prpy: Probabilistic Robot Localization Python Library

Release 0.1

Pere Ridao

Nov 14, 2023

CONTENTS

Probabilistic Robot Localization 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.

Note: This documentation is still under construction.

CHAPTER

ONE

API:

1.1 Pose Representation

1.1.1 Pose 3DOF

class `Pose3D.`**`Pose3D`**(*input_array*)

Bases: `numpy.ndarray`

Definition of a robot pose in 3 DOF (x, y, yaw). The class inherits from a ndarray. This class extends the ndarray with the $oplus$ and $ominus$ operators and the corresponding Jacobians.

`oplus`(*BxC*)

Given a Pose3D object *AxB* (the self object) and a Pose3D object *BxC*, it returns the Pose3D object *AxC*.

$$
\begin{aligned}
{}^{\mathbf{A}}\mathbf{x_B} &= \begin{bmatrix} {}^{A}x_B & {}^{A}y_B & {}^{A}\psi_B \end{bmatrix}^T \\
{}^{\mathbf{B}}\mathbf{x_C} &= \begin{bmatrix} {}^{B}x_C & {}^{B}y_C & {}^{B}\psi_C \end{bmatrix}^T
\end{aligned}
$$

The operation is defined as:

$$
{}^{\mathbf{A}}\mathbf{x_C} = {}^{\mathbf{A}}\mathbf{x_B} \oplus {}^{\mathbf{B}}\mathbf{x_C} = \begin{bmatrix} {}^{A}x_B +^{B} x_C \cos({}^{A}\psi_B) -^{B} y_C \sin({}^{A}\psi_B) \\ {}^{A}y_B +^{B} x_C \sin({}^{A}\psi_B) +^{B} y_C \cos({}^{A}\psi_B) \\ {}^{A}\psi_B +^{B} \psi_C \end{bmatrix} \tag{1.1}
$$

Parameters **`BxC`** – C-Frame pose expressed in B-Frame coordinates

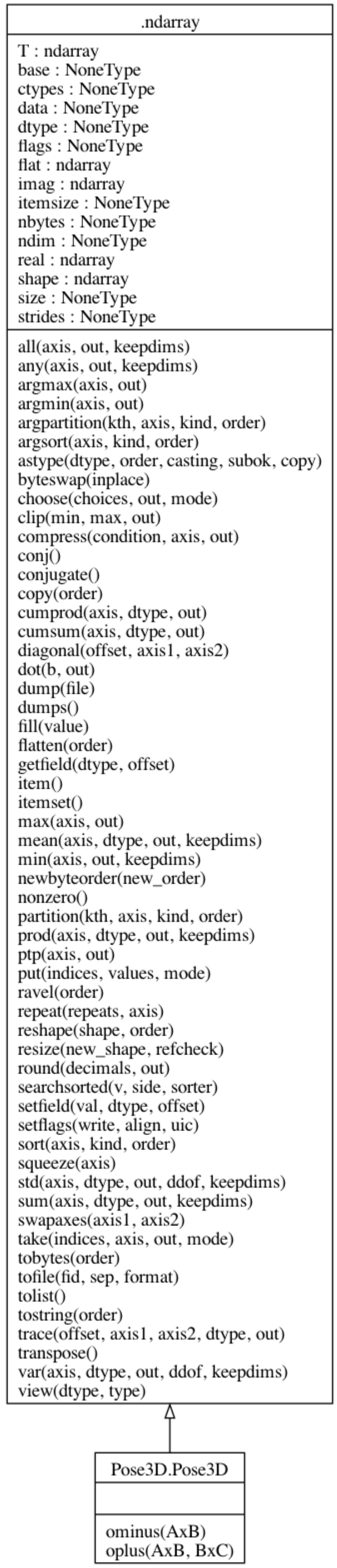
Returns C-Frame pose expressed in A-Frame coordinates

`ominus`()

Inverse pose compounding of the *AxB* pose (the self objetc):

$$
{}^{B}x_A = \ominus {}^{A}x_B = \begin{bmatrix} -{}^{A}x_B \cos({}^{A}\psi_B) -^{A} y_B \sin({}^{A}\psi_B) \\ {}^{A}x_B \sin({}^{A}\psi_B) -^{A} y_B \cos({}^{A}\psi_B) \\ -{}^{A}\psi_B \end{bmatrix} \tag{1.2}
$$

Returns A-Frame pose expressed in B-Frame coordinates (eq. (1.2))


.ndarray
T : ndarray
base : NoneType
ctypes : NoneType
data : NoneType
dtype : NoneType
flags : NoneType
flat : ndarray
imag : ndarray
itemsize : NoneType
nbytes : NoneType
ndim : NoneType
real : ndarray
shape : ndarray
size : NoneType
strides : NoneType
all(axis, out, keepdims)
any(axis, out, keepdims)
argmax(axis, out)
argmin(axis, out)
argpartition(kth, axis, kind, order)
argsort(axis, kind, order)
astype(dtype, order, casting, subok, copy)
byteswap(inplace)
choose(choices, out, mode)
clip(min, max, out)
compress(condition, axis, out)
conj()
conjugate()
copy(order)
cumprod(axis, dtype, out)
cumsum(axis, dtype, out)
diagonal(offset, axis1, axis2)
dot(b, out)
dump(file)
dumps()
fill(value)
flatten(order)
getfield(dtype, offset)
item()
itemset()
max(axis, out)
mean(axis, dtype, out, keepdims)
min(axis, out, keepdims)
newbyteorder(new_order)
nonzero()
partition(kth, axis, kind, order)
prod(axis, dtype, out, keepdims)
ptp(axis, out)
put(indices, values, mode)
ravel(order)
repeat(repeats, axis)
reshape(shape, order)
resize(new_shape, refcheck)
round(decimals, out)
searchsorted(v, side, sorter)
setfield(val, dtype, offset)
setflags(write, align, uic)
sort(axis, kind, order)
squeeze(axis)
std(axis, dtype, out, ddof, keepdims)
sum(axis, dtype, out, keepdims)
swapaxes(axis1, axis2)
take(indices, axis, out, mode)
tobytes(order)
tofile(fid, sep, format)
tolist()
tostring(order)
trace(offset, axis1, axis2, dtype, out)
transpose()
var(axis, dtype, out, ddof, keepdims)
view(dtype, type)
Pose3D.Pose3D
ominus(AxB)
oplus(AxB, BxC)

1.2 Robot Simulation

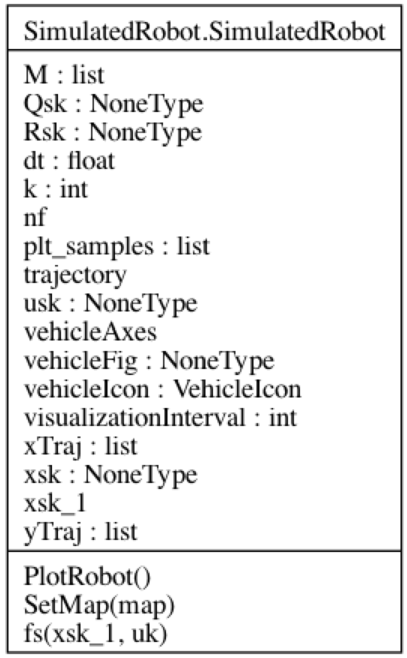


Fig. 1: SimulatedRobot Class Diagram.

`class SimulatedRobot.`**`SimulatedRobot`**(*xs0*, *map=[]*, **args*)

Bases: `object`

This is the base class to simulate a robot. There are two operative frames: the world N-Frame (North East Down oriented) and the robot body frame body B-Frame. Each robot has a motion model and a measurement model. The motion model is used to simulate the robot motion and the measurement model is used to simulate the robot measurements.

All Robot simulation classes must derive from this class .

`dt = 0.1`

class attribute containing sample time of the simulation

`__init__`(*xs0*, *map=[]*, **args*)

Parameters

- **`xs0`** – initial simulated robot state x_{s_0} used to initialize the the motion model
- **`map`** – feature map of the environment $M = [^N x_{F_1}^T, ..., ^N x_{F_{nf}}^T]^T$

Constructor. First, it initializes the robot simulation defining the following attributes:

- **k** : time step
- **Qsk** : **To be defined in the derived classes**. Object attribute containing Covariance of the simulation motion model noise
- **usk** : **To be defined in the derived classes**. Object attribute contining the simulated input to the motion model

- **xsk** : **To be defined in the derived classes**. Object attribute contining the current simulated robot state
- **zsk** : **To be defined in the derived classes**. Object attribute contining the current simulated robot measurement
- **Rsk** : **To be defined in the derived classes**. Object attribute contining the observation noise covariance matrix
- **xsk** : current pose is the initial state
- **xsk_1** : previouse state is the initial robot state
- **M** : position of the features in the N-Frame
- **nf** : number of features

Then, the robot animation is initialized defining the following attributes:

- **vehicleIcon** : Path file of the image of the robot to be used in the animation
- **vehicleFig** : Figure of the robot to be used in the animation
- **vehicleAxes** : Axes of the robot to be used in the animation
- **xTraj** : list containing the x coordinates of the robot trajectory
- **yTraj** : list containing the y coordinates of the robot trajectory
- **visualizationInterval** : time-steps interval between two consecutive frames of the animation

`PlotRobot()`

Updates the plot of the robot at the current pose

`fs(`*xsk_1*, *uk*`)`

Motion model used to simulate the robot motion. Computes the current robot state x_k given the previous robot state x_{k-1} and the input u_k. It also updates the object attributes xsk, xsk_1 and usk to be made them available for plotting purposes. *To be overriden in child class*.

Parameters

- **`xsk_1`** – previous robot state x_{k-1}
- **`usk`** – model input u_{s_k}

Returns current robot state x_k

`SetMap(`*map*`)`

Initializes the map of the environment.

`_PlotSample(`*x*, *P*, *n*`)`

Plots n samples of a multivariate gaussian distribution. This function is used only for testing, to plot the uncertainty through samples. :param x: mean pose of the distribution :param P: covariance of the distribution :param n: number of samples to plot

1.2.1 3 DOF Diferential Drive Robot Simulation

class DifferentialDriveSimulatedRobot.**DifferentialDriveSimulatedRobot**(*xs0*, *map=[]*, **args*)

Bases: `SimulatedRobot.SimulatedRobot`

This class implements a simulated differential drive robot. It inherits from the `SimulatedRobot` class and overrides some of its methods to define the differential drive robot motion model.

__init__(*xs0*, *map=[]*, **args*)

Parameters

- **xs0** – initial simulated robot state $\mathbf{x_{s_0}} = [{}^N x_{s_0}\ {}^N y_{s_0}\ {}^N \psi_{s_0}\ ]^T$ used to initialize the motion model
- **map** – feature map of the environment $M = [{}^N x_{F_1}, ..., {}^N x_{F_{nf}}]$

Initializes the simulated differential drive robot. Overrides some of the object attributes of the parent class `SimulatedRobot` to define the differential drive robot motion model:

- **Qsk** : Object attribute containing Covariance of the simulation motion model noise.

$$Q_k = \begin{bmatrix} \sigma_{\dot{u}}^2 & 0 & 0 \\ 0 & \sigma_{\dot{v}}^2 & 0 \\ 0 & 0 & \sigma_{\dot{r}}^2 \end{bmatrix} \tag{1.3}$$

- **usk** : Object attribute containing the simulated input to the motion model containing the forward velocity u_k and the angular velocity r_k

$$\mathbf{u_k} = \begin{bmatrix} u_k & r_k \end{bmatrix}^{\mathbf{T}} \tag{1.4}$$

- **xsk** : Object attribute containing the current simulated robot state

$$x_k = \begin{bmatrix} {}^N x_k & {}^N y_k & {}^N \theta_k & {}^B u_k & {}^B v_k & {}^B r_k \end{bmatrix}^T \tag{1.5}$$

where ${}^N x_k$, ${}^N y_k$ and ${}^N \theta_k$ are the robot position and orientation in the world N-Frame, and ${}^B u_k$, ${}^B v_k$ and ${}^B r_k$ are the robot linear and angular velocities in the robot B-Frame.

- **zsk** : Object attribute containing $z_{s_k} = [n_L\ n_R]^T$ observation vector containing number of pulses read from the left and right wheel encoders.
- **Rsk** : Object attribute containing $R_{s_k} = diag(\sigma_L^2, \sigma_R^2)$ covariance matrix of the noise of the read pulses`.
- **wheelBase** : Object attribute containing the distance between the wheels of the robot ($w = 0.5$ m)
- **wheelRadius** : Object attribute containing the radius of the wheels of the robot ($R = 0.1$ m)
- **pulses_x_wheelTurn** : Object attribute containing the number of pulses per wheel turn ($pulseXwheelTurn = 1024$ pulses)
- **Polar2D_max_range** : Object attribute containing the maximum Polar2D range ($Polar2D_max_range = 50$ m) at which the robot can detect features.
- **Polar2D_feature_reading_frequency** : Object attribute containing the frequency of Polar2D feature readings (50 tics -sample times-)
- **Rfp** : Object attribute containing the covariance of the simulated Polar2D feature noise ($R_{fp} = diag(\sigma_\rho^2, \sigma_\phi^2)$)

Check the parent class `prpy.SimulatedRobot` to know the rest of the object attributes.

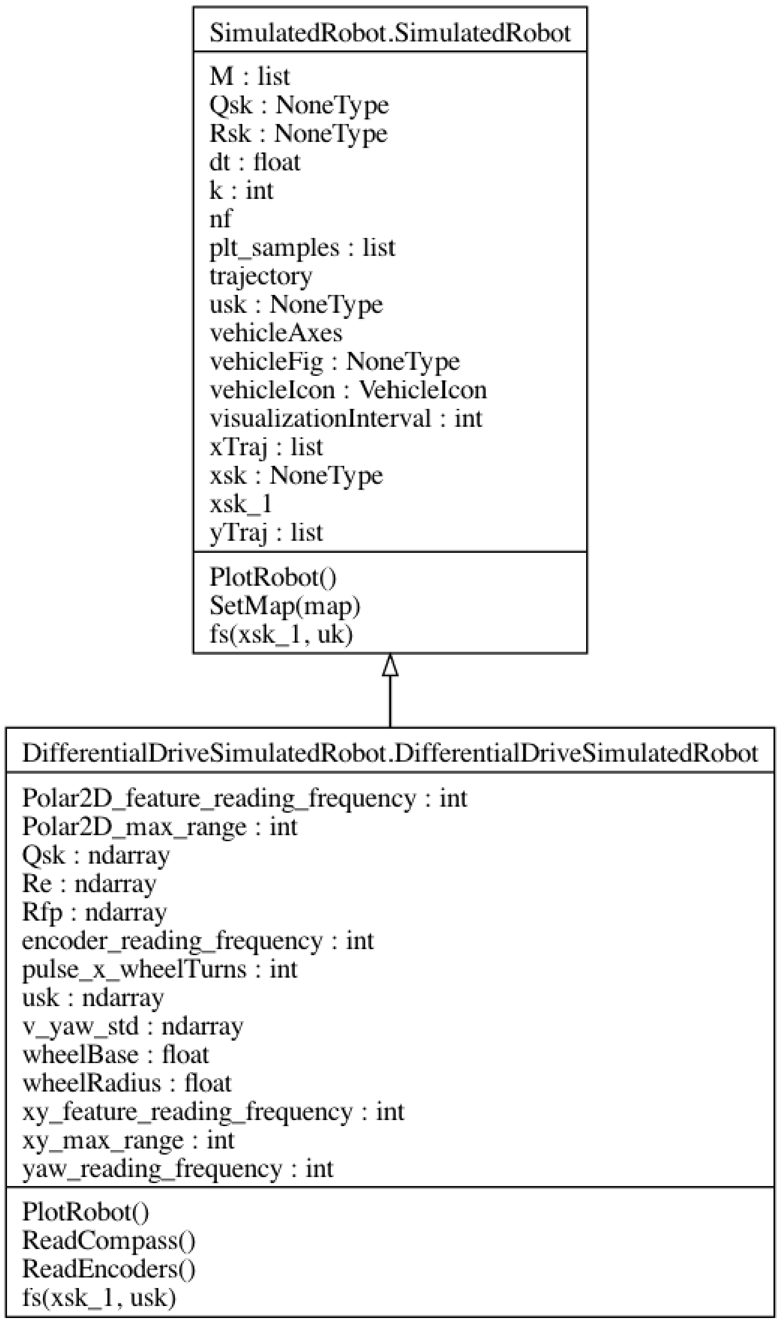


Fig. 2: DifferentialDriveSimulatedRobot Class Diagram.

fs(*xsk_1*, *usk*)

Motion model used to simulate the robot motion. Computes the current robot state x_k given the previous robot state x_{k-1} and the input u_k:

$$\begin{aligned}
\eta_{s_{k-1}} &= \begin{bmatrix} x_{s_{k-1}} & y_{s_{k-1}} & \theta_{s_{k-1}} \end{bmatrix}^T \\
\nu_{s_{k-1}} &= \begin{bmatrix} u_{s_{k-1}} & v_{s_{k-1}} & r_{s_{k-1}} \end{bmatrix}^T \\
x_{s_{k-1}} &= \begin{bmatrix} \eta^T_{s_{k-1}} & \nu^T_{s_{k-1}} \end{bmatrix}^T \\
u_{s_k} &= \nu_d = \begin{bmatrix} u_d & r_d \end{bmatrix}^T \\
w_{s_k} &= \dot{\nu}_{s_k} \\
x_{s_k} &= f_s(x_{s_{k-1}}, u_{s_k}, w_{s_k}) \\
&= \begin{bmatrix} \eta_{s_{k-1}} \oplus (\nu_{s_{k-1}} \Delta t + \frac{1}{2} w_{s_k}) \\ \nu_{s_{k-1}} + K(\nu_d - \nu_{s_{k-1}}) + w_{s_k} \Delta t \end{bmatrix} \quad ; \quad K = diag(k_1, k_2, k_3) \quad k_i > 0
\end{aligned} \tag{1.6}$$

Where $\eta_{s_{k-1}}$ is the previous 3 DOF robot pose (x,y,yaw) and $\nu_{s_{k-1}}$ is the previous robot velocity (velocity in the direction of x and y B-Frame axis of the robot and the angular velocity). u_{s_k} is the input to the motion model contaning the desired robot velocity in the x direction (u_d) and the desired angular velocity around the z axis (r_d). w_{s_k} is the motion model noise representing an acceleration perturbation in the robot axis. The w_{s_k} acceleration is the responsible for the slight velocity variation in the simulated robot motion. K is a diagonal matrix containing the gains used to drive the simulated velocity towards the desired input velocity.

Finally, the class updates the object attributes xsk, xsk_1 and usk to made them available for plotting purposes.

To be completed by the student.

Parameters

- **xsk_1** – previous robot state $x_{s_{k-1}} = \begin{bmatrix} \eta^T_{s_{k-1}} & \nu^T_{s_{k-1}} \end{bmatrix}^T$
- **usk** – model input $u_{s_k} = \nu_d = \begin{bmatrix} u_d & r_d \end{bmatrix}^T$

Returns current robot state x_{s_k}

ReadEncoders()

Simulates the robot measurements of the left and right wheel encoders.

To be completed by the student.

Return zsk,Rsk $zk = [n_L \; n_R]^T$ observation vector containing number of pulses read from the left and right wheel encoders. $R_{s_k} = diag(\sigma_L^2, \sigma_R^2)$ covariance matrix of the read pulses.

ReadCompass()

Simulates the compass reading of the robot.

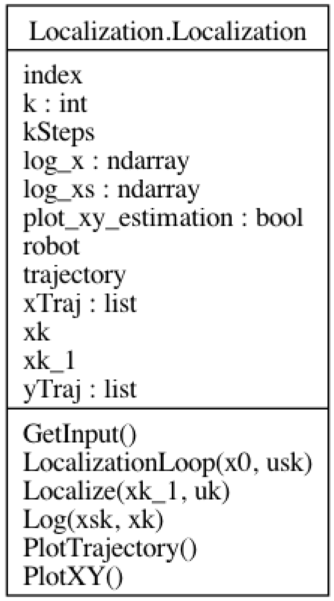
Returns yaw and the covariance of its noise *R_yaw*

PlotRobot()

Updates the plot of the robot at the current pose

1.3 Robot Localization

1.3.1 Robot Localization



class `Localization.`**`Localization`**(*index*, *kSteps*, *robot*, *x0*, **args*)

Bases: `object`

Localization base class. Implements the localization algorithm.

`__init__`(*index*, *kSteps*, *robot*, *x0*, **args*)

Constructor of the DRLocalization class.

Parameters

- **`index`** – Logging index structure (`prpy.Index`)
- **`kSteps`** – Number of time steps to simulate
- **`robot`** – Simulation robot object (`prpy.Robot`)
- **`args`** – Rest of arguments to be passed to the parent constructor
- **`x0`** – Initial Robot pose in the N-Frame

`GetInput`()

Gets the input from the robot. To be overidden by the child class.

Return uk input variable

`Localize`(*xk_1*, *uk*)

Single Localization iteration invoked from `prpy.DRLocalization.Localization()`. Given the previous robot pose, the function reads the inout and computes the current pose.

Parameters **`xk_1`** – previous robot pose

Return xk current robot pose

`LocalizationLoop`(*x0*, *usk*)

Given an initial robot pose x_0 and the input to the `prpy.SimulatedRobot` this method calls iteratively `prpy.DRLocalization.Localize()` for k steps, solving the robot localization problem.

Parameters **x0** – initial robot pose

Log(*xsk*, *xk*)
Logs the results for later plotting.

Parameters

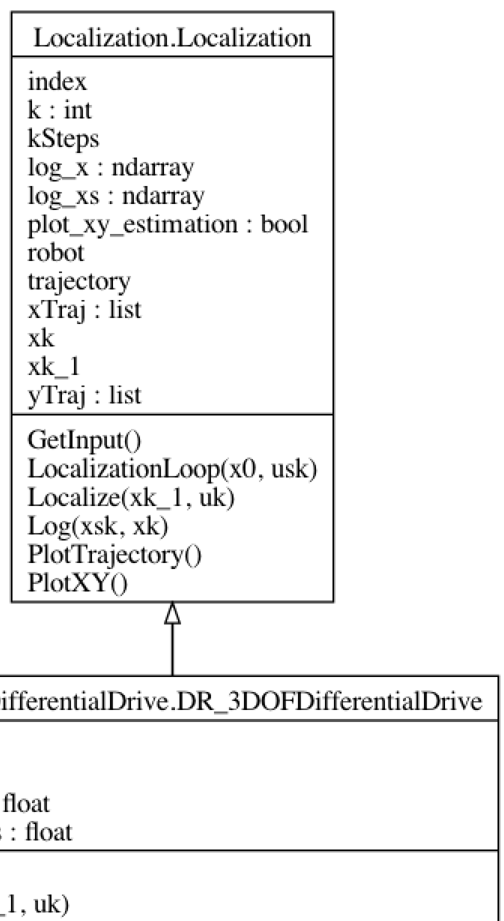
- **xsk** – ground truth robot pose from the simulation
- **xk** – estimated robot pose

PlotXY()
Plots, in a new figure, the ground truth (orange) and estimated (blue) trajectory of the robot at the end of the Localization Loop.

PlotTrajectory()
Plots the estimated trajectory (blue) of the robot during the localization process.

1.3.2 Dead Reckoning

3 DOF Differential Drive Mobile Robot Example



class DR_3DOFDifferentialDrive.**DR_3DOFDifferentialDrive**(*index*, *kSteps*, *robot*, *x0*, **args*)
Bases: *Localization.Localization*

Dead Reckoning Localization for a Differential Drive Mobile Robot.

`__init__`(*index*, *kSteps*, *robot*, *x0*, **args*)
Constructor of the `prlab.DR_3DOFDifferentialDrive` class.

Parameters **`args`** – Rest of arguments to be passed to the parent constructor

`Localize`(*xk_1*, *uk*)
Motion model for the 3DOF ($x_k = [x_k \; y_k \; \psi_k]^T$) Differential Drive Mobile robot using as input the readings of the wheel encoders ($u_k = [n_L \; n_R]^T$).

Parameters

- **`xk_1`** – previous robot pose estimate ($x_{k-1} = [x_{k-1} \; y_{k-1} \; \psi_{k-1}]^T$)
- **`uk`** – input vector ($u_k = [u_k \; v_k \; w_k \; r_k]^T$)

Return xk current robot pose estimate ($x_k = [x_k \; y_k \; \psi_k]^T$)

`GetInput`()
Get the input for the motion model. In this case, the input is the readings from both wheel encoders.

Returns uk: input vector ($u_k = [n_L \; n_R]^T$)

1.4 Particle Filter

`class ParticleFilter.`**`ParticleFilter`**(*index*, *kSteps*, *robot*, *particles*, **args*)
Bases: *`Localization.Localization`*

Particle Filter Localization.

This class implements basic plotting and logging functionality for the Particle Filter, as well as the interface for the child classes to implement.

A particle filter is a Monte Carlo algorithm that approximates the posterior distribution of the robot by a set of weighted particles. Note that the "weight" (which is a terrible term) is simply the probability of the particle being correct. Therefore, each particle is an estimate, and each estimate has some probability of being correct.

`__init__`(*index*, *kSteps*, *robot*, *particles*, **args*)
Constructor of the Particle Filter class.

Parameters

- **`index`** – Logging index structure (`Index`)
- **`kSteps`** – Number of time steps to simulate
- **`robot`** – Simulation robot object (`Robot`)
- **`particles`** – initial particles as a list of Pose objects (or at least a list of numpy arrays)
- **`args`** – Rest of arguments to be passed to the parent constructor

`MotionModel`(*particle*, *u*, *noise*)
" Motion model of the Particle Filter to be overwritten by the child class.

Parameters

- **`particle`** – particle state vector
- **`uk`** – input vector
- **`noise`** – sample from a noise distribution to be added to the input

Return particle updated particle state vector

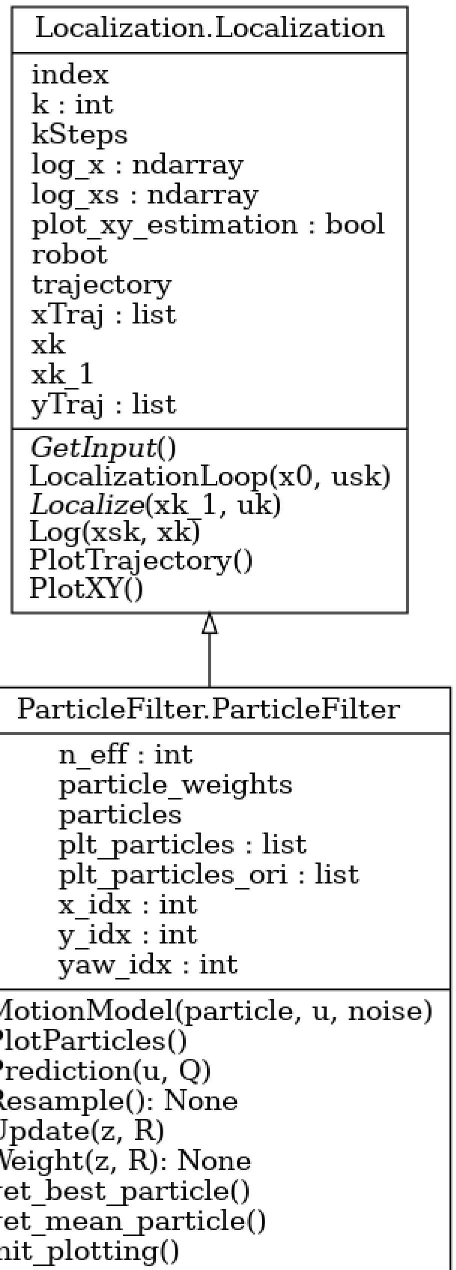


Fig. 3: ParticleFilter Class Diagram.

`Weight`(z, R) → None

Weight each particle by the liklihood of the particle being correct. The probability the particle is correct is given by the probability that it is correct given the measurements (z).

Parameters

- **z** – measurement vector
- **R** – measurement noise covariance

Returns None

`Resample`() → None

Resample the particles based on their weights to ensure diversity and prevent particle degeneracy.

This function implements the resampling step of a particle filter algorithm. It uses the weights assigned to each particle to determine their likelihood of being selected. Particles with higher weights are more likely to be selected, while those with lower weights have a lower chance.

The resampling process helps to maintain a diverse set of particles that better represents the underlying probability distribution of the system state.

After resampling, the attributes 'particles' and 'weights' of the ParticleFilter instance are updated to reflect the new set of particles and their corresponding weights.

Returns None

`Prediction`(u, Q)

Predict the next state of the system based on a given motion model.

This function updates the state of each particle by predicting its next state using a motion model.

Parameters

- **u** – input vector
- **Q** – the covariance matrix associated with the input vector

Returns None

`Update`(z, R)

Update the particle weights based on sensor measurements and perform resampling.

This function adjusts the weights of particles based on how well they match the sensor measurements.

The updated weights reflect the likelihood of each particle being the true state of the system given the sensor measurements.

After updating the weights, the function may perform resampling to ensure that particles with higher weights are more likely to be selected, maintaining diversity and preventing particle degeneracy.

Parameters

- **z** – measurement vector
- **R** – the covariance matrix associated with the measurement vector

`get_mean_particle`()

Calculate the mean particle based on the current set of particles and their weights. :return: mean particle

`get_best_particle`()

Calculate the best particle based on the current set of particles and their weights. :return: best particle

`init_plotting`()

Init the plotting of the particles and the mean particle.

PlotParticles()

Plots all the particles and the mean particle. Particles are plotted as green dots, and the mean particle is plotted as a blue dot. Particle orientation is plotted as a green line, and the mean particle orientation is plotted as a blue line. Particle size is proportional to the particle weight. Note that the size is scaled for visualization purposes, and does not reflect the actual weight.

1.5 MonteCarlo Localization

class MCLocalization.**MCLocalization**(*index*, *kSteps*, *robot*, *particles*, **args*)

Bases: *ParticleFilter.ParticleFilter*

Monte Carlo Localization class.

This class is used as "Dead Reckoning" localization using a Particle Filter. It implements the Prediction method from `ParticleFilter` and the Localize and LocalizationLoop methods from `Localization`.

__init__(*index*, *kSteps*, *robot*, *particles*, **args*)

Constructor. :param index: Named tuple used to map the state vector, the simulation vector and the observation vector (`prpy.IndexStruct`) :param kSteps: simulation time steps :param robot: Simulated Robot object :param particles: initial particles as a list of Pose objects (or at least a list of numpy arrays) :param args: arguments to be passed to the parent constructor

Prediction(*u*, *Q*)

Prediction overriden from `ParticleFilter`. Note: Use the MotionModel method from `ParticleFilter` to update the particles to keep it generic. Then, child classes can overwrite the MotionModel method to implement their own motion model.

Localize()

Single Localization iteration. Given the previous robot pose, the function reads the inout and computes the current pose.

Returns **xk** current robot pose (we can assume the mean of the particles or the most likely particle)

LocalizationLoop(*x0*, *usk*)

Given an initial robot pose x_0 and the input to the `SimulatedRobot` this method calls iteratively `DRLocalization.Localize()` for k steps, solving the robot localization problem.

Parameters

- **x0** – initial robot pose
- **usk** – input vector for the simulation

1.6 Particle Filter Map Based Localization

class PFMBLocalization.**PFMBL**(*zf_dim*, *M*, **args*)

Bases: *MCLocalization.MCLocalization*

Particle Filter Map Based Localization class.

This class defines a Map Based Localization using a Particle Filter. It inherits from `MCLocalization`, so the Prediction step is already implemented. It needs to implement the Update function, and consecuently the Weight and Resample functions.

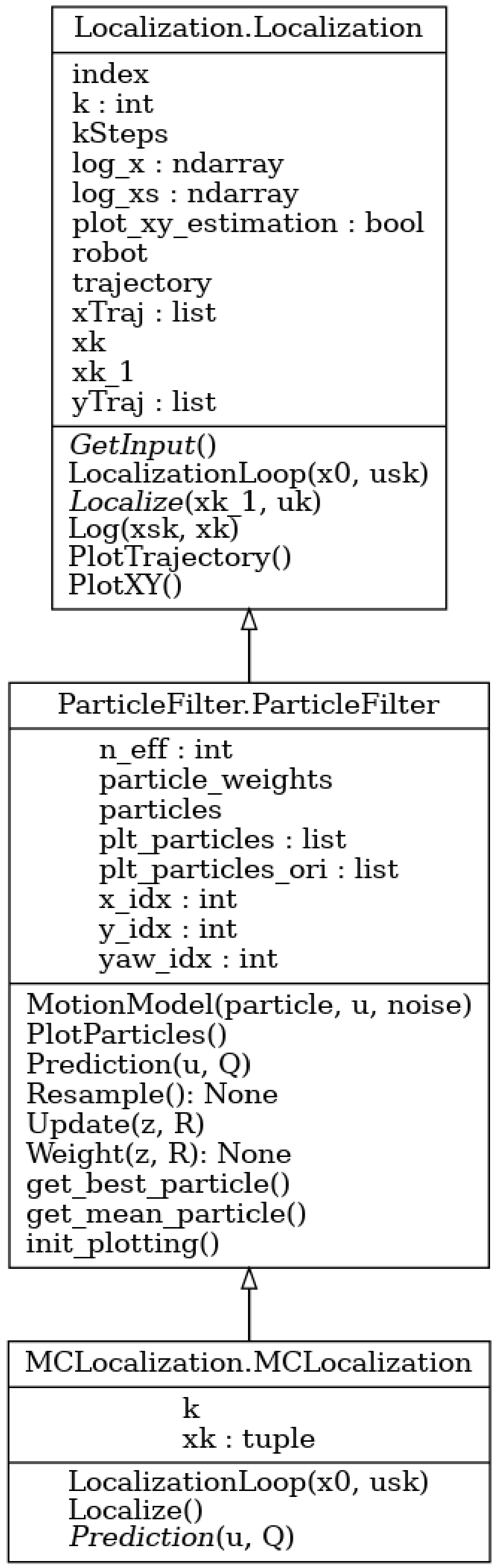


Fig. 4: MCLocalization Class Diagram.

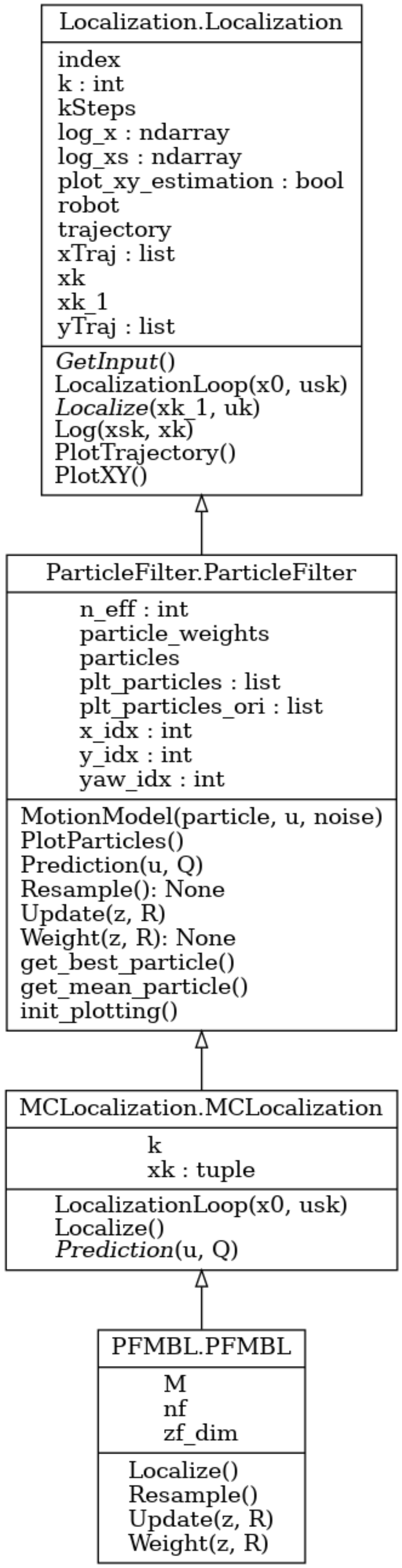


Fig. 5: PFMBL Class Diagram.

`__init__`(*zf_dim*, *M*, **args*) → None

Constructor. :param index: Named tuple used to map the state vector, the simulation vector and the observation vector (`prpy.IndexStruct`) :param kSteps: simulation time steps :param robot: Simulated Robot object :param particles: initial particles as a list of Pose objects (or at least a list of numpy arrays) :param args: arguments to be passed to the parent constructor

`Weight`(*z*, *R*)

Weight each particle by the liklihood of the particle being correct. The probability the particle is correct is given by the probability that it is correct given the measurements (z).

Parameters

- **z** – measurement vector
- **R** – measurement noise covariance

Returns None

`Resample`()

Resample the particles based on their weights to ensure diversity and prevent particle degeneracy.

This function implements the resampling step of a particle filter algorithm. It uses the weights assigned to each particle to determine their likelihood of being selected. Particles with higher weights are more likely to be selected, while those with lower weights have a lower chance.

The resampling process helps to maintain a diverse set of particles that better represents the underlying probability distribution of the system state.

After resampling, the attributes 'particles' and 'weights' of the ParticleFilter instance are updated to reflect the new set of particles and their corresponding weights.

Returns None

`Update`(*z*, *R*)

Update the particle weights based on sensor measurements and perform resampling.

This function adjusts the weights of particles based on how well they match the sensor measurements.

The updated weights reflect the likelihood of each particle being the true state of the system given the sensor measurements.

After updating the weights, the function may perform resampling to ensure that particles with higher weights are more likely to be selected, maintaining diversity and preventing particle degeneracy.

Parameters

- **z** – measurement vector
- **R** – the covariance matrix associated with the measurement vector

`Localize`()

Single Localization iteration. Given the previous robot pose, the function reads the inout and computes the current pose.

Returns **xk** current robot pose (we can assume the mean of the particles or the most likely particle)

1.7 PF 3DOF Dead Reckoning

class PF_3DOF_DR.**PF_3DOF_DR**(**args*)

Bases: *MCLocalization.MCLocalization*

__init__(**args*)

Constructor. :param index: Named tuple used to map the state vector, the simulation vector and the observation vector (`prpy.IndexStruct`) :param kSteps: simulation time steps :param robot: Simulated Robot object :param particles: initial particles as a list of Pose objects (or at least a list of numpy arrays) :param args: arguments to be passed to the parent constructor

GetInput()

Get the input for the motion model.

Returns

- **uk, Qk**. uk: input vector ($u_k = [n_L \; n_R]^T$), Qk: covariance of the input noise

MotionModel(*particle*, *u*, *noise*)

" Motion model of the Particle Filter to be overwritten by the child class.

Parameters

- **particle** – particle state vector
- **uk** – input vector
- **noise** – sample from a noise distribution to be added to the input

Return particle updated particle state vector

1.8 PF 3DOF Map Based Localization

class PF_3DOF_MBL.**PF_3DOF_MBL**(**args*)

Bases: *PFMBLocalization.PFMBL*

__init__(**args*)

Constructor. :param index: Named tuple used to map the state vector, the simulation vector and the observation vector (`prpy.IndexStruct`) :param kSteps: simulation time steps :param robot: Simulated Robot object :param particles: initial particles as a list of Pose objects (or at least a list of numpy arrays) :param args: arguments to be passed to the parent constructor

GetInput()

Get the input for the motion model.

Returns

- **uk, Qk**. uk: input vector ($u_k = [n_L \; n_R]^T$), Qk: covariance of the input noise

GetMeasurements()

Read the measurements from the robot. Returns a vector of range distances to the map features. Only those features that are within the `SimulatedRobot.SimulatedRobot.Distance_max_range` of the sensor are returned. The measurements arribe at a frequency defined in the `SimulatedRobot.SimulatedRobot.Distance_feature_reading_frequency` attribute.

Returns vector of distances to the map features, covariance of the measurement noise

MotionModel(*particle*, *u*, *noise*)

" Motion model of the Particle Filter to be overwritten by the child class.

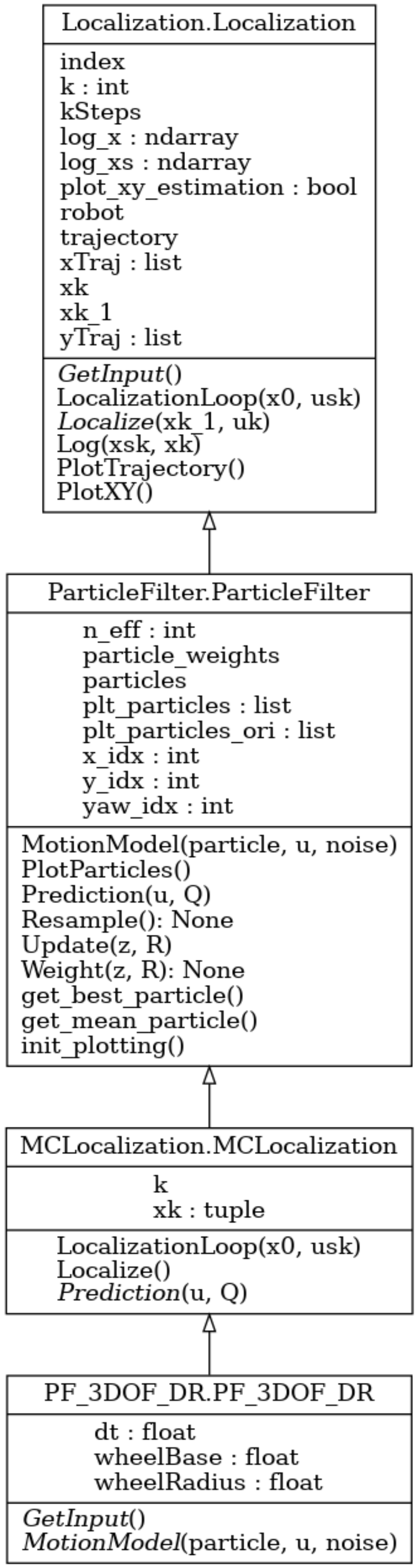


Fig. 6: PF_3DOF_DR Class Diagram.

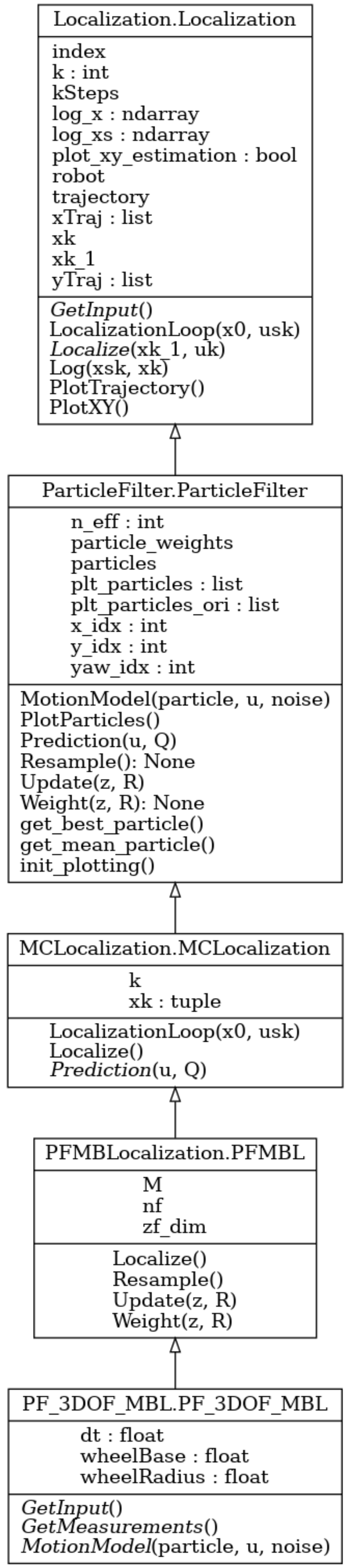


Fig. 7: PF_3DOF_MBL Class Diagram.

Parameters

- `particle` – particle state vector
- `uk` – input vector
- `noise` – sample from a noise distribution to be added to the input

Return particle updated particle state vector

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

D

F

G

I

L

M

O

P

R

S

U

W