maths libraries, but it is recommended to use numpy, as numpy will be necessary later on in the course

```
import numpy
theta = numpy.arctan2(y,x)
```

# Activity

- Modify the previous node to publish ed and etheta.
- Set a target, and drive the robot around, checking that the angle to the target and the distance from the target are updated correctly
- Remember to wrap all angles to within 1 circle

# The Control System



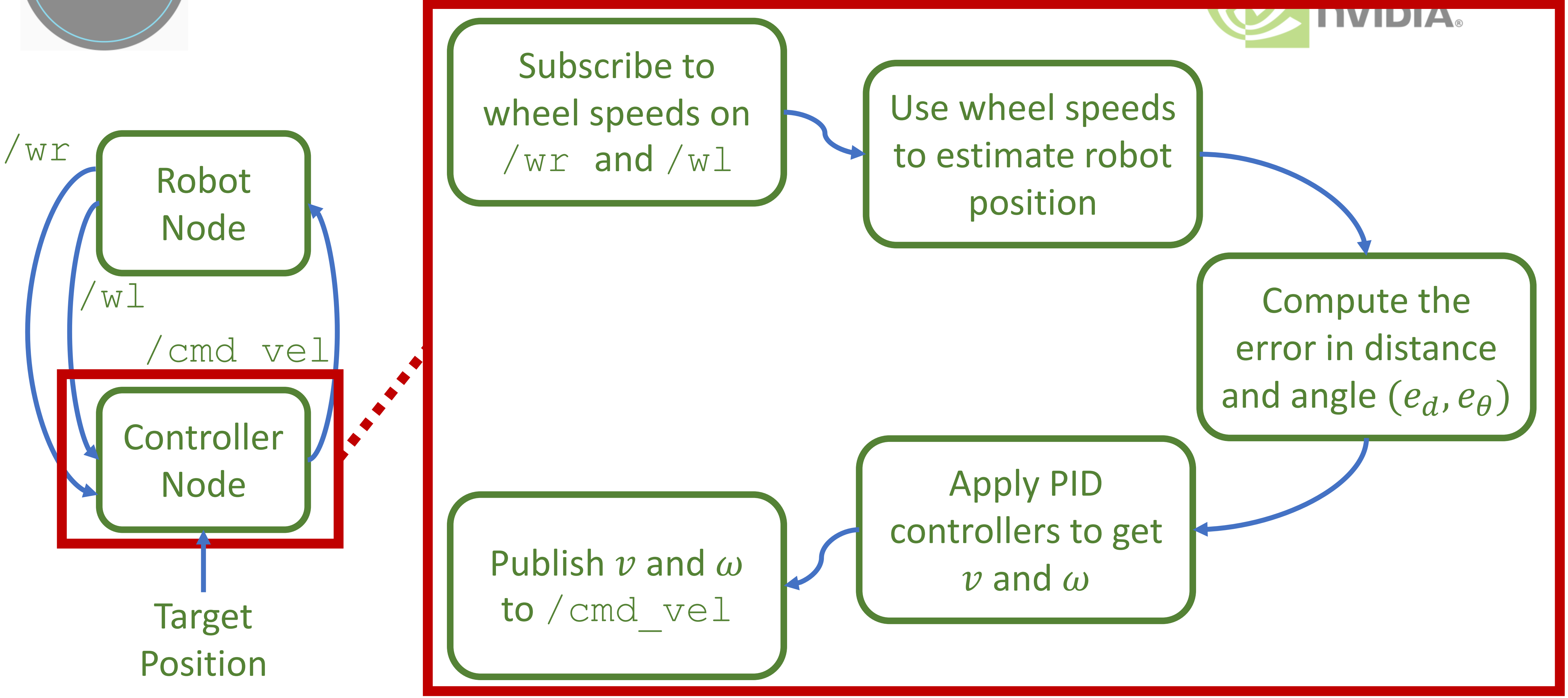
# The Controller

- Since the robot is inherently stable, a simple PID scheme should be sufficient.
- Start with a pair of proportional controllers:

$$v = K_v e_d$$
$$\omega = K_\omega e_\theta$$

… and add integral and derivative elements if necessary.

# ROS setup



/wr
Robot Node
/wl
/cmd_vel
Controller Node
Target Position
Subscribe to wheel speeds on /wr and /wl
Use wheel speeds to estimate robot position
Compute the error in distance and angle $(e_d, e_\theta)$
Apply PID controllers to get $v$ and $\omega$
Publish $v$ and $\omega$ to /cmd_vel

# /cmd_vel

- Message type: Geometry Messages – Twist
- Import command in python:
  - `from geometry_msgs.msg import Twist`
- 6 fields accessed as follows:
  - `msg.linear.x, msg.linear.y, msg.linear.z`
  - `msg.angular.x, msg.angular.y, msg.angular.z`

```
Vector3 linear
    float64 x
    float64 y
    float64 z
Vector3 angular
    float64 x
    float64 y
    float64 z
```

# Tips and Tricks

- Write and test your node with the PuzzleBot off the ground:
  - Use this to check the basics of your code are working correctly, such as the sign (+/-) of your controller parameters $K_v$ and $K_\omega$
  - Does the robot turn towards the goal?
  - Does the robot move towards or away from the goal?
- Tune one of the controllers at a time. You may find it easier to tune $K_\omega$ first, while setting your robot to move with a fixed forward speed.
- If in doubt, *lower* the value of the control constants.
- You may find it helpful to use a launch file to load your controller constants in from a config.yaml file.

# Accuracy

- It will not be possible to tune the controllers such that the robot moves perfectly into position.
  - You will need a threshold after which your algorithm decides it has successfully arrived.
  - Suggested initial threshold: 10 cm
- Additionally, if you measure the position of the robot, it will likely not match up with the measurement computed from the encoders.
  - This is inevitable due to additive noise in the encoder readings.
  - The solution to this is to use sensors that can measure the position of the robot relative to its environment.

# ROS Tools
## ROS Launch Syntax

- Launch files are sets of commands written in xml that allow executing various scripts at the same time.
- The general syntaxis is the following

```
<?xml version="1.0"?>
<launch>
   [Body of the launchfile]
</launch>
```

- This syntaxis allows to run any object used within the ROS architecture and has a wide variety of tools that allow to parametrize the launch file so that it can be adapted to the requirements of you project.
- An extensive documentation can be found in [http://wiki.ros.org/roslaunch](http://wiki.ros.org/roslaunch)

# ROS Tools
## ROS Launch code tools

- Running a node

```
<node name=“listener” pkg=“basic_comms” type=“listener.py” output=“screen”/>
```

- Running another file or launch file

```
<include file="$(dirname)/other.launch" />
```

- Set parameters

```
<param name="publish_frequency" type="double" value="10.0" />
```

- Pass args to the launch file

```
<arg name=”camera_id" value=”cam_3" />
```

- Load files into the system

```
<rosparam command="load" file="$(find package_name)/config/file_name.yaml" />
```

# The Task