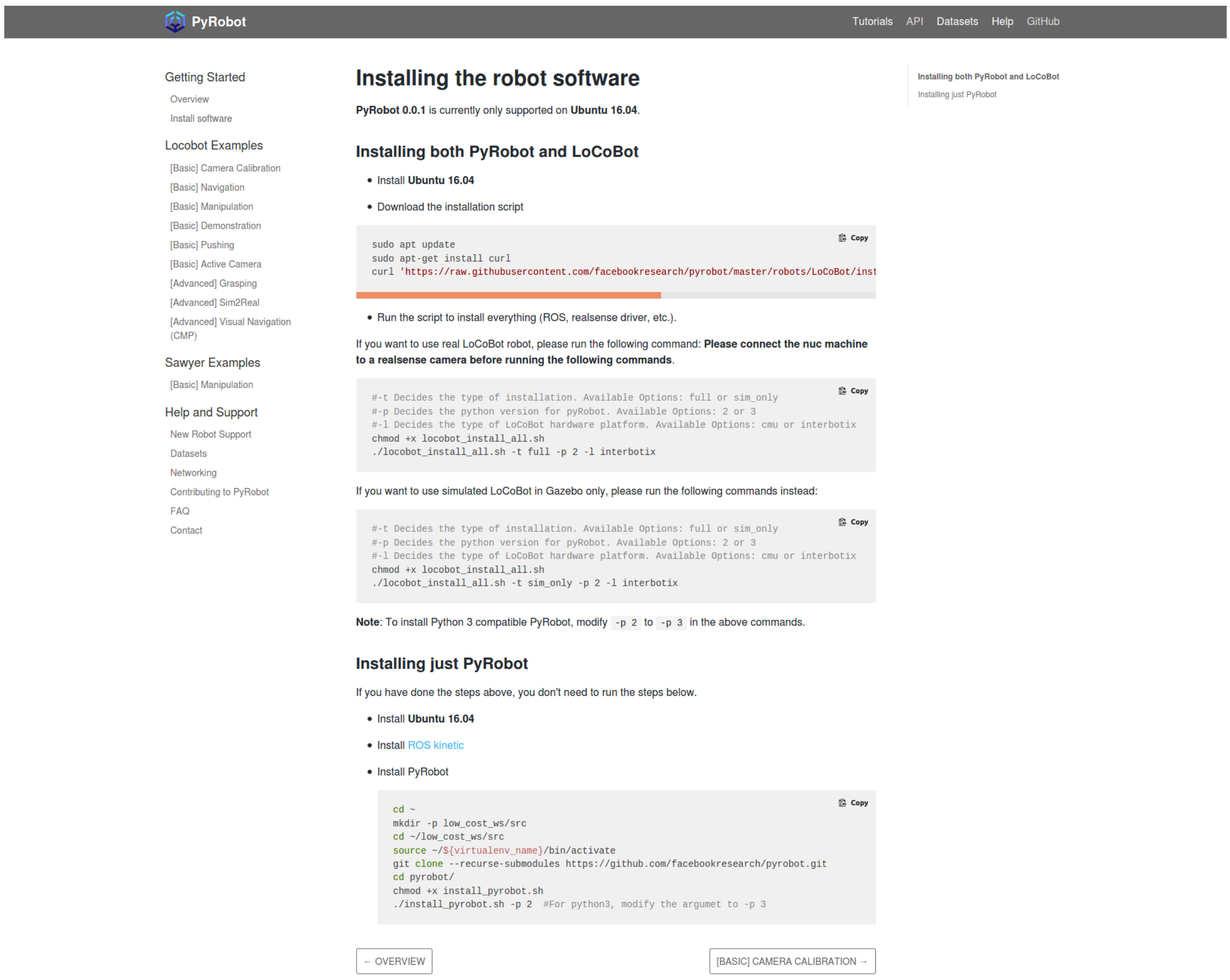
# Table of Contents

# Install Software



## Installing the robot software

**PyRobot 0.0.1** is currently only supported on **Ubuntu 16.04**.

### Installing both PyRobot and LoCoBot

- Install **Ubuntu 16.04**
- Download the installation script

```
sudo apt update
sudo apt-get install curl
curl 'https://raw.githubusercontent.com/facebookresearch/pyrobot/master/robots/LoCoBot/inst
```

- Run the script to install everything (ROS, realsense driver, etc.).

If you want to use real LoCoBot robot, please run the following command: **Please connect the nuc machine to a realsense camera before running the following commands**.

```
#-t Decides the type of installation. Available Options: full or sim_only
#-p Decides the python version for pyRobot. Available Options: 2 or 3
#-l Decides the type of LoCoBot hardware platform. Available Options: cmu or interbotix
chmod +x locobot_install_all.sh
./locobot_install_all.sh -t full -p 2 -l interbotix
```

If you want to use simulated LoCoBot in Gazebo only, please run the following commands instead:

```
#-t Decides the type of installation. Available Options: full or sim_only
#-p Decides the python version for pyRobot. Available Options: 2 or 3
#-l Decides the type of LoCoBot hardware platform. Available Options: cmu or interbotix
chmod +x locobot_install_all.sh
./locobot_install_all.sh -t sim_only -p 2 -l interbotix
```

**Note**: To install Python 3 compatible PyRobot, modify `-p 2` to `-p 3` in the above commands.

### Installing just PyRobot

If you have done the steps above, you don't need to run the steps below.

- Install **Ubuntu 16.04**
- Install ROS kinetic
- Install PyRobot

```
cd ~
mkdir -p low_cost_ws/src
cd ~/low_cost_ws/src
source ~/${virtualenv_name}/bin/activate
git clone --recurse-submodules https://github.com/facebookresearch/pyrobot.git
cd pyrobot/
chmod +x install_pyrobot.sh
./install_pyrobot.sh -p 2  #For python3, modify the argumet to -p 3
```

Installing both PyRobot and LoCoBot
Installing just PyRobot
← OVERVIEW
[BASIC] CAMERA CALIBRATION →

## Installing both PyRobot and LoCoBot

- Install **Ubuntu 16.04**
- Download the installation script

```
sudo apt update
sudo apt-get install curl
curl
'https://raw.githubusercontent.com/facebookresearch/pyrobot/master/robots/LoCoBo
t/install/locobot_install_all.sh' > locobot_install_all.sh
```

If you want to use real LoCoBot robot, please run the following command: **Please connect the nuc machine to a realsense camera before running the following commands.**

```
#-t Decides the type of installation. Available Options: full or sim_only
#-p Decides the python version for pyRobot. Available Options: 2 or 3
#-l Decides the type of LoCoBot hardware platform. Available Options: cmu or
interbotix
```

```
chmod +x locobot_install_all.sh
./locobot_install_all.sh -t full -p 2 -l interbotix
```

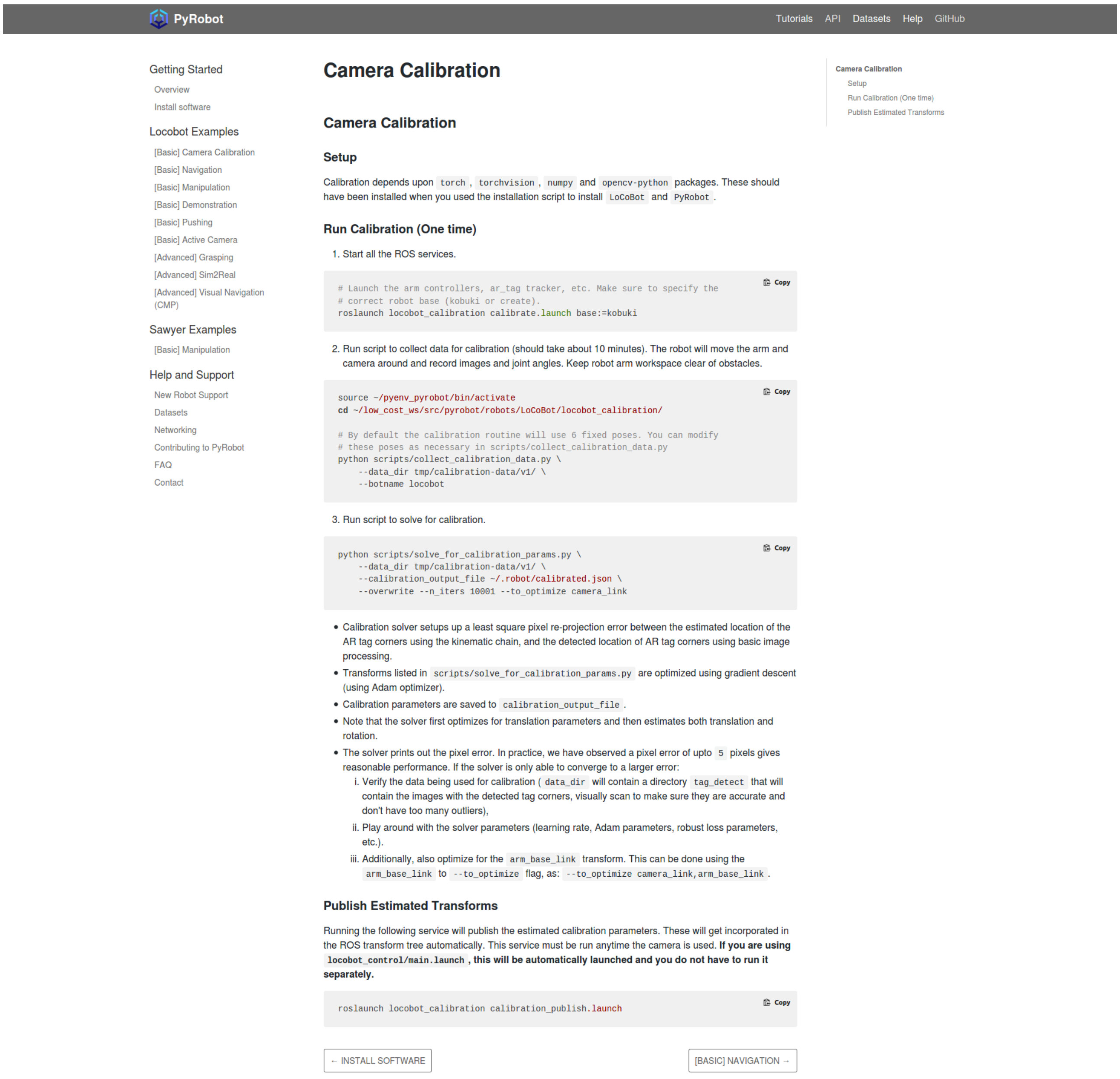
## Installing just PyRobot

If you have done the steps above, you don't need to run the steps below.

- Install **Ubuntu 16.04**
- Install ROS kinetic
- Install PyRobot

```
cd ~
mkdir -p low_cost_ws/src
cd ~/low_cost_ws/src
source ~/${virtualenv_name}/bin/activate
git clone --recurse-submodules https://github.com/facebookresearch/pyrobot.git
cd pyrobot/
chmod +x install_pyrobot.sh
./install_pyrobot.sh -p 2  #For python3, modify the argumet to -p 3
```

# [Basic] Camera Calibration



## Camera Calibration

### Camera Calibration

#### Setup

Calibration depends upon `torch` , `torchvision` , `numpy` and `opencv-python` packages. These should have been installed when you used the installation script to install `LoCoBot` and `PyRobot` .

#### Run Calibration (One time)

1. Start all the ROS services.

```
# Launch the arm controllers, ar_tag tracker, etc. Make sure to specify the
# correct robot base (kobuki or create).
roslaunch locobot_calibration calibrate.launch base:=kobuki
```

2. Run script to collect data for calibration (should take about 10 minutes). The robot will move the arm and camera around and record images and joint angles. Keep robot arm workspace clear of obstacles.

```
source ~/pyenv_pyrobot/bin/activate
cd ~/low_cost_ws/src/pyrobot/robots/LoCoBot/locobot_calibration/

# By default the calibration routine will use 6 fixed poses. You can modify
# these poses as necessary in scripts/collect_calibration_data.py
python scripts/collect_calibration_data.py \
    --data_dir tmp/calibration-data/v1/ \
    --botname locobot
```

3. Run script to solve for calibration.

```
python scripts/solve_for_calibration_params.py \
    --data_dir tmp/calibration-data/v1/ \
    --calibration_output_file ~/.robot/calibrated.json \
    --overwrite --n_iters 10001 --to_optimize camera_link
```

- Calibration solver setups up a least square pixel re-projection error between the estimated location of the AR tag corners using the kinematic chain, and the detected location of AR tag corners using basic image processing.
- Transforms listed in `scripts/solve_for_calibration_params.py` are optimized using gradient descent (using Adam optimizer).
- Calibration parameters are saved to `calibration_output_file` .
- Note that the solver first optimizes for translation parameters and then estimates both translation and rotation.
- The solver prints out the pixel error. In practice, we have observed a pixel error of upto `5` pixels gives reasonable performance. If the solver is only able to converge to a larger error:
    i. Verify the data being used for calibration ( `data_dir` will contain a directory `tag_detect` that will contain the images with the detected tag corners, visually scan to make sure they are accurate and don't have too many outliers),
    ii. Play around with the solver parameters (learning rate, Adam parameters, robust loss parameters, etc.).
    iii. Additionally, also optimize for the `arm_base_link` transform. This can be done using the `arm_base_link` to `--to_optimize` flag, as: `--to_optimize camera_link,arm_base_link` .

#### Publish Estimated Transforms

Running the following service will publish the estimated calibration parameters. These will get incorporated in the ROS transform tree automatically. This service must be run anytime the camera is used. **If you are using `locobot_control/main.launch` , this will be automatically launched and you do not have to run it separately.**

```
roslaunch locobot_calibration calibration_publish.launch
```

## Camera Calibration

### Setup

Calibration depends upon `torch`, `torchvision`, `numpy` and `opencv-python` packages. These should have been installed when you used the installation script to install `LoCoBot` and `PyRobot`.

### Run Calibration (One time)

1. Start all the ROS services.

```
# Launch the arm controllers, ar_tag tracker, etc. Make sure to specify the
# correct robot base (kobuki or create).
roslaunch locobot_calibration calibrate.launch base:=kobuki
```

2. Run script to collect data for calibration (should take about 10 minutes). The robot will move the arm and camera around and record images and joint angles. Keep robot arm workspace clear of obstacles.

```
source ~/pyenv_pyrobot/bin/activate
cd ~/low_cost_ws/src/pyrobot/robots/LoCoBot/locobot_calibration/

# By default the calibration routine will use 6 fixed poses. You can modify
# these poses as necessary in scripts/collect_calibration_data.py
python scripts/collect_calibration_data.py \
    --data_dir tmp/calibration-data/v1/ \
    --botname locobot
```

3. Run script to solve for calibration.

```
python scripts/solve_for_calibration_params.py \
    --data_dir tmp/calibration-data/v1/ \
    --calibration_output_file ~/.robot/calibrated.json \
    --overwrite --n_iters 10001 --to_optimize camera_link
```

- Calibration solver setups up a least square pixel re-projection error between the estimated location of the AR tag corners using the kinematic chain, and the detected location of AR tag corners using basic image processing.
- Transforms listed in `scripts/solve_for_calibration_params.py` are optimized using gradient descent (using Adam optimizer).
- Calibration parameters are saved to `calibration_output_file`.
- Note that the solver first optimizes for translation parameters and then estimates both translation and rotation.
- The solver prints out the pixel error. In practice, we have observed a pixel error of upto `5` pixels gives reasonable performance. If the solver is only able to converge to a larger error:
    1. Verify the data being used for calibration (`data_dir` will contain a directory `tag_detect` that will contain the images with the detected tag corners, visually scan to make sure they are accurate and don't have too many outliers),
    2. Play around with the solver parameters (learning rate, Adam parameters, robust loss parameters, etc.).
    3. Additionally, also optimize for the `arm_base_link` transform. This can be done using the `arm_base_link` to `--to_optimize` flag, as: `--to_optimize camera_link,arm_base_link`.

## Publish Estimated Transforms

Running the following service will publish the estimated calibration parameters. These will get incorporated in the ROS transform tree automatically. This service must be run anytime the camera is used. **If you are using `locobot_control/main.launch`, this will be automatically launched and you do not have to run it separately.**

```
roslaunch locobot_calibration calibration_publish.launch
```

# [Basic] Navigation

## Setup

This tutorial can also be run using a simulated version of the robot. Before we get started with this, ensure the following:

- The robot base is supported by PyRobot. Check if your robot is supported here.
- The robot base is switched ON. With the LoCoBot the base beeps when connected.
- The robot's launch file has been run. Note that you have to set `use_base:=true`.

LoCoBot[Real Robot]

```
roslaunch locobot_control main.launch use_base:=true
```

LoCoBot[Simulator]

```
roslaunch locobot_control main.launch use_base:=true use_sim:=true
```

- The appropriate python virtual environment has been sourced.

```
load_pyrobot_env
```

## Base State

Base state is represented by `[x, y, yaw]`, the `x`-coordinate, `y`-coordinate of the `base_link` and the robot heading `yaw`. By default this state is estimated via intertial sensors, or wheel encoders. This state can be queried as follows (You can run the following code in a **python** terminal):

Source virtual env

```
#import appopriate dependencies
from pyrobot import Robot

# Create robot.
robot = Robot('locobot')

# Get the current pose of the base i.e, a state of the form (x, y, yaw). Note
# that over here we are querying the 'odom' state, which is state estimated
# from inertial sensors and wheel encoders on the base.
current_state = robot.base.get_state('odom')

# (Advanced) If you are running visual SLAM, then you can also query the state
# as estimated from visual sensors, using the following. State estimates from
# visual SLAM can be more accurate.
current_state = robot.base.get_state('vslam')
```

# Basic Control

In this section, we will talk about two modes of controlling the LoCoBot base- Velocity control and Position control. One should note the same tutorials also apply to LoCoBot-Lite with very minimal change to the code.

We will talk about velocity and position control by introducing examples.

## Velocity control

This mode of control allows us to command the base with a particular linear and angular velocity for a specified ammount of time.

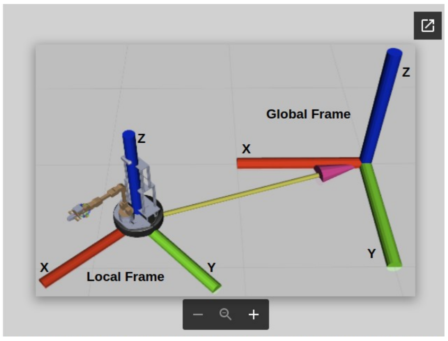
An example script that performs velocity control,

Source virtual env

```
#import appopriate dependencies
from pyrobot import Robot

# Create the Robot object that interfaces with the robot.
robot = Robot('locobot')
# If you want to use LoCoBot-Lite instead, replace the argument 'locobot' with
# 'locobot_lite'

# linear_velocity in m/s
linear_velocity = 0.1

# rotational_velocity in radian / s
rotational_velocity = 0.5

# execution_time in seconds
execution_time = 4

# Command to execute motion
robot.base.set_vel(fwd_speed=linear_velocity,
                   turn_speed=rotational_velocity,
                   exe_time=execution_time)

# To stop the robot at any time:
robot.base.stop()
```

## Frames of reference

As shown in the figure below, there are two frames of reference available for the base - Local and Global frames.



Local frame is the frame of reference attached to the base of the robot and moves with it as the robot moves i.e, all the points in this frame are relative to the robot.

Global frame is a stationary frame of reference. It is the initial frame that the robot started at.

## Position control

This mode of control allows us to command the base to go to a specified target (of the form `[x, y, yaw]`) in the environment.

We currently support three different base-controllers for position control: `ILQR`, `Proportional` and `Movebase`.

Following code shows an example of position control.

```
from pyrobot import Robot

# base_config_dict is a dictionary that contains different base configuration
# parameters. 'base_controller' can be set to 'ilqr' or 'proportional' or
# 'movebase' to use the respective controllers.
base_config_dict={'base_controller': 'ilqr'}

# crate the Robot object with the specified base_config
robot = Robot('locobot', base_config=base_config_dict)

# target position we want the base to go to
target_position = [1,1,0.5] # this is a 2D pose of the form [x, y, yaw]

# Now command the robot to go to the target pose in the enviroment
# 'go_to_absolute' assumes that the target is in world frame.
robot.base.go_to_absolute(target_position)

# Targets can be specified in robot's coordinate frame.
# robot.base.go_to_relative(target_position)
```

As shown below, we can also modify go_to_absolute or go_to_relative function arguments to enable or disable different features.

```
# smooth ensures that the robot only follows smooth motions while going to goal

# smooth mean no on-spot rotations.
robot.base.go_to_absolute(target_position, smooth=True)

# close_loop ensures that the controller acts in closed loop by using onboard
# odommetry
robot.base.go_to_absolute(target_position, close_loop=True)
```

Below are few different position control examples.

Example 1

```
from pyrobot import Robot

base_config_dict={'base_controller': 'proportional'}
robot = Robot('locobot', base_config=base_config_dict)
target_position = [1.0, 0.0, 0.0] # go forward 1 meter
robot.base.go_to_relative(target_position, smooth=False, close_loop=True)
```

Example 2

```
from pyrobot import Robot

base_config_dict={'base_controller': 'ilqr'}
robot = Robot('locobot', base_config=base_config_dict)
target_position = [-1.0, 0.0, 0.0] # go reverse 1 meter
robot.base.go_to_relative(target_position, smooth=False, close_loop=True)
```

Example 3

```
from pyrobot import Robot

base_config_dict={'base_controller': 'movebase'}
robot = Robot('locobot', base_config=base_config_dict)
target_position = [0.0, 0.0, 1.5707] # rotate on-spot by 90 degrees
robot.base.go_to_relative(target_position, smooth=False, close_loop=True)
```

Example 4

```
from pyrobot import Robot

base_config_dict={'base_controller': 'ilqr'}
robot = Robot('locobot', base_config=base_config_dict)
target_position = [1.0, 1.0, 0.0]
robot.base.go_to_relative(target_position, smooth=False, close_loop=True)
robot.base.go_to_relative(target_position)
```

## Trajectory Tracking

Feedback controllers implemented in the library can be used to close the loop on a given state-space trajectory. The base state is characterized by `[x, y, yaw]`, the `x`-coordinate, `y`-coordinate and the robot heading `yaw`. Given a state-space trajectory (at the frequency at which the tracker is run), the `track_trajectory` function can close the loop on the trajectory.

```
from pyrobot import Robot

import numpy as np
base_config_dict={'base_controller': 'ilqr'}
robot = Robot('locobot', base_config=base_config_dict)

# Generate a straight line trajectory, for the robot to track. Note that this
# trajectory needs to be generated at the frequency of the feedback controller
# that will track it.

# We need to generate a trajectory at the following rate.
dt = 1. / robot.configs.BASE.BASE_CONTROL_RATE

# We will generate a trajectory 1m long and track it such that the robot moves
# at half the max speed of the robot.
v = robot.configs.BASE.MAX_ABS_FWD_SPEED / 2.

# distance moved per time step
r = v * dt

# number of time steps
t = int(1/r)

# initial state
x, y, yaw = robot.base.get_state('odom')

# generate state trajectory
states = []
for i in range(t):
    states.append([x+r*np.cos(yaw)*i, y+r*np.sin(yaw)*i, yaw])
states = np.array(states)

# Call trajectory tracker
robot.base.track_trajectory(states, close_loop=True)
```

In this example, we tracked a very simple trajectory, but the implementation can track more complex trajectories as well. More a more advanced example see here.

## Position control with map (Real robot)

Position control with map is an enhanced position control feature that allows us to leverage the map contructed by the onboard SLAM algorithms while going to a specific target in the environement. This feature allows to avoid obstacles in the environment while going to a target postion as the robot only travels through the free space deemed worthy by the onboard SLAM.

To use this feature we need to modify the initially specified launch file arguments as follows,

LoCoBot[Real Robot]

```
roslaunch locobot_control main.launch use_base:=true use_vslam:=true
use_camera:=true
```

**Warning:**

1. After running the above launch command, please do not move in front of the camera or temporarily block the camera view. These actions would be registered as permanent obstacles by SLAM which is running perpetualy in the background. The SLAM algorithm here does not deal with the dynamic obstacles.
2. SLAM only works in environments that have rich RGB feature points and could fail otherwise.
3. While using PyRobot, if you launch the robot with different settings, you need to exit the python terminal, and import PyRobot in a new python terminal.

Here is an example showing position control with map in action.

```
from pyrobot import Robot


# 'base_planner' is the planner used to generate collision free plans to goal
base_config_dict={'base_controller': 'proportional', 'base_planner':'movebase'}

robot = Robot('locobot', base_config=base_config_dict)

# 'use_map' argument determines if map should be used or not in position
control.
robot.base.go_to_relative([2.0, 0.0, 0.0], use_map=True, smooth=False,
close_loop=True)
```

# [Basic] Manipulation

## Setup

This tutorial can also be run using a simulated version of the robot. Before we get started with this, ensure the following:

- The robot arm is supported by PyRobot. Check if your robot is supported here.
- The robot arm is switched ON. With the LoCoBot this is done by connecting the power supply and USB to the arm.

LoCoBot Setup Instructions:

LoCoBot's launch file has been run. Note that you have to set `use_arm:=true`.

```
roslaunch locobot_control main.launch use_arm:=true
```

Similar to the real robot, for LoCoBot gazebo simulator, run the following command,

```
roslaunch locobot_control main.launch use_arm:=true use_sim:=true
```

- The appropriate python virtual environment has been sourced before running any PyRobot package.

Source virtual env

```
load_pyrobot_env
```

## Basic movements

In your favorite Python command shell run the following to setup the robot object

LoCoBot Setup Instructions

```
from pyrobot import Robot
import numpy as np
impot time
robot = Robot('locobot')
```

This creates a `Robot` object that encapsulates the robot's basic motion utilities.

### Joint control

To move the joints of the robot to a desired joint configuration, run the following snippet:

LoCoBot

```
target_joints = [[0.408, 0.721, -0.471, -1.4, 0.920],[-0.675, 0, 0.23, 1,-0.70]]
robot.arm.go_home()
for joint in target_joints:
    robot.arm.set_joint_positions(joint, plan=False)
    time.sleep(1)
robot.arm.go_home()
```

`Robot.arm.go_home()` makes the arm to move to its *home* position. Since we are using a 5-joint (DoF, degree-of-freedom) arm on the LoCoBot, the `target_joint` is a 5D vector of desired individual joint angles from the base of the arm to its wrist. Then finally through the `set_joint_positions` method the Robot will move to the desired `target_joint`. The `plan=False` argument means that the robot will not use MoveIt to plan around obstacles (like the base or the arm itself). To plan around obstacles, look at using MoveIt.

## End-effector pose control

In this example, we will look at controlling the end-effector pose of the arm. A *pose* object has two components: *position* and *rotation*. *Position* is a 3D numpy array representing the desired position. *Orientation* can be a rotation matrix [3x3], euler angles [3,], or quaternion [4,].

LoCoBot

```
import time
target_poses = [{'position': np.array([0.279, 0.176, 0.217]),
                 'orientation': np.array([[0.5380200, -0.6650449, 0.5179283],
                                          [0.4758410, 0.7467951, 0.4646209],
                                          [-0.6957800, -0.0035238,
0.7182463]])},
                {'position': np.array([0.339, 0.0116, 0.255]),
                 'orientation': np.array([0.245, 0.613, -0.202, 0.723])},
                ]
robot.arm.go_home()
for pose in target_poses:
    robot.arm.set_ee_pose(**pose)
    time.sleep(1)
robot.arm.go_home()
```

Note that since the LoCoBot only has 5 DoFs, it can only reach target poses that lie in its configuration space. Check the API for more on how to use the method `set_ee_pose`.

## End-effector Position and Pitch Roll Control

### This is a LoCoBot-specific example.

As LoCoBot is a 5-DOF robot, you can specify its pose with an end-effector position (x,y,z), a pitch angle and a roll angle (no need to specify yaw angle as it only has 5 degrees of freedom).

```
target_poses = [
    {'position': np.array([0.28, 0.17, 0.22]),
     'pitch': 0.5,
     'numerical': False},
    {'position': np.array([0.28, -0.17, 0.22]),
     'pitch': 0.5,
     'roll': 0.5,
     'numerical': False}]
```

```
robot.arm.go_home()

for pose in target_poses:
    robot.arm.set_ee_pose_pitch_roll(**pose)
    time.sleep(1)
robot.arm.go_home()
```

## End-effector Cartesian Path control

In this example, we will move the arm in the X,Y,Z coordinates in straight-line paths from the current pose.

LoCoBot

```
robot.arm.go_home()
displacement = np.array([0, 0, -0.15])
robot.arm.move_ee_xyz(displacement, plan=True)
robot.arm.go_home()
```

If `plan=True`, it will call the internal cartesian path planning in MoveIt.

If `plan=False`, it will simply perform linear interpolation along the target straight line and do inverse kinematics (you can choose whether you want to use the numerical inverse kinematics or analytical inverse kinematics by passing `numerical=True` or `numerical=False`) on each waypoints. Since LoCoBot is a 5-DOF robot, the numerical inverse kinematics sometimes fail to find the solution even though there exists a solution. So analytical inverse kinematics might work better in such cases.

## Joint torque control

**Warning for usage.** Each though each joint accepts the torque command, there is no gravity compensation implemented for LoCoBot yet. Torque control mode is not recommended for LoCoBot and it's not well tested. Sawyer does support torque control and it's well tested.

For direct torque control, one can use the `set_joint_torques` method as follows. Firstly, you will need to kill the previously launched driver and relaunched it with the following command.

LoCoBot Only

```
roslaunch locobot_control main.launch use_arm:=true torque_control:=true
```

Then you can use the following command to send torque values to robots. Try to keep the arm in initial condition as shown in the below video, as the behavior will be different for a different configuration. For this example, we are going to apply torque only on joint 4. This will move robot joint 4 to the extreme. After completion, the joint will be free again. The torque requirements may vary from robot to robot. So if joint 4 doesn't move using following script, try to apply a higher magnitude of torque.

LoCoBot

```
from pyrobot import Robot
import time
arm_config = dict(control_mode='torque')
robot = Robot('locobot', arm_config=arm_config)
target_torque = 4 * [0]
target_torque[3] = -0.45
robot.arm.set_joint_torques(target_torque)
time.sleep(2)
target_torque[3] = 0.0
robot.arm.set_joint_torques(target_torque)
```

### Gripper control

Opening and closing the gripper is done via the `gripper` object.

```
import time

robot.gripper.open()
time.sleep(1)

robot.gripper.close()
```

## Planning using MoveIt

To avoid hitting obstacles like the base or the arm itself, we support planning via MoveIt!. To use this, we need to first set the robot with the approriate planning parameters:

LoCoBot

```
config = dict(moveit_planner='ESTkConfigDefault')
robot = Robot('locobot', arm_config=config)
```

After this run `set_joint_positions` with the argument `plan=True`.

LoCoBot

```
target_joints = [
        [0.408, 0.721, -0.471, -1.4, 0.920],
        [-0.675, 0, 0.23, 1, -0.70]
    ]
robot.arm.go_home()
for joint in target_joints:
    robot.arm.set_joint_positions(joint, plan=True)
    time.sleep(1)
robot.arm.go_home()
```

## Sensing

Most arms come with proprioceptive feedback. The following functions can be used to read the current state of the robot (joint angles, velocities, torques), etc.

```
# Get all joint angles of the robot
current_joints = robot.arm.get_joint_angles()
```

```
# Get state of a specific joint
current_joint = robot.arm.get_joint_angle("joint_5")

# Get all joint velocities of the robot
current_velocity = robot.arm.get_joint_velocities()

# Get current joint torques (if mode='torque')
current_torques = robot.arm.get_joint_torques()
```

# [Basic] Demostration

## Getting started

- Make sure robot is turned ON
- Launch the robot server. Note that you have to set `teleop:=true`. Pass in the corresponding flags when launching the robot.

LoCoBot Arm [Real]

```
# You will only be able to control the arm (no base and no camera)

roslaunch locobot_control main.launch use_arm:=true teleop:=true
```

Arm+Base [Real]

```
# You will only be able to control the arm and the base

roslaunch locobot_control main.launch use_arm:=true use_base:=true teleop:=true
```

Arm+Base+Camera [Real]

```
# You will only be able to control the arm, the base, and the camera

roslaunch locobot_control main.launch use_arm:=true use_base:=true
use_camera:=true teleop:=true
```

Arm+Base+Camera [Gazebo]

```
roslaunch locobot_control main.launch use_arm:=true use_sim:=true teleop:=true
use_camera:=true use_base:=true
```

- Start the teleop server in a new terminal

```
load_pyrobot_env
cd ~/low_cost_ws/src/pyrobot/robots/LoCoBot/locobot_control/nodes
python robot_teleop_server.py
```

- Now we can run the teleoperation client (in a new terminal). The screen should display the relevant commands to control the arm.

```
load_pyrobot_env
cd ~/low_cost_ws/src/pyrobot/robots/LoCoBot/locobot_control/nodes
python keyboard_teleop_client.py
```

# [Basic] Pushing

## Get Started

**Note** This tutorial is tested to work only with Python2.7 version on PyRobot

- Make sure robot is turned ON
- Launch the robot driver

Launch driver

```
roslaunch locobot_control main.launch use_arm:=true use_camera:=true
```

- Run the pushing example

`pushing.py` script accepts two parameters: `floor_height` and `gripper_len`. You need to tune these two parameters a bit. `floor_height` means the `z`-coordinate of the floor. Points with `z`-coordinate smaller than this value will be filtered. Only points heigher than the floor will be used for clustering. `gripper_len` means the length of the gripper (from `gripper_link` to the tip of the gripper). You may need to tune these two parameters when you run the example.

If the gripper goes down too much and it hits the floor while pushing the object, you can make the `gripper_len` bigger. If the gripper doesn't go down enough and it doesn't touch the object, you can try making the `gripper_len` smaller.

To tune `floor_height`, you need to run the following script first.

Visualize filtered point cloud

```
source ~/pyenv_pyrobot/bin/activate
cd ~/low_cost_ws/src/pyrobot/examples/locobot/manipulation
python realtime_point_cloud.py --floor_height=0.01
```

This script shows the point cloud in the scene after filtering (points with depth more than 1.5m and points with `z` coordinate (height) larger than `floor_height` will be removed). The remaining point cloud will be used for clustering and pushing in `pushing.py`. So you need to tune the `floor_height` such that the floor is completely removed (which means its value cannot be too small) and as much points of the the objects as possible are remained (which means its value cannot be too big).

After tuning, you can run the pushing script in a new terminal as follows:

Run pushing script

```
source ~/pyenv_pyrobot/bin/activate
cd ~/low_cost_ws/src/pyrobot/examples/locobot/manipulation
python pushing.py --floor_height=0.01 --gripper_len=0.12
```

**Warning**: we are using the analytical inverse kinematics in this example, which means no collision checking is performed here. So the robot arm may hit the robot itself sometimes.

# [Basic] Active Camera

## Getting started

- Make sure robot is turned ON
- Launch the robot driver

Launch driver

```
roslaunch locobot_control main.launch use_camera:=true
```

- `Start the python virtual environment`

Source virtual env

```
load_pyrobot_env
```

## Example usage

- Initialize a robot instance

Create robot

```
from pyrobot import Robot
bot = Robot('locobot')
bot.camera.reset()
```

- Set camera pan or tilt angle, or set pan and tilt angles together

Set Pan

```
pan = 0.4
bot.camera.set_pan(pan, wait=True)
```

Set Tilt

```
tilt = 0.4
bot.camera.set_tilt(tilt, wait=True)
```

Set Pan and Tilt

```
camera_pose = [0, 0.7]
bot.camera.set_pan_tilt(camera_pose[0], camera_pose[1], wait=True)
```

- Get rgb or depth images

RGB

```
from pyrobot.utils.util import try_cv2_import

cv2 = try_cv2_import()
rgb = bot.camera.get_rgb()
cv2.imshow('Color', rgb[:, :, ::-1])
cv2.waitKey(3000)
```

Depth

```
from pyrobot.utils.util import try_cv2_import

cv2 = try_cv2_import()
import numpy as np
depth = bot.camera.get_depth()
actual_depth_values = depth.astype(np.float64) / 1000.0
cv2.imshow('Depth', depth)
cv2.waitKey(3000)
```

RGB and Depth

```
from pyrobot.utils.util import try_cv2_import

cv2 = try_cv2_import()
rgb, depth = bot.camera.get_rgb_depth()
cv2.imshow('Color', rgb[:, :, ::-1])
cv2.imshow('Depth', depth)
cv2.waitKey(3000)
```

- Get 3D point coordinates of pixels in RGB image

In manipulation, we often get into the case where we want to know where an RGB pixel is in the 3D world, meaning the correspondence between RGB pixels and their 3D locations with respect to the robot. In PyRobot, we have provided a utility function (`pix_to_3dpt`) to do this transformation. This function takes as input the row indices and column indices of the pixels. It will return the 3D point locations and the RGB color values of these pixels. You can specify `in_cam=True` to get the 3D locations in the camera frame. Otherwise, the 3D locations will be with respect to the **LoCoBot base frame**.

Pixel via Scalar

```
bot.camera.set_tilt(0.6)
r = 295 # row number in the RGB image
c = 307 # column number in the RGB image
pt, color = bot.camera.pix_to_3dpt(r,c)
print('3D point:', pt)
print('Color:', color)
rgb = bot.camera.get_rgb()
cv2.imshow('Color', rgb[:, :, ::-1])
cv2.waitKey(10000)
```

## Pixels via List

```
bot.camera.set_tilt(0.6)
r = [295, 360]
c = [307, 296]
# this will return the point and color of
# pixel (295, 307) and pixel (360, 296)
pt, color = bot.camera.pix_to_3dpt(r,c)
print('3D point:', pt)
print('Color:', color)
rgb = bot.camera.get_rgb()
cv2.imshow('Color', rgb[:, :, ::-1])
cv2.waitKey(10000)
```

## Pixels via Numpy Array

```
import numpy as np

bot.camera.set_tilt(0.6)
r = np.array([295, 360])
c = np.array([307, 296])
# this will return the point and color of
# pixel (295, 307) and pixel (360, 296)
pt, color = bot.camera.pix_to_3dpt(r,c)
print('3D point:', pt)
print('Color:', color)
rgb = bot.camera.get_rgb()
cv2.imshow('Color', rgb[:, :, ::-1])
cv2.waitKey(10000)
```

# [Advanced] Grasping

## Getting started

Here we demonstrate how the PyRobot API can be used to grasp objects by using a learned model. Ideally PyRobot has already been installed, which will allow us to use the Python API for basic robot operations. Before we get started with this, ensure the following:

- The robot arm is supported by PyRobot. Check if your robot is supported here.
- The robot arm is switched ON. With the LoCoBot this is done by connecting the power supply and USB to the arm.
- Setup the virtual environment. Since the grasp models we are using need PyTorch, we need to install it in a new virtual environment. To do this easily, run the following lines.

```
virtualenv_name="pyenv_locobot_grasping"
virtualenv --system-site-packages -p python ~/${virtualenv_name}
source ~/${virtualenv_name}/bin/activate
cd ~/low_cost_ws/src/pyrobot
pip install -e .
cd ~/low_cost_ws/src/pyrobot/examples/grasping
pip install -r requirements.txt
```

- The robot's launch file has been run. Note that you have to set `use_arm:=true` and `use_camera:=true`.

```
roslaunch locobot_control main.launch use_arm:=true use_camera:=true
```

## Running the example

We use a sampling based grasping algorithm to grasp objects using the LoCoBot. The grasping script `locobot.py` accepts 4 parameters: `n_grasps`, `n_samples`, `patch_size`, and `no_visualize`. `n_grasps` is the number of times the robot attempts a grasp. `n_samples` is the number of samples of size `patch_size` that are input into the grasp model. The larger the number of samples, the more is the inference time. Infering on 100 patches should take around 30 seconds on the NUC. After every grasp inference, a window showing the best found grasp is displayed until the user hits the space key. Running the script with `--no_visualize` disables this visualization.

```
source ~/pyenv_locobot_grasping/bin/activate
cd ~/low_cost_ws/src/pyrobot/examples/grasping
python locobot.py --n_grasps=5 --n_samples=100 --patch_size=100
```

# [Advanced] Sim2Real

## What is Sim2Real?

In this section we will train a simple RL policy which will learn inverse kinematics for the arm using RL(TD3). The input to the RL agent is state(joint angles of arm) & goal location(x,y,z) and the control action is the change in each joint angles to achieve the desired goal.

For TD3 implementation , we used the publicly available code.

Here is a demo video showing what one can accomplish through this tutorial.

## Installation

- Activate the virtual environment

Source virtual env

```
source ~/pyenv_pyrobot/bin/activate
```

- Install Dependencies

```
pip install gym
pip install pybullet
```

- Start by either *training the agent from scratch* **or** *downloading pretrained model*

Train the agent (this might take around an hour)

```
cd ~/low_cost_ws/src/pyrobot/examples/sim2real
python train.py --exp_name expID01 --valid_goals --save_models
```

Download pretrained model

```
cd ~/low_cost_ws/src/pyrobot/examples/sim2real
wget -O log.zip https://www.dropbox.com/s/8cttp55odd7e79o/log.zip?dl=0
unzip log.zip
rm log.zip
```

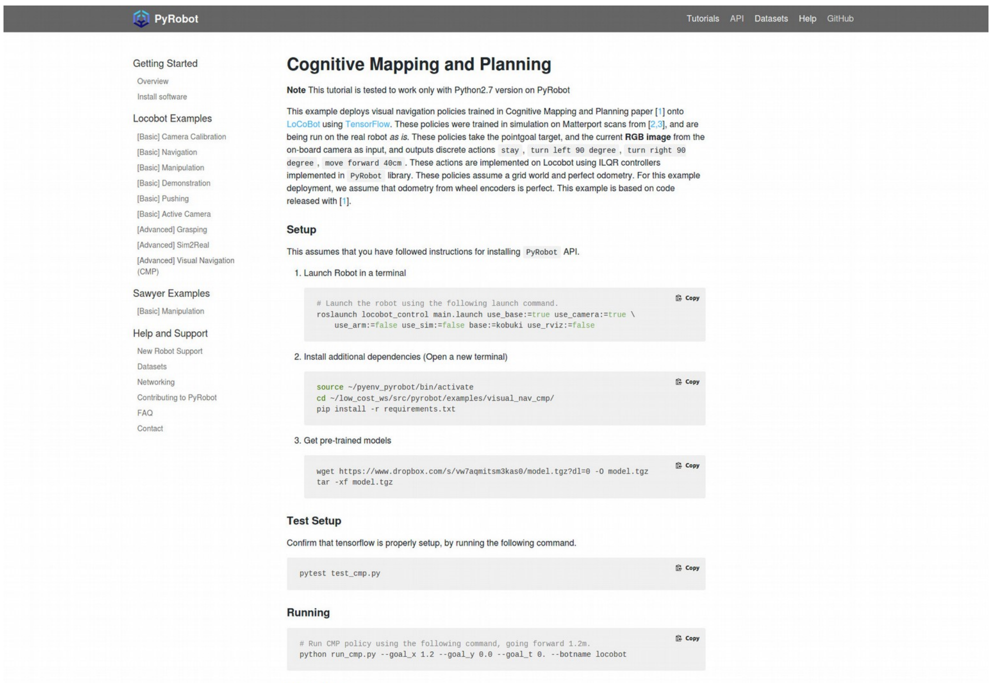
- Launch the real robot (open a new terminal)

```
roslaunch locobot_control main.launch use_arm:=true
```

- Test the above trained policy on the real robot

```
source ~/pyenv_pyrobot/bin/activate
cd ~/low_cost_ws/src/pyrobot/examples/sim2real
python test.py --use_real_robot --valid_goals --directory
./log/expID01/pytorch_models --file_name LocoBotEnv-v0_0
```

# [Advanced] Visual Navigation



## Cognitive Mapping and Planning

**Note** This tutorial is tested to work only with Python2.7 version on PyRobot

This example deploys visual navigation policies trained in Cognitive Mapping and Planning paper [1] onto LoCoBot using TensorFlow. These policies were trained in simulation on Matterport scans from [2,3], and are being run on the real robot *as is*. These policies take the pointgoal target, and the current **RGB image** from the on-board camera as input, and outputs discrete actions `stay`, `turn left 90 degree`, `turn right 90 degree`, `move forward 40cm`. These actions are implemented on Locobot using ILQR controllers implemented in `PyRobot` library. These policies assume a grid world and perfect odometry. For this example deployment, we assume that odometry from wheel encoders is perfect. This example is based on code released with [1].

### Setup

This assumes that you have followed instructions for installing `PyRobot` API.

1. Launch Robot in a terminal

```
# Launch the robot using the following launch command.
roslaunch locobot_control main.launch use_base:=true use_camera:=true \
    use_arm:=false use_sim:=false base:=kobuki use_rviz:=false
```

2. Install additional dependencies (Open a new terminal)

```
source ~/pyenv_pyrobot/bin/activate
cd ~/low_cost_ws/src/pyrobot/examples/visual_nav_cmp/
pip install -r requirements.txt
```

3. Get pre-trained models

```
wget https://www.dropbox.com/s/vw7aqmitsm3kas0/model.tgz?dl=0 -O model.tgz
tar -xf model.tgz
```

### Test Setup

Confirm that tensorflow is properly setup, by running the following command.

```
pytest test_cmp.py
```

### Running

```
# Run CMP policy using the following command, going forward 1.2m.
python run_cmp.py --goal_x 1.2 --goal_y 0.0 --goal_t 0. --botname locobot
```

1. Launch Robot in a terminal

```
# Launch the robot using the following launch command.

roslaunch locobot_control main.launch use_base:=true use_camera:=true \
    use_arm:=false use_sim:=false base:=kobuki use_rviz:=false
```

2. Install additional dependencies (Open a new terminal)

```
source ~/pyenv_pyrobot/bin/activate
cd ~/low_cost_ws/src/pyrobot/examples/visual_nav_cmp/
pip install -r requirements.txt
```

3. Get pre-trained models

```
wget https://www.dropbox.com/s/vw7aqmitsm3kas0/model.tgz?dl=0 -O model.tgz
tar -xf model.tgz
```

## Test Setup

Confirm that tensorflow is properly setup, by running the following command.

```
pytest test_cmp.py
```

## Running

```
# Run CMP policy using the following command, going forward 1.2m.
```

```
python run_cmp.py --goal_x 1.2 --goal_y 0.0 --goal_t 0. --botname locobot
```

## Demo Runs

1. Go forward 4m:

```
python run_cmp.py --goal_x 4.0 --goal_y 0.0 --goal_t 0. --compensate
```

2. Go forward 2m, left 2.4m:

```
python run_cmp.py --goal_x 2.0 --goal_y 2.4 --goal_t 0. --compensate
```

3. Go forward 3.2m:

```
python run_cmp.py --goal_x 3.2 --goal_y 0.0 --goal_t 0. --compensate
```