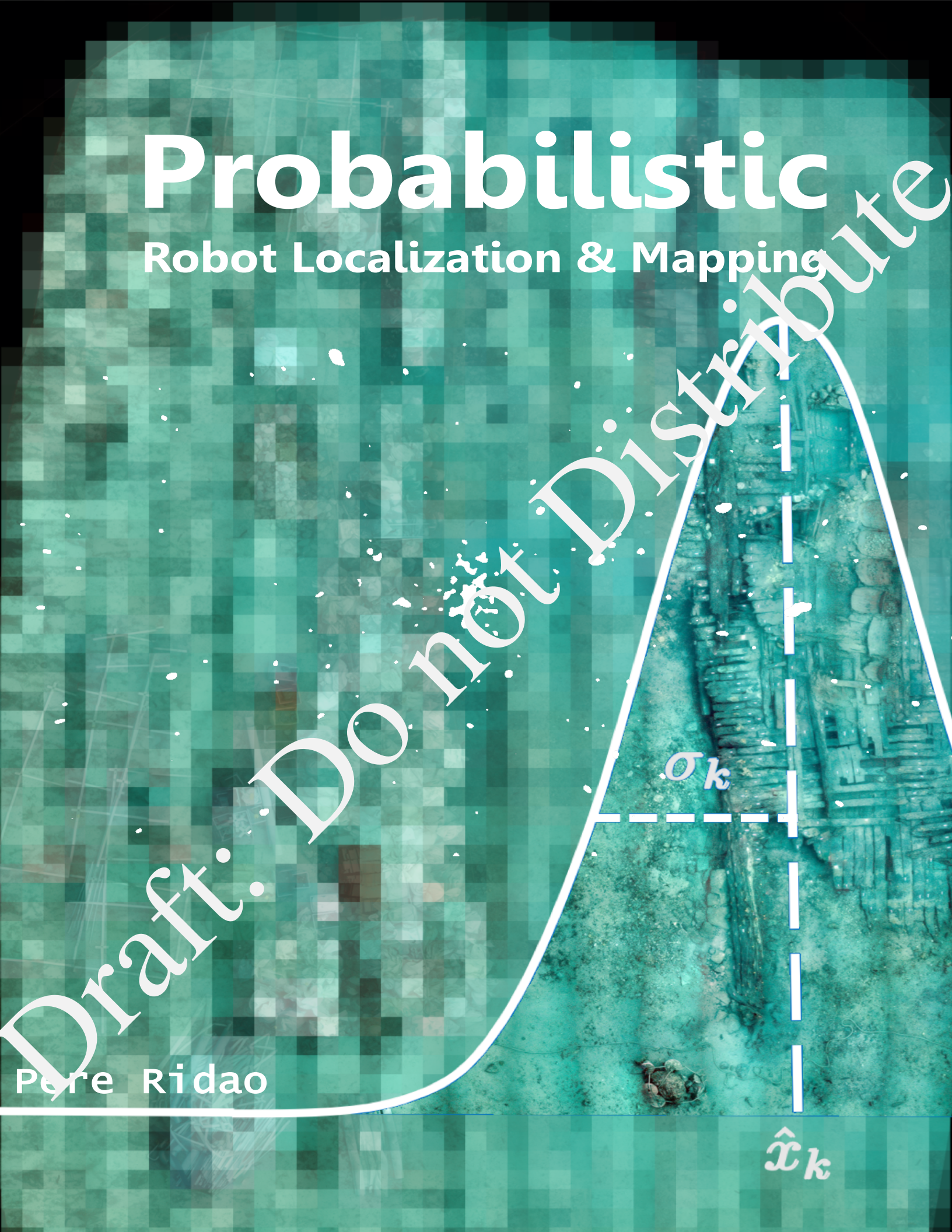
Probabilistic

Robot Localization & Mapping



Pere Ridao

Acronyms

DOF	Degree Of Freedom.
EKF	Extended Kalman Filter.
MBL	Map Based Localization.

MIR – EMJMD IN MARINE AND MARITIME INTELLIGENT ROBOTICS (MIR) ERASMUS MUNDUS JOINT MASTER'S DEGREE

Draft Lecture notes and proposed exercises of the Marine Localization & Mapping Course. DO NOT DISTRIBUTE

Release, March 2023

Contents

[2]This lab is labelled LAB4 in the repository, as it was 4^{th} lab during the course 2024/25. With the reconfiguration of the labs in course 2025 it became the lab 3.

1. Probabilistic Robotics Labs

1.1 PRPY Package Documentation

The *Probabilistic Robot Localization Package* (PRPY) is Python Library containing the main algorithms explained in the Probabilisitic Robot Localization Book used in the Probabilisitic Robotics and the Hands-on Localization Courses of the Intelligent Field Robotic Systems (IFRoS) European Erasmus Mundus Master. An HTML online documentation is available in the course moodle (PRPY web site) and a pdf version is also available in the moodel of the course (PRPY pdf documentation).

1.2 Organization of the Labs

The experimental part of the course is organized around 4 labs consisting on completing parts of the PRPY package. The list of the labs follows:

1. **Simulation**
2. **Dead Reckoning**
3. **Map Based Localization using Features**
4. **Feature Based EKF SLAM**

1.3 Lab 1: Simulation

The main objective of this lab is to simulate a differential drive mobile robot. This involves:

1. Installing the required software
2. Understanding the Pose compounding operation with the help of a python notebook
3. Studying the SimulatedRobot base class and completing the uncompleted code in the specialized DifferentialDriveSimulatedRobot class
4. Testing the simulation by programming the robot to execute both a circular and a figure-eight trajectory.

Let us start installing the software needed for the Lab.

1.3.1 Installing Jupyter Notebook

A Jupyter Notebook is an interactive computational environment which combines code execution, documentation, and visualization in a single web-based interface. Users can create and run code cells to perform calculations, manipulate data, and generate plots, while interspersing explanatory text, equations, and multimedia content in Markdown cells.

To install Jupyter Notebook using the Python package manager `pip`, you can follow these steps:

1. **Open a Terminal or Command Prompt**: Depending on your operating system (Windows, macOS, or Linux), open a terminal or command prompt with administrative privileges.
2. **Update `pip`**: It's a good practice to ensure that `pip` is up to date. You can do this by running the following command:
   ```
   $ pip install -upgrade pip
   ```
3. **Install Jupyter Notebook**: Once `pip` is up to date, you can install Jupyter Notebook using the following command:
   ```
   $ pip install jupyter
   ```
4. **Verify Installation**: After the installation is complete, you can verify that Jupyter Notebook is installed correctly by running:
   ```
   $ jupyter notebook
   ```
   This command should start the Jupyter Notebook server and open a new web browser window displaying the Jupyter Notebook dashboard.

1.3.2 Installing Python Libraries in your IDE

In this section, we'll guide you through the process of installing NumPy, Matplotlib, and the Robotics Toolbox for Python (roboticstoolbox-python) in your IDE (Visual Studio Code or PyCharm).

Open your IDE

Launch your IDE on your computer.

Create or Open a Python Project

Either create a new Python project or open an existing one in your IDE.

Open the Terminal

Inside the IDE, open the integrated terminal.

Install NumPy and Matplotlib

In the terminal, run the following commands to install NumPy and Matplotlib using `pip`:

```
$ pip install numpy
```

```
$ pip install matplotlib
```

Install Robotics Toolbox for Python

If you want to install the Robotics Toolbox for Python (roboticstoolbox-python), you can use `pip` as well:

```
$ pip install roboticstoolbox-python
```

Verify Installation

You can verify that the libraries are installed correctly by creating a Python script and importing them. For example:

```
import numpy as np
import matplotlib.pyplot as plt
import roboticstoolbox as rtb
```

If no errors occur, the libraries are successfully installed.

1.3.2.1 Cloning the PR_LAB1 Repository

In this section, we'll guide you through the process of cloning the "PR_LAB1" repository from GitHub onto your computer. This repository contains materials related to a laboratory exercise or project. Make sure you have Git installed on your computer before proceeding.
Follow these steps to clone the repository:

1. **Open a Terminal or Command Prompt**: Depending on your operating system (Windows, macOS, or Linux), open a terminal or command prompt.
2. **Navigate to the Desired Directory**: Use the 'cd' command to navigate to the directory where you want to store the repository. For example:
   ```
   $ cd Documents
   ```
3. **Clone the Repository**: Run the following command to clone the "PR_LAB1" repository from GitHub. Replace '<your-username>' with your GitHub username:
   ```
   $ git clone https://github.com/IFROS-MIRS/PR_LAB1.git
   ```
   If you have GitHub authentication set up, you won't need to provide credentials. If not, GitHub may prompt you to enter your username and password.
4. **Verify Cloning**: Once the cloning process is complete, navigate to the cloned repository's directory using the 'cd' command:
   ```
   $ cd PR_LAB1
   ```
   You should now be inside the cloned repository. You can check the contents to verify that everything has been cloned correctly.

That's it! You've successfully cloned the "PR_LAB1" repository onto your computer. You can now work on the contents of the repository or use them for your laboratory exercise or project.

1.3.3 Pose Compounding Notebook

The **goal** of this exercise explains how to define a robot pose in 3 Degree Of Freedom (DOF) as well as how to compose different poses using the direct and the inverse compounding operations.

1.3.3.1 Step-by-Step Guide

In this section, we'll walk you through the process of completing and executing the "Lab0_compounding.ipynb" notebook using Jupyter Notebook. Follow these steps to work with the notebook:

1. **Start Jupyter Notebook**: Open a terminal or command prompt on your computer and navigate to the directory where the "Lab0_compounding.ipynb" notebook is located. Launch Jupyter Notebook by running the following command:
   ```
   $ jupyter notebook
   ```
   This will open the Jupyter Notebook dashboard in your web browser.
2. **Access the Notebook**: In the Jupyter Notebook dashboard, locate and click on the "Lab0_compounding.ipynb" notebook. This will open the notebook in a new tab.
3. **Navigate the Notebook**: The notebook consists of a series of cells, including Markdown cells for instructions and code cells for calculations. Read the instructions in each cell carefully.
4. **Execute Code Cells**: For code cells, you'll need to run them to perform calculations. To execute a code cell, select it and click the "Run" button in the Jupyter Notebook toolbar or use the keyboard shortcut Shift + Enter. The code will run, and any output or results will be displayed below the cell.
5. **Submit Your Work**: If this notebook is part of an assignment, follow your instructor's guidelines for submission. You may need to export the notebook as a PDF and submit it through the designated platform.

By following these steps, you'll be able to complete and execute "Lab0_compounding.ipynb" in Jupyter Notebook.

1.3.4 Simulation of a Differential Drive Mobile Robot

The **goal** of this part of the labis to simulate the motion of a differention drive mobile robot. The `SimulatedRobot` and `DifferentialDriveSimulatedRobot` classes are integral components of a robot simulation framework designed to model and study the localization and mapping algorithms of robotic systems within a virtual environment. The **goal** of this lab is to simulate a differential drive mobile robot.

1.3.4.1 SimulatedRobot Class

The `SimulatedRobot` class serves as the foundational element of the simulation framework. It defines a generic robot model, providing attributes and methods for motion modeling, sensor simulation, parameter initialization, and visualization. This class is designed to be extensible, allowing developers to create specialized robot simulation classes by inheriting from it.

1.3.4.2 DifferentialDriveSimulatedRobot Class

The `DifferentialDriveSimulatedRobot` class is a specialized extension of the `SimulatedRobot` class. It focuses on modeling the behavior of robots with a differential drive configuration, typically consisting of two independently controlled wheels. This class overrides certain methods and attributes of the `SimulatedRobot` class to tailor them to the specific characteristics of differential drive robots. It provides motion modeling specific to differential drive robots, simulates sensors such as wheel encoders and compasses, and customizes parameterization for accurate simulation.

1.3.5 Implementing the robot simulation

In this part of the lab we will complete the simulation code for a 3 DOF Differential Drive Mobile Robot.

1. **Understanding the `SimulatedRobot` base class**:
   (a) Read its documentation in the `SimulatedRobot` page (Click on the name).
   (b) Check the code of the *RobotSimulation.py* file.

2. **`DifferentialDriveSimulatedRobot` class**:
   (a) Read its documentation in the `DifferentialDriveSimulatedRobot` page (click on the name).
   (b) Check the uncompleted code provided for this class in the *DifferentialDriveSimulatedRobot.py* file.
   (c) Complete the code of the class by implementing the methods labeled as *"# Todo: To be completed by the student"*
3. **Test the Robot simulation**:
   (a) Program the robot simulation to perform a circular shaped trajectory.
   (b) Program the robot simulation to perform a trajectory shaped as an "8".

1.4 Lab 2: Dead Reckoning

The main **goal** of Lab 2 is to **implement and test the Dead Reckoning Localization algorithm** for a simulated 3 DOF Differential Drive Mobile Robot.
This involves:

1. Understanding the base `Localization` class.
2. Completing the core code for the **`DR_3DOFDifferentialDrive`** class by implementing the methods necessary to calculate the robot's pose solely based on its motion model (wheel encoder readings).
3. Testing the resulting Dead Reckoning algorithm by evaluating its localization results while the robot follows a **circular trajectory**.

1.4.1 Understanding the involved classes

Two solve this lab you need to bve familiarized with the `Localization` class and you will need to complete the `DR_3DOFDifferentialDrive` class.

1.4.1.1 Localization Class

The `Localization` class serves as the base class for implementing and executing localization algorithms within a simulated robotic environment. This class provides the fundamental structure and methods necessary for simulating and solving the robot localization problem.
A detailed description of the class can be found in the `Localization` page.

1.4.1.2 DifferentialDriveSimulatedRobot Class

The `DifferentialDriveSimulatedRobot` class represents a simulated robot with a differential drive motion model. It serves as an extension of the `SimulatedRobot` class and overrides several of its methods to define the specific motion model and behaviors of a differential drive robot.
A detailed description of the class can be found in the `DR_3DOFDifferentialDrive` page.

1.4.2 Implementing the Dead Reckoning Equations

In this part of the lab we will solve the Dead Reckoning Localization of a 3 DOF Differential Drive Mobile Robot.

1. Understanding the `Localization` base class:
   (a) Read its documentation in the `Localization` page (click on the name).
   (b) Check the provided code of the *Localization.py* file.
2. Completing the `DR_3DOFDifferentialDrive` class:
   (a) Read its documentation in the `DR_3DOFDifferentialDrive` page (click on the name).
   (b) Check the uncompleted code provided for this class in the *DR_3DOFDifferentialDrive.py* file.
   (c) Complete the code of the class by implementing the methods labeled as *"# Todo: To be completed by the student"*
3. Testing the Robot Dead Reckoning Localization:
   (a) Circular Trajectory: Check the localization results of the implemented Dead Reckoning algorithm.

1.5 Lab 3: Map Based EKF Localization

This exercise tackles the problem of the localization of the Differential Drive Mobile Robot using an a priory known map of point features. The features are represented in the map in Cartesian coordinates but they are observed by the robot in Polar Coordinates.

1.5.1 Cloning the PR_LAB4[1] Repository

In this section, we'll guide you through the process of cloning the "PR_LAB4" repository from GitHub onto your computer. This repository contains materials related to a laboratory exercise or project. Make sure you have Git installed on your computer before proceeding.
Follow these steps to clone the repository:

1. **Open a Terminal or Command Prompt**: Depending on your operating system (Windows, macOS, or Linux), open a terminal or command prompt.
2. **Navigate to the Desired Directory**: Use the 'cd' command to navigate to the directory where you want to store the repository. For example:
   ```
   $ cd Documents
   ```
3. **Clone the Repository**: Run the following command to clone the "PR_LAB1" repository from GitHub. Replace '<your-username>' with your GitHub username:
   ```
   $ git clone https://github.com/IFROS-MIRS/PR_LAB4.git
   ```
   If you have GitHub authentication set up, you won't need to provide credentials. If not, GitHub may prompt you to enter your username and password.
4. **Verify Cloning**: Once the cloning process is complete, navigate to the cloned repository's directory using the 'cd' command:
   ```
   $ cd PR_LAB4
   ```
   You should now be inside the cloned repository. You can check the contents to verify that everything has been cloned correctly.

That's it! You've successfully cloned the "PR_LAB4" repository onto your computer. You can now work on the contents of the repository or use them for your laboratory exercise or project.

1.5.2 Updating with your solution of LAB1

Once you have cloned the repository, you need to update the python files that you already programmed during lab1. This include the files:

1. *DifferentialDriveSimulatedRobot.py*
2. *DR_3DOFDifferentialDrive.py*
3. The *Pose3D.py* has been replaced by a new file named *Pose3D.py*. It defines a base class named *Pose* from where the class *Pose3D* inherits. You need to update the methods of the *Pose3D* class, within the *Pose3D.py* file , with the contents you already programmed during LAB1. You will need to change the *from Pose3D import ** of *DR_3DOFDifferentialDrive.py* file by an *from Pose import **.

Next you have to update certain variables within your files:

1. self.yaw_reading_frequency=1 (DifferentialDriveSimulatedRobot.py)
2. self.visualizationInterval=1 (SimulatedRobot.py)
3. self.vehicleIcon=VehicleIcon('DifferentialDrive.png', scale=1, rotation=90) (SimulatedRobot.py)

Now we are ready to solve the first part of the lab.

[1]This lab is labelled LAB4 in the repository, as it was 4^{th} lab during the course 2024/25. With the reconfiguration of the labs in course 2025 it became the lab 3.

1.5.3 Part I: Localization using Dead Reckoning and a Compass

The goal of part I is to implement an Extended Kalman Filter (EKF) Localization method using the robot odometry displacement as the motion model and making updates with the compass readings. You need to follow the next steps:

1. Read the documentation of the `GFLocalization` class, check its code and complete the methods labelled *# To Do: To be completed by the Student*.
2. Read the documentation of the `EKF` class, check its code and complete the methods labelled *# To Do: To be completed by the Student*.
3. Read the documentation of the `EKF_3DOFDifferentialDriveInputDisplacement` class, and complete its methods labelled *# To Do: to be completed by the student*.
4. To test the EKF filter, you can program different values for the attribute *yaw_reading_frequency* of the `DifferentialDriveSimulatedRobot` class:
   (a) Using a value greater than the number of *ksteps* (number of simulation iterations), will avoid the compass updates.
   (b) Using a value equal to 1, will use the compass updates at each simulation step.
   (c) Using a value equal to 100, will use a compass update every 10 seconds.

   Comment about the results obtained.
5. Optional: Constant Velocity Motion Model with encoder and compass updates
   (a) Read the documentation of the `EKF_3DOFDifferentialDriveCtVelocity` class, in the file *EKF_3DOFDifferentialDriveCtVelocity.py* and complete its methods labelled *# To Do: To be completed by the Student*.
   (b) To **test the EKF filter**, in the same conditions of the previous one.

1.5.4 Part II: Including Map Feature Suport

Follow the next steps:

1. `Feature` class: Base class used for Feature definition. All features must inherit from this one. It defines the interface of a Feature.
   (a) Read the documentation of the `Feature` class.
   (b) Check its already provided code in the file *Feature.py*.
2. `CartessianFeature` class: Implementation of a Cartesian Feature. Inherits from the `Feature` class and provides the implementation of the pose-feature compounding operation $\boxplus$ and its related jacobians $J_{1\boxplus}$ and $J_{2\boxplus}$. It also includes the methods *ToCartesian()* and *J_2c()* which are used to be able to plot the features.
   (a) Read the documentation of the `CartessianFeature` class.
   (b) Complete the missing code labelled *# To Do: To be completed by the Student*.
3. `MapFeature` class: This class provides the functionality required to use features for map based localization.
   (a) Read the documentation of the `MapFeature` class.
   (b) Check its code in the file *MapFeature.py* and complete the requested methods.
4. `Cartesian2DMapFeature` class: This class inherits from the `MapFeature` class and implements a 2D Cartesian feature model for the Map Based Localization (MBL) problem. The Cartesian coordinates are used for both, observing as well as storing it within the map.
   (a) Read the documentation of the `Cartesian2DMapFeature` class.
   (b) Check its code in the file *MapFeature.py* and complete the requested methods.

5. Optional:
   (a) 2D Feature Stored in Cartesian space but observed in Polar space.
      i. Read the documentation of the `Cartesian2DStoredPolarObservedMapFeature` *class*.
      ii. Check its code in the file *MapFeature.py* and complete the requested methods.
   (b) Create a `PolarFeature` class using features observed and stored in Polar Coordinates.

1.5.5 Part III: Map Based EKF Localization

Follow the next steps:

1. `FEKFMBL` class: This class extends the `GFLocalization` (providing the basic functionality of a localization algorithm) and `MapFeature` (providing the basic functionality required to use features) by adding functionality to use a map based on features.
   (a) Read the documentation of the `FEKFMBL` class.
   (b) Check its code in the file *FEKFMBL.py* and complete the requested methods.
2. **The MBL_3DOFDDInputDisplacementMM_2DCartesianFeatureOM class**: This class implements a Feature EKF Map based Localization of a 3 DOF Differential Drive Mobile Robot using a 2D Cartesian feature map and an input displacement motion model .
   (a) Read the documentation of the **MBL_3DOFDDInputDisplacementMM_2DCartesianFeatureOM class**.
   (b) Check its code in the file *MBL_3DOFDDInputDisplacementMM_2DCartesianFeatureOM.py* and complete the requested methods.
3. **Test the Localization algorithm**, in the following conditions:
   (a) Without compass updates neither feature updates. To achieve this you can play with attribute s *yaw_reading_frequency* and *xy_feature_reading_frequency* .
   (b) Without compass updates but including feature updates every 50 simulation steps. To achieve this you can play with attributes *yaw_reading_frequency* and *xy_feature_reading_frequency*.

   Comment of the result obtained.
4. **Optional**
   (a) **2D Features Stored in Cartesian Space but Observed in Polar Space**
      i. Read the documentation of the **MBL_3DOFDDInputDisplacementMM_2DCartesianFeaturePolarOb class**.
      ii. Check its code in the file *MBL_3DOFDDInputDisplacementMM_2DCartesianFeaturePolarObservedOM.p* and complete the requested methods.
      iii. **Test** the method in the same conditions than the previous algorithms.
   (b) **Polar Features**: Implement the code required to run the file *MBL_3DOFDDInputDisplacementMM_2DPolarFe* which implements a Map Based EKF Localization of a Differential Drive Mobile robot using an input displacement motion model and uses Polar features observed and stored in Polar coordinates. **Test** the method in the same conditions than the previous algorithms.
   (c) **Constant Velocity Motion Model**
      i. Read the documentation of the **MBL_3DOFDDCtVelocityMM_2DCartesianFeatureOM class**.
      ii. Check its code in the file *MBL_3DOFDDCtVelocityMM_2DCartesianFeatureOM.py* and complete the requested methods.
      iii. **Test** the method in the same conditions than the previous algorithms.

1.6 Feature EKF SLAM Localization

This exercise tackles the problem of the localization of the Differential Drive Mobile Robot without any previous knowledge of the map arrangement. The features are observed by the robot in Cartesian Coordinates and stored in the state vector also as Cartesian Coordinates.
The major challenge of this lab is dealing with a state vector that contains the mapped features and grows in size when new, unmapped features are detected. As the map is part of the state vector, and therefore is also optimized, the **prediction** function must be updated, as well as the **observation model and its jacobians**. Moreover, it emerges the need to handle the addition of new features into the state. In this lab, these issues are tackled in isolation.

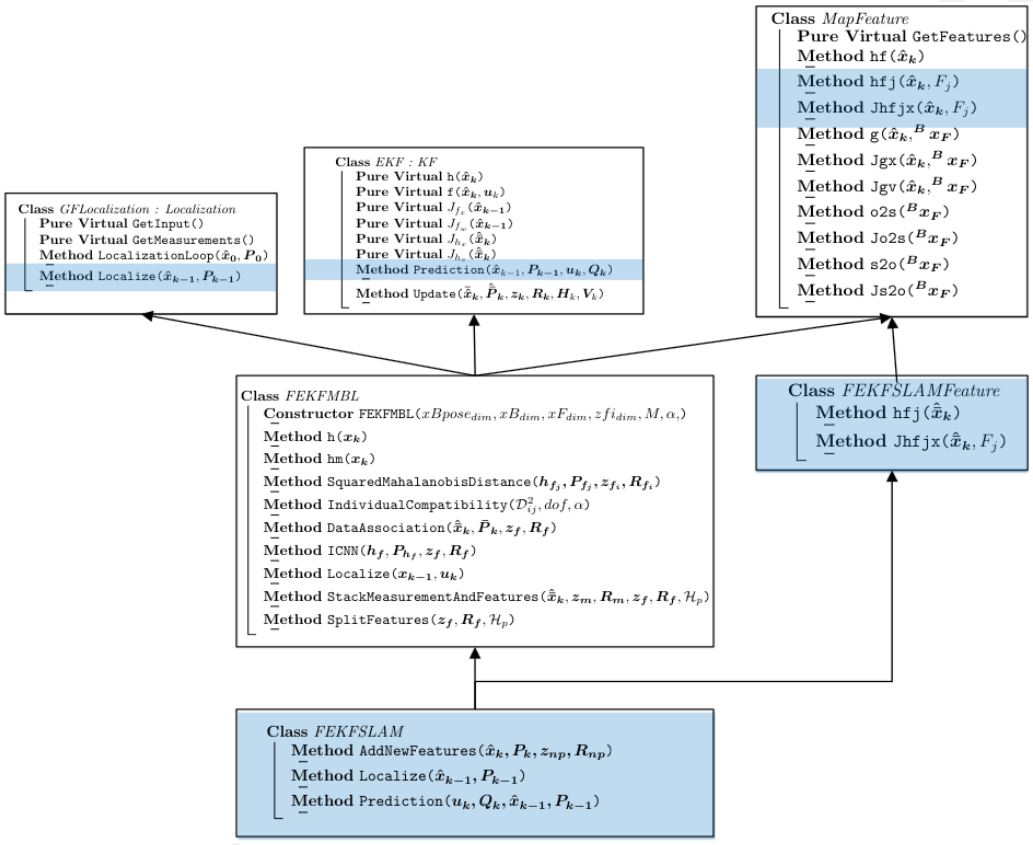


Figure 1.1: SLAM class hierarchy

1.6.1 Installation

This lab is not set up in a git repository, as only a few modifications of the **Map Based EKF Localization** lab are required. Download the following attached files into the root folder of your Map-Based EKF lab:

- FEKFSLAM.py
- FEKFSLAMFeature.py
- FEKFSLAM_3DOFDD_InputVelocityMM_2DCartesianFeatureOM.py

Figure 1.1 depicts the class hierarchy and the functions that need to be overwritten to upgrade a Map-based localization EKF to a full SLAM solution.

Before starting the lab, take your time to go through the files and understand the role of each class.

1.6.2 Part I: Localization without feature measurements

The goal of part I is to implement the **Prediction** method in isolation. Therefore, no feature updates nor state augmentation will be used.

1. Read the documentation of the **FEKFSLAM class**.
2. Force the initial vector state (and its associated covariance matrix) to include the position of the features in the map.
3. Implement the upgraded **Prediction** function.
4. Implement the **Localize** function to fulfill the needs of this part. Compass measurements are not necessary but can be included.
5. Test the Localization by running the **FEKFSLAM_3DOFDD_InputVelocityMM-_2DCartesianFeatureOM.py** script. Try it with different state vector sizes to double-check that it works.

1.6.3 Part II: SLAM with a priori-known features

The goal of part II is to perform the **Update** part of the SLAM algorithm without dealing with the observation of unmapped features.

1. Check that the manually included features in Part I are correct. Set their uncertainty to be small. This case resembles Map based localization.
2. Read the documentation of the **FEKFSLAMFeature class**.
3. Implement the **hfj** and **Jhfx** methods of the **FEKFSLAMFeature** class.
4. Upgrade the **Localize** method to perform updates.
5. Test the localization.
6. Increase the initial uncertainty of the features in the vector state. Test the localization.
7. Initialize the map features with some error (within the initial uncertainty). Test the localization.
8. Comment the results.

1.6.4 Part III: Full SLAM

Finally, the goal of part III is to implement the full SLAM algorithm. At this point, the last piece is to augment the state vector when new, unmapped features are detected.

1. Undo the manually included features from the state vector and its associated covariance matrix. The initial state vector must contain only the robot state.
2. Read the documentation of the **FEKSLAM class**.
3. Implement the inverse observation equation and its jacobians (**g**, **Jgx**, and **Jgv**) in the **MapFeature** class, in case you did not implement it in the previous lab.
4. Implement the **AddNewFeatures** function.
5. Upgrade the **Localize** method to add new features, but without performing updates.
6. Test the localization without updates. Comment the results.
7.
8. Upgrade the **Localize** method to perform updates.
9. Test the localization. Comment the results.
10. Optional:
    (a) 2D Features Stored in Cartesian Space but Observed in Polar Space.
    (b) Constant Velocity Motion Model.

Bibliography

[1] Pere Ridao and Roger Pi. *prpy: Probabilistic Robot Localization Python Library*. October 2023. Available in pdf in the ./docs/latex folder of the lab repository and in html in the `file:./docs/html/index.html` file.