prpy: Probabilistic Robot Localization Python Library

Release 0.1

Pere Ridao

Oct 23, 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: `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

`J_1oplus`(*BxC*)

Jacobian of the pose compounding operation (eq. (1.1)) with respect to the first pose:

$$
J_{1\oplus} = \frac{\partial^{A} x_B \oplus^{B} x_C}{\partial^{A} x_B} = \begin{bmatrix} 1 & 0 & -^{B}x_C \sin({}^{A}\psi_B) -^{B} y_C \cos({}^{A}\psi_B) \\ 0 & 1 & {}^{B}x_C \cos({}^{A}\psi_B) -^{B} y_C \sin({}^{A}\psi_B) \\ 0 & 0 & 1 \end{bmatrix} \tag{1.2}
$$

The method returns a numerical matrix containing the evaluation of the Jacobian for the pose *AxB* (the self object) and the 2^{nd} posepose *BxC*.

Parameters
: **`BxC`** – 2nd pose

Returns
: Evaluation of the $J_{1\oplus}$ Jacobian of the pose compounding operation with respect to the first pose (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)

J_2oplus()

Jacobian of the pose compounding operation ((1.1)) with respect to the second pose:

$$J_{2\oplus} = \frac{\partial^A x_B \oplus^B x_C}{\partial^B x_C} = \begin{bmatrix} \cos(^A\psi_B) & -\sin(^A\psi_B) & 0 \\ \sin(^A\psi_B) & \cos(^A\psi_B) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.3}$$

The method returns a numerical matrix containing the evaluation of the Jacobian for the 1^{st} posepose *AxB* (the self object).

Returns

Evaluation of the $J_{2\oplus}$ Jacobian of the pose compounding operation with respect to the second pose (eq. (1.3))

ominus()

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

$$^B x_A = \ominus^A x_B = \begin{bmatrix} -^A x_B \cos(^A\psi_B) -^A y_B \sin(^A\psi_B) \\ ^A x_B \sin(^A\psi_B) -^A y_B \cos(^A\psi_B) \\ -^A\psi_B \end{bmatrix} \tag{1.4}$$

Returns

A-Frame pose expressed in B-Frame coordinates (eq. (1.4))

J_ominus()

Jacobian of the inverse pose compounding operation ((1.1)) with respect the pose *AxB* (the self object):

$$J_{\ominus} = \frac{\partial \ominus^A x_B}{\partial^A x_B} = \begin{bmatrix} -\cos(^A\psi_B) & -\sin(^A\psi_B) & ^A x_B \sin(^A\psi_B) -^A y_B \cos(^A\psi_B) \\ \sin(^A\psi_B) & -\cos(^A\psi_B) & ^A x_B \cos(^A\psi_B) +^A y_B \sin(^A\psi_B) \\ 0 & 0 & -1 \end{bmatrix} \tag{1.5}$$

Returns the numerical matrix containing the evaluation of the Jacobian for the pose *AxB* (the self object).

Returns

Evaluation of the $J_{\ominus}$ Jacobian of the inverse pose compounding operation with respect to the pose (eq. (1.5))

1.2 Robot Simulation

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

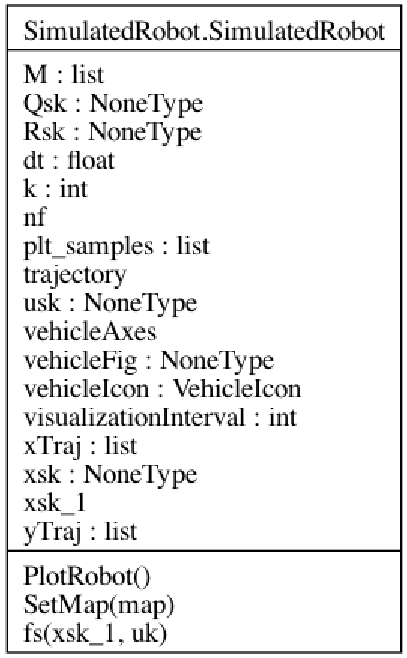


Fig. 1: SimulatedRobot Class Diagram.

- **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*, *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. 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`

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.

$$Q_k = \begin{bmatrix} \sigma_{\dot u}^2 & 0 & 0 \\ 0 & \sigma_{\dot v}^2 & 0 \\ 0 & 0 & \sigma_{\dot r}^2 \end{bmatrix} \tag{1.6}$$

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

$$\mathbf{u_k} = \begin{bmatrix} u_k & r_k \end{bmatrix}^{\mathbf{T}} \tag{1.7}$$

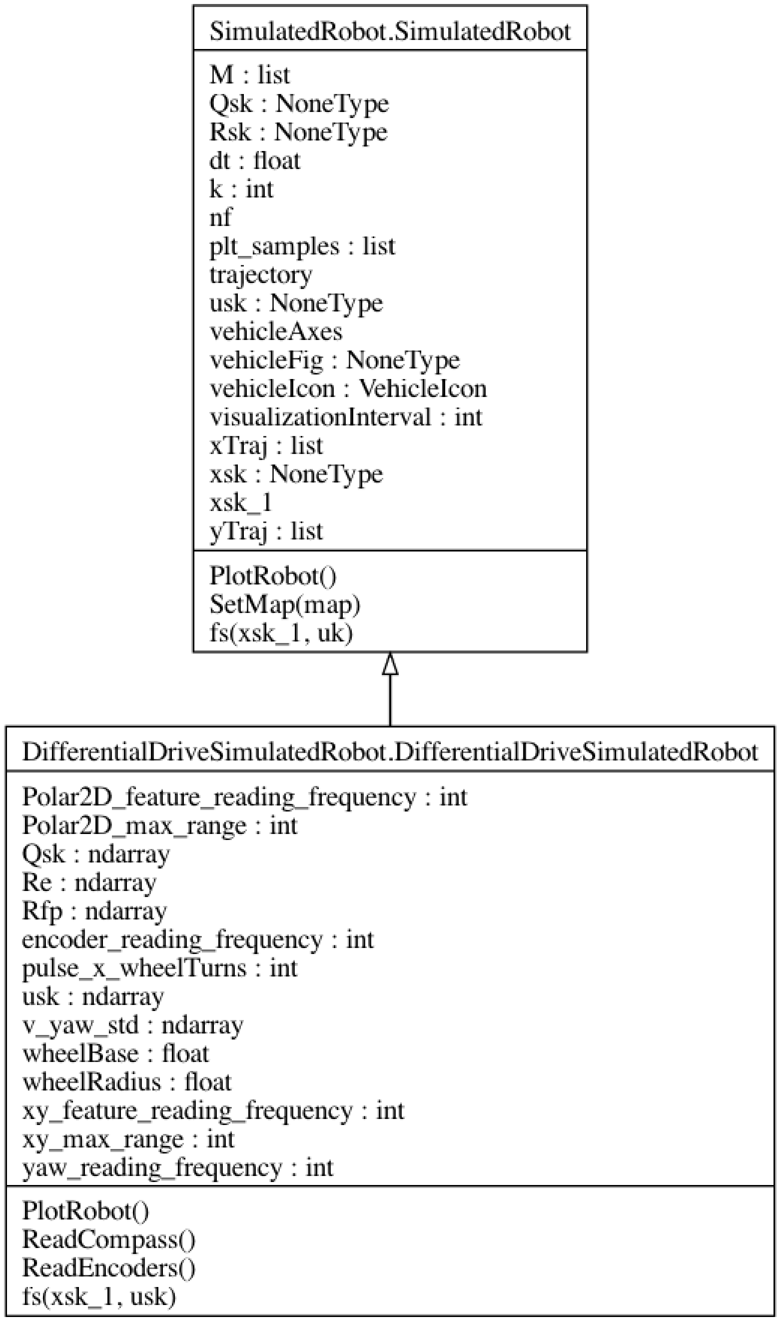


Fig. 2: DifferentialDriveSimulatedRobot Class Diagram.

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

$$x_k = \begin{bmatrix} {}^N x_k & {}^N y_k & {}^N \theta_k & {}^B u_k & {}^B v_k & {}^B r_k \end{bmatrix}^T \tag{1.8}$$

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.

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:

$$\begin{aligned}
\eta_{s_{k-1}} &= \begin{bmatrix} x_{s_{k-1}} & y_{s_{k-1}} & \theta_{s_{k-1}} \end{bmatrix}^T \\
\nu_{s_{k-1}} &= \begin{bmatrix} u_{s_{k-1}} & v_{s_{k-1}} & r_{s_{k-1}} \end{bmatrix}^T \\
x_{s_{k-1}} &= \begin{bmatrix} \eta_{s_{k-1}}^T & \nu_{s_{k-1}}^T \end{bmatrix}^T \\
u_{s_k} &= \nu_d = \begin{bmatrix} u_d & r_d \end{bmatrix}^T \\
w_{s_k} &= \dot{\nu}_{s_k} \\
x_{s_k} &= f_s(x_{s_{k-1}}, u_{s_k}, w_{s_k}) \\
&= \begin{bmatrix} \eta_{s_{k-1}} \oplus (\nu_{s_{k-1}} \Delta t + \frac{1}{2} w_{s_k} \Delta t^2) \\ \nu_{s_{k-1}} + K(\nu_d - \nu_{s_{k-1}}) + w_{s_k} \Delta t \end{bmatrix} \quad ; \quad K = diag(k_1, k_2, k_3) \quad k_i > 0
\end{aligned} \tag{1.9}$$

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 $x_{s_{k-1}} = \begin{bmatrix} \eta_{s_{k-1}}^T & \nu_{s_{k-1}}^T \end{bmatrix}^T$
- **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
$zk = [\Delta n_L \ \Delta n_R]^T$ observation vector containing number of pulses read from the left and right wheel encoders during the last differential motion. $R_{s_k} = diag(\sigma_L^2, \sigma_R^2)$ covariance matrix of the read pulses.

`ReadCompass()`

Simulates the compass reading of the robot.

Returns
yaw and the covariance of its noise *R_yaw*

`ReadCartesian2DFeature()`

Simulates the reading of 2D cartesian features. The features are placed in the map in cartesian coordinates.

Returns

zsk: [[x1 y1],...,[xn yn]]
Cartesian position of the feature observations.

Rsk: block_diag(R_1,...,R_n), where R_i=[[r_xx r_xy],[r_xy r_yy]] is the
2x2 i-th feature observation covariance. Covariance of the Cartesian feature observations. Note the features are uncorrelated among them. They are independent. However, the x and y coordinates of each feature are correlated.

`ReadRanges()`

Simulates the reading of distance towards 2D Cartessian features. Returns a vector of distances towards the features within the maximum range `Distance_max_range`. The functions works at a frequency of `Distance_feature_reading_frequency`.

Returns
vector of distances towards the features.

`PlotRobot()`

Updates the plot of the robot at the current pose

1.3 Filters

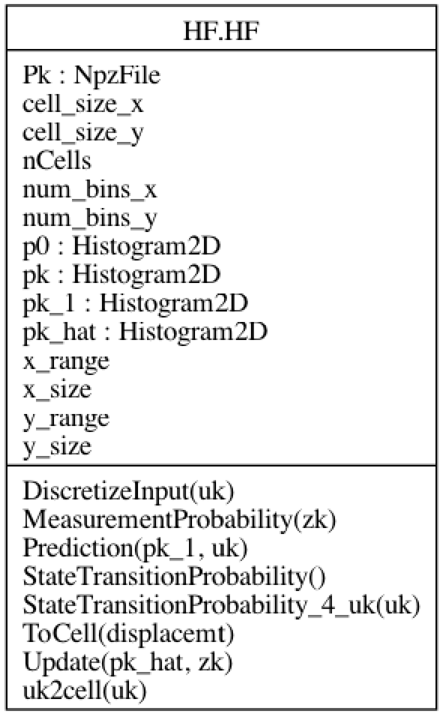
1.3.1 Histogram Filter

Histogram Filter

class `HF.HF(p0, *args)`

Bases: `object`

Histogram Filter base class. Implements the histogram filter algorithm using a discrete Bayes Filter.



`__init__`(*p0*, **args*)

” The histogram filter is initialized with the initial probability histogram *p0* and the state transition probability matrix *Pk*. The state transition probability matrix is computed by the derived class through the pure virtual method *StateTransitionProbability*. The histogram filter is implemented as a discrete Bayes Filter. The state transition probability matrix is used in the prediction step and the measurement probability matrix is used in the update step. :param p0: initial probability histogram

`ToCell`(*displacemt*)

Converts a metric displacement to a cell displacement.

Parameters
`displacemt` – input displacement in meters

Returns
displacement in cells

`StateTransitionProbability`()

Returns the state transition probability matrix. This is a pure virtual method that must be implemented by the derived class.

Returns
Pk state transition probability matrix

`StateTransitionProbability_4_uk`(*uk*)

Returns the state transition probability matrix for the given control input *uk*. This is a pure virtual method that must be implemented by the derived class.

Parameters
`uk` – control input. In localization, this is commonly the robot displacement. For example, in the case of a differential drive robot, this is the robot displacement in the robot frame commonly computed through the odometry.

Returns
Puk state transition probability matrix for a given uk

`MeasurementProbability`(*zk*)

Returns the measurement probability matrix for the given measurement *zk*. This is a pure virtual method that must be implemented by the derived class.

Parameters
zk – measurement.

Returns
pzk measurement probability histogram

`uk2cell`(*uk*)

Converts the control input *uk* to a cell displacement. :param uk: :return:

`Prediction`(*pk_1*, *uk*)

Computes the prediction step of the histogram filter. Given the previous probability histogram *pk_1* and the control input *uk*, it computes the predicted probability histogram *pk_hat* after the robot displacement *uk* according to the motion model described by the state transition probability.

Parameters

- **pk_1** – previous probability histogram
- **uk** – control input

Returns
pk_hat predicted probability histogram

`Update`(*pk_hat*, *zk*)

Computes the update step of the histogram filter. Given the predicted probability histogram *pk_hat* and the measurement *zk*, it computes first the measurement probability histogram *pzk* and then uses the Bayes Rule to compute the updated probability histogram *pk*. :param pk_hat: predicted probability histogram :param zk: measurement :return: pk: updated probability histogram

`class Histogram.Histogram2D`(*num_bins_x*, *num_bins_y*, *x_range*, *y_range*)

Bases: `object`

Class for creating and manipulating a 2D histogram.

`__init__`(*num_bins_x*, *num_bins_y*, *x_range*, *y_range*)

Initialize a new Histogram2D instance.

Param
num_bins_x (int): Number of bins in the X-direction. num_bins_y (int): Number of bins in the Y-direction. x_range (numpy.ndarray): Range of values for the X-axis. y_range (numpy.ndarray): Range of values for the Y-axis.

`property histogram_2d`

Get the 2D histogram data as a NumPy array.

Returns
numpy.ndarray: The 2D histogram data.

`property histogram_1d`

Get the histogram data as a 1D NumPy array.

Returns
numpy.ndarray: The 1D histogram data.

plot_histogram()

Plot the 2D histogram using Matplotlib.

property element

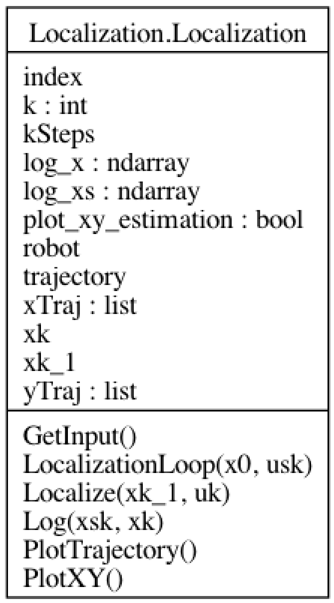
Property to access individual elements of the histogram using range values.

Returns

ElementAccessor: An instance of ElementAccessor for getting and setting individual elements by range.

1.4 Localization

1.4.1 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.

1.4.2 Dead Reckoning

3 DOF Differential Drive Mobile Robot Example

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

Bases: *Localization*

Dead Reckoning Localization for a Differential Drive Mobile Robot.

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

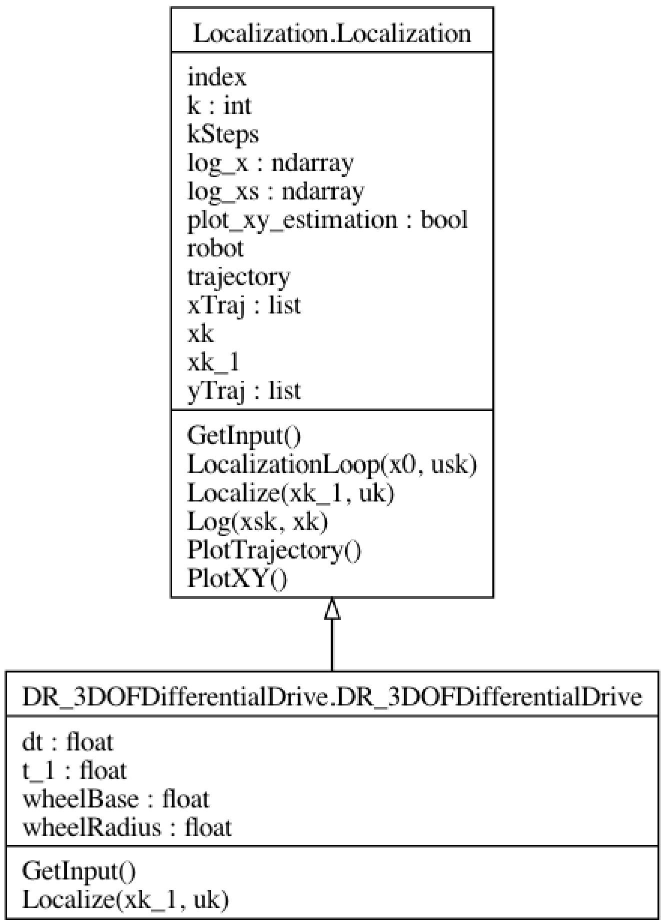
Constructor of the `prlab.DR_3DOFDifferentialDrive` class.

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


Localization.Localization
index
k : int
kSteps
log_x : ndarray
log_xs : ndarray
plot_xy_estimation : bool
robot
trajectory
xTraj : list
xk
xk_1
yTraj : list
GetInput()
LocalizationLoop(x0, usk)
Localize(xk_1, uk)
Log(xsk, xk)
PlotTrajectory()
PlotXY()
DR_3DOFDifferentialDrive.DR_3DOFDifferentialDrive
dt : float
t_1 : float
wheelBase : float
wheelRadius : float
GetInput()
Localize(xk_1, uk)

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

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

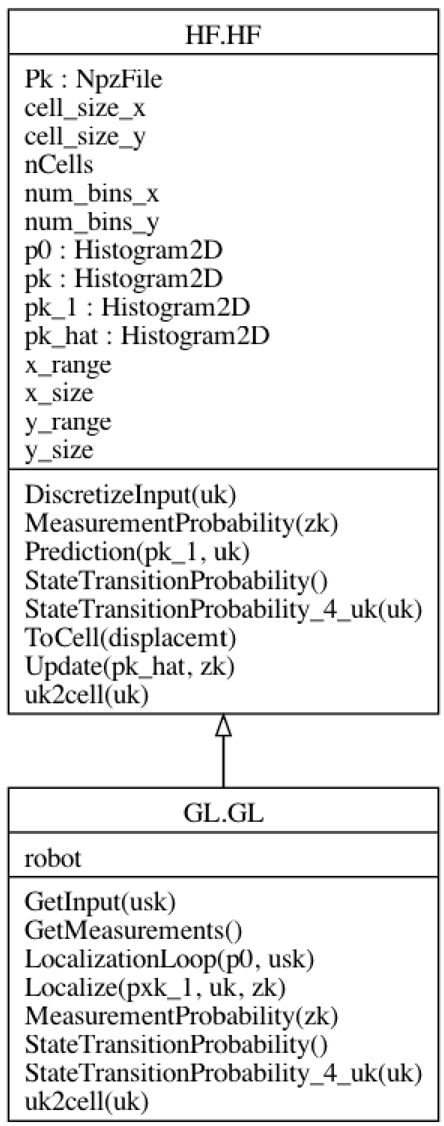
`GetInput()`

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

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

1.4.3 Grid Localization

Grid Localization



class GL.**GL**(*p0*, *index*, *kSteps*, *robot*, *x0*, **args*)

Bases: *HF*

Grid Localization base class. Inherits from a HF. Implements the grid localization algorithm using a discrete Bayes Filter.

__init__(*p0*, *index*, *kSteps*, *robot*, *x0*, **args*)

Constructor of the *GL* class. Initializes the Dead reckoning localization algorithm as well as the histogram filter algorithm.

Parameters

- **dx_max** – maximum x displacement in meters
- **dy_max** – maximum y displacement in meters
- **range_dx** – range of x displacements in meters
- **range_dy** – range of y displacements in meters
- **p0** – initial probability histogram
- **index** – index struture containing plotting information
- **kSteps** – number of time steps to simulate the robot motion
- **robot** – robot object
- **x0** – initial robot pose
- **args** – additional arguments

GetMeasurements()

Read the measurements from the robot. To be overriden by the child class.

StateTransitionProbability_4_uk(*uk*)

Returns the state transition probability matrix for the given control input *uk*. It is used in the `Predict()` method of the `HF` class, to compute the predicted probability histogram. This is a pure virtual method that must be implemented by the derived class.

Parameters

uk – control input. In localization, this is commonly the robot displacement. For example, in the case of a differential drive robot, this is the robot displacement in the robot frame commonly computed through the odometry.

Returns

Puk state transition probability matrix for a given uk

StateTransitionProbability()

Computes the complete state transition probability matrix. This is a pure virtual method that must be implemented by the derived class.

Returns

state transition probability matrix $P_k = px_k|x_{k-1}, uk$

MeasurementProbability(*zk*)

Computes the measurement probability histogram given the robot pose η_k and the measurement z_k. Method to be overriden by the child class.

Parameters

zk – vector of measurements

Returns

Measurement probability histogram $p_z = p(z_k|\eta_k)$

GetInput(*usk*)

Gets the number of cells the robot has displaced along its DOFs in the world N-Frame. Method to be overriden by the child class.

Parameters
usk – control input of the robot simulation. Required because it might be necessarey to call the :meth:SimulatedRobot.fs` method iterative until the robot displace at least one cell.

Returns
uk: vector containing the number of cells the robot has displaced in all the axis of the world N-Frame

uk2cell(*uk*)

" Converts the number of cells the robot has displaced along its DOFs in the world N-Frame to an index that can be used to acces the state transition probability matrix. This is a pure virtual method that must be implemented by the derived class.

Parameters
uk – vector containing the number of cells the robot has displaced in all the axis of the world N-Frame

Returns
index: index that can be used to access the state transition probability matrix

LocalizationLoop(*p0*, *usk*)

Given an initial position histogram p_0 and the input to the `DifferentialDrive.SimulatedRobot` this method calls iteratively *`GL.Localize()`* for k steps, solving the robot localization problem.

Parameters

- **p0** – initial robot pose
- **usk** – control input of the robot simulation

Localize(*pxk_1*, *uk*, *zk*)

Solves a localization iteration calling, successively to the `HF.Prediction()` first, followed by the `HF.Update()`.

Parameters

- **pxk_1** – histogram of the previous robot position
- **uk** – robot displacement in number of cells in the world N-Frame
- **zk** – vector containing the measurements of the robot position in the world N-Frame

Returns
pk: histogram of the robot position after the prediction and the update steps

AUV Grid Localization Example

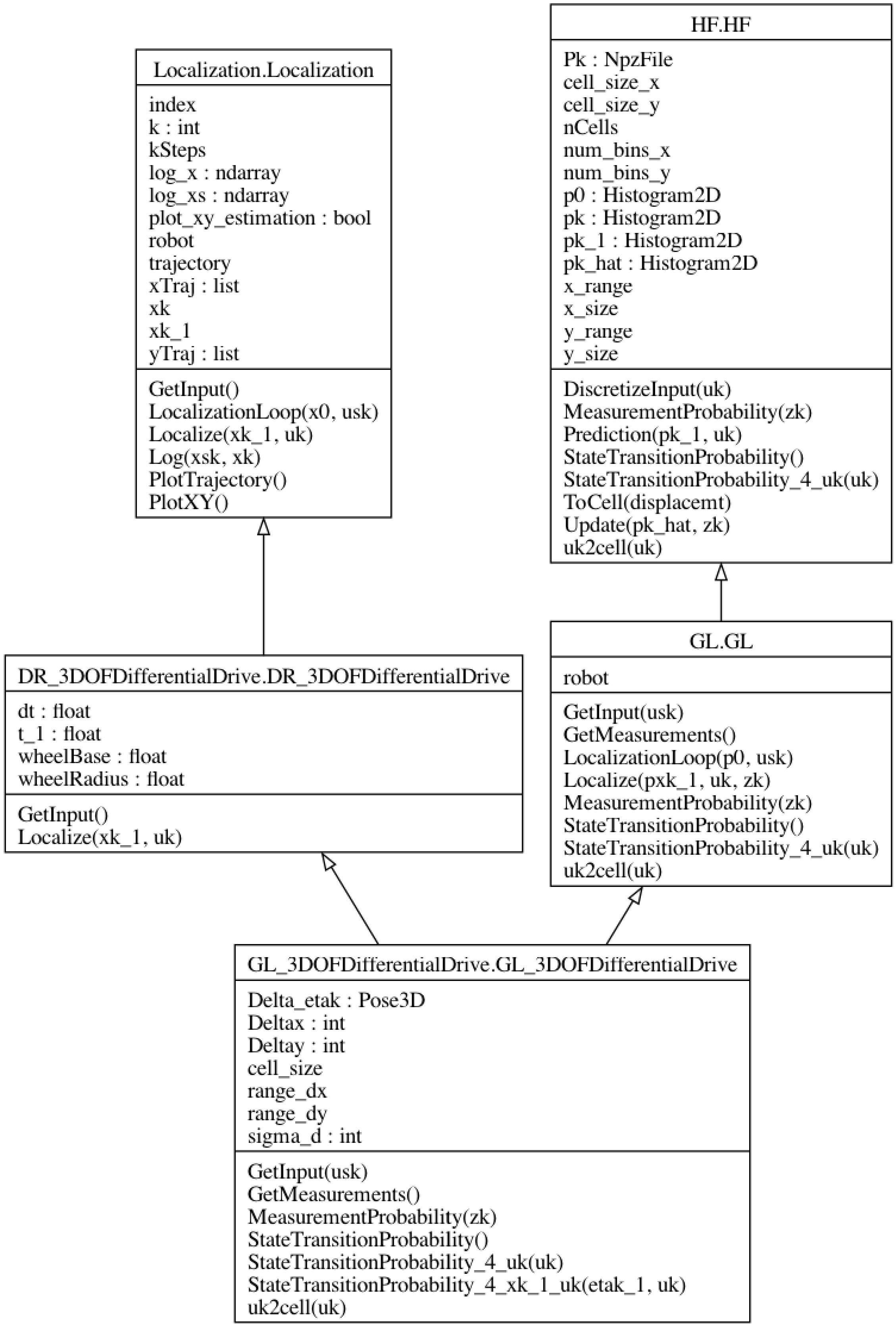
class GL_3DOFDifferentialDrive.**GL_3DOFDifferentialDrive**(*dx_max*, *dy_max*, *range_dx*, *range_dy*, *p0*, *index*, *kSteps*, *robot*, *x0*, **args*)

Bases: *`GL`*, *`DR_3DOFDifferentialDrive`*

Grid Reckoning Localization for a 3 DOF Differential Drive Mobile Robot.

__init__(*dx_max*, *dy_max*, *range_dx*, *range_dy*, *p0*, *index*, *kSteps*, *robot*, *x0*, **args*)

Constructor of the `GL_4DOFAUV` class. Initializes the Dead reckoning localization algorithm as well as the histogram filter algorithm.


HF.HF
Pk : NpzFile
cell_size_x
cell_size_y
nCells
num_bins_x
num_bins_y
p0 : Histogram2D
pk : Histogram2D
pk_1 : Histogram2D
pk_hat : Histogram2D
x_range
x_size
y_range
y_size
DiscretizeInput(uk)
MeasurementProbability(zk)
Prediction(pk_1, uk)
StateTransitionProbability()
StateTransitionProbability_4_uk(uk)
ToCell(displacemt)
Update(pk_hat, zk)
uk2cell(uk)
Localization.Localization
index
k : int
kSteps
log_x : ndarray
log_xs : ndarray
plot_xy_estimation : bool
robot
trajectory
xTraj : list
xk
xk_1
yTraj : list
GetInput()
LocalizationLoop(x0, usk)
Localize(xk_1, uk)
Log(xsk, xk)
PlotTrajectory()
PlotXY()
GL.GL
robot
GetInput(usk)
GetMeasurements()
LocalizationLoop(p0, usk)
Localize(pxk_1, uk, zk)
MeasurementProbability(zk)
StateTransitionProbability()
StateTransitionProbability_4_uk(uk)
uk2cell(uk)
DR_3DOFDifferentialDrive.DR_3DOFDifferentialDrive
dt : float
t_1 : float
wheelBase : float
wheelRadius : float
GetInput()
Localize(xk_1, uk)
GL_3DOFDifferentialDrive.GL_3DOFDifferentialDrive
Delta_etak : Pose3D
Deltax : int
Deltay : int
cell_size
range_dx
range_dy
sigma_d : int
GetInput(usk)
GetMeasurements()
MeasurementProbability(zk)
StateTransitionProbability()
StateTransitionProbability_4_uk(uk)
StateTransitionProbability_4_xk_1_uk(etak_1, uk)
uk2cell(uk)

Parameters

- **dx_max** – maximum x displacement in meters
- **dy_max** – maximum y displacement in meters
- **range_dx** – range of x displacements in meters
- **range_dy** – range of y displacements in meters
- **p0** – initial probability histogram
- **index** – index struture containing plotting information
- **kSteps** – number of time steps to simulate the robot motion
- **robot** – robot object
- **x0** – initial robot pose
- **args** – additional arguments

`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

`StateTransitionProbability_4_uk(`*uk*`)`

Returns the state transition probability matrix for the given control input *uk*. It is used in the `Predict()` method of the `HF` class, to compute the predicted probability histogram. This is a pure virtual method that must be implemented by the derived class.

Parameters
uk – control input. In localization, this is commonly the robot displacement. For example, in the case of a differential drive robot, this is the robot displacement in the robot frame commonly computed through the odometry.

Returns
Puk state transition probability matrix for a given uk

`StateTransitionProbability_4_xk_1_uk(`*etak_1*, *uk*`)`

Computes the state transition probability histogram given the previous robot pose η_{k-1} and the input u_k:

$$p(\eta_k|\eta_{k-1}, u_k)$$

Parameters

- **etak_1** – previous robot pose in cells
- **uk** – input displacement in number of cells

Returns
state transition probability $p_k = p(\eta_k|\eta_{k-1}, u_k)$

`StateTransitionProbability()`

Computes the complete state transition probability matrix. The matrix is a $n_u imesm_u imesn^2$ matrix, where n_u and m_u are the number of possible displacements in the x and y axis, respectively, and n is the number of cells in the map. For each possible displacement u_k, each previous robot pose x_{k-1} and each current robot pose x_k, the probability $p(x_k|x_{k-1}, u_k)$ is computed.

Returns
state transition probability matrix $P_k = px_k|x_{k-1}, uk$

`uk2cell`(*uk*)

Converts the number of cells the robot has displaced along its DOFs in the world N-Frame to an index that can be used to acces the state transition probability matrix.

Parameters
uk – vector containing the number of cells the robot has displaced in all the axis of the world N-Frame

Returns
index: index that can be used to access the state transition probability matrix

`MeasurementProbability`(*zk*)

Computes the measurement probability histogram given the robot pose η_k and the measurement z_k. In this case the measurement is the vector of the distances to the landmarks in the map.

Parameters
zk – $z_k = [r_0 \; r_1 \; ..r_k]$ where r_i is the distance to the i-th landmark in the map.

Returns
Measurement probability histogram $p_z = p(z_k|\eta_k)$

`GetInput`(*usk*)

Provides an implementation for the virtual method `GL.GetInput()`. Gets the number of cells the robot has displaced in the x and y directions in the world N-Frame. To do it, it calls several times the parent method `super().GetInput()`, corresponding to the Dead Reckoning Localization of the robot, until it has displaced at least one cell in any direction. Note that an iteration of the robot simulation `SimulatedRobot.fs()` is normally done in the `GL.LocalizationLoop()` method of the `GL.Localization` class, but in this case it is done here to simulate the robot motion between the consecutive calls to `super().GetInput()`.

Parameters
usk – control input of the robot simulation

Returns
uk: vector containing the number of cells the robot has displaced in the x and y directions in the world N-Frame

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

D

E

F

G

H

J

L

M

O

P

R

S

T

U