prpy: Probabilistic Robot Localization Python Library

Release 0.1

Pere Ridao

Nov 13, 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

class Pose.**Pose**

Bases: `ndarray`

Definition of a robot pose interface from where all the particular poses of different DOF inherit. This class defines a robot pose AxB as the pose of the B-Frame expressed in the A-Frame coordinates.

oplus(*BxC*)

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

The operation is defined as:

$$ {}^{\mathbf{A}}\mathbf{x_C} = {}^{\mathbf{A}}\mathbf{x_B} \oplus {}^{\mathbf{B}}\mathbf{x_C} \tag{1.1} $$

This is a pure virtual method that must be implemented by a child class.

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:

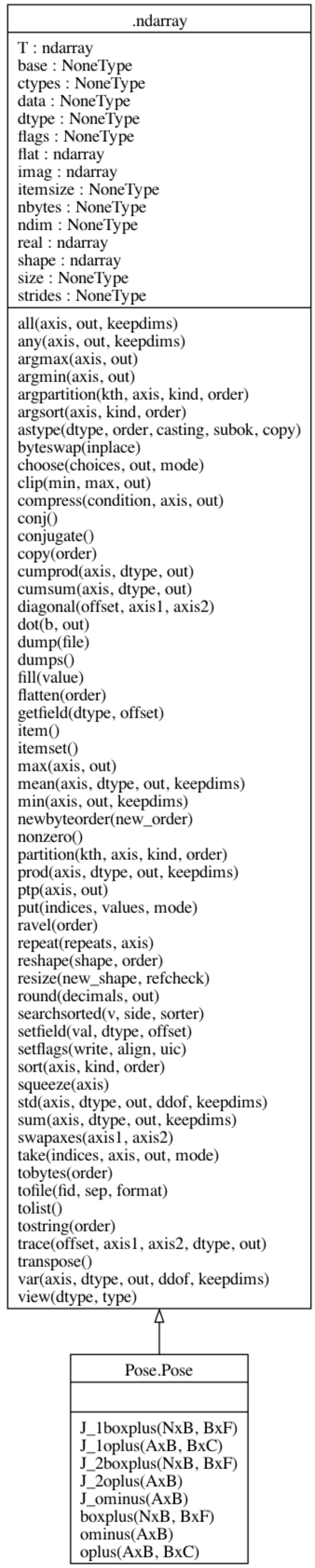
$$ J_{1\oplus} = \frac{\partial^A x_B \oplus^B x_C}{\partial^A x_B} \tag{1.2} $$

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

This is a pure virtual method that must be implemented by a child class.

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)
Pose.Pose
J_1boxplus(NxB, BxF)
J_1oplus(AxB, BxC)
J_2boxplus(NxB, BxF)
J_2oplus(AxB)
J_ominus(AxB)
boxplus(NxB, BxF)
ominus(AxB)
oplus(AxB, BxC)

J_2oplus()

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

$$J_{2\oplus} = \frac{\partial ^A x_B \oplus ^B x_C}{\partial ^B x_C} \tag{1.3}$$

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

This is a pure virtual method that must be implemented by a child class.

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 object):

$$^B x_A = \ominus ^A x_B \tag{1.4}$$

This is a pure virtual method that must be implemented by a child class.

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):

$$J_{\ominus} = \frac{\partial \ominus ^A x_B}{\partial ^A x_B} \tag{1.5}$$

Returns the numerical matrix containing the evaluation of the Jacobian for the pose *AxB* (the self object). **This is a pure virtual method that must be implemented by a child class.**

Returns

Evaluation of the $J_{\ominus}$ Jacobian of the inverse pose compounding operation with respect to the pose (eq. (1.12))

boxplus(*BxF*)

Given a Pose object *NxB* (the self object) and a Feature object *BxF*, it returns the Feature object *NxF* providing the same feature but now expresend in the N-Frame.

Parameters

BxF – Feature object expressed in the B-Frame

Returns

NxF Feature object expressed in the N-Frame

J_1boxplus(*BxF*)

Jacobian of the pose-feature compounding operation (eq. (1.20)) with respect to the robot pose:

$$J_{1\boxplus} = \frac{\partial ^N x_B \boxplus ^B x_F}{\partial ^N x_B} \tag{1.6}$$

Parameters

BxF – Feature object expressed in the B-Frame

Returns

$J_{1\boxplus}$ Jacobian of the feature compounding operation with respect to the robot pose (eq. (1.6))

`J_2boxplus`(*BxF*)

Jacobian of the pose-feature compounding operation (eq. (1.20)) with respect to the feature:

$$J_{2\boxplus} = \frac{\partial^N x_B \boxplus^B x_F}{\partial^B x_F} \tag{1.7}$$

Parameters

`BxF` – Feature object expressed in the B-Frame

Returns

$J_{2\boxplus}$ Jacobian of the feature compounding operation with respect to the feature (eq. (1.7))

1.1.2 Pose 3DOF

`class Pose.Pose3D`(*input_array=array([[0.], [0.], [0.]])*)

Bases: `Pose`

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.

`__init__`(*input_array=array([[0.], [0.], [0.]])*)

`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:

$${}^\mathbf{A}\mathbf{x_C} = {}^\mathbf{A}\mathbf{x_B} \oplus {}^\mathbf{B}\mathbf{x_C} = \begin{bmatrix} {}^A x_B +^B x_C \cos({}^A\psi_B) -^B y_C \sin({}^A\psi_B) \\ {}^A y_B +^B x_C \sin({}^A\psi_B) +^B y_C \cos({}^A\psi_B) \\ {}^A\psi_B +^B \psi_C \end{bmatrix} \tag{1.8}$$

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. (1.8)) with respect to the first pose:

$$J_{1\oplus} = \frac{\partial^A x_B \oplus^B x_C}{\partial^A x_B} = \begin{bmatrix} 1 & 0 & -^B x_C \sin({}^A\psi_B) -^B y_C \cos({}^A\psi_B) \\ 0 & 1 & {}^B x_C \cos({}^A\psi_B) -^B y_C \sin({}^A\psi_B) \\ 0 & 0 & 1 \end{bmatrix} \tag{1.9}$$

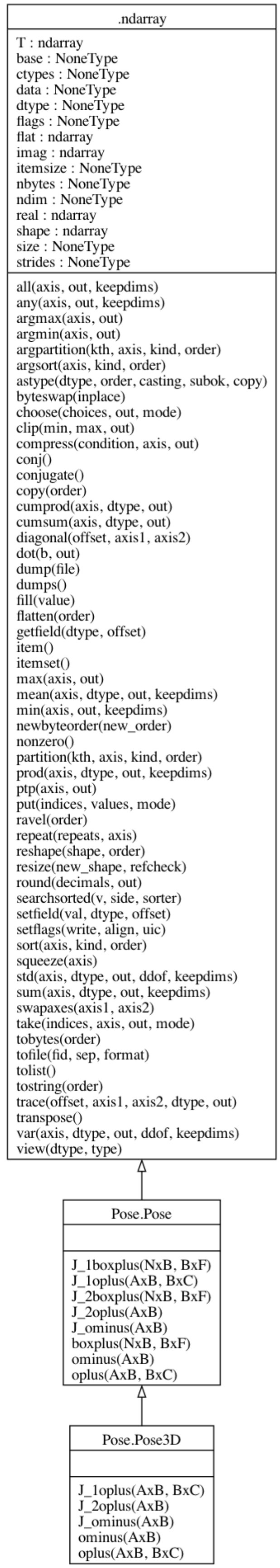
The method returns a numerical matrix containing the evaluation of the Jacobian for the pose *AxB* (the self object) and the 2^{nd} posepose *BxC*.

Parameters

`BxC` – 2nd pose

Returns

Evaluation of the $J_{1\oplus}$ Jacobian of the pose compounding operation with respect to the first pose (eq. (1.9))


.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)
Pose.Pose
J_1boxplus(NxB, BxF)
J_1oplus(AxB, BxC)
J_2boxplus(NxB, BxF)
J_2oplus(AxB)
J_ominus(AxB)
boxplus(NxB, BxF)
ominus(AxB)
oplus(AxB, BxC)
Pose.Pose3D
J_1oplus(AxB, BxC)
J_2oplus(AxB)
J_ominus(AxB)
ominus(AxB)
oplus(AxB, BxC)

J_2oplus()

Jacobian of the pose compounding operation (([1.8](#))) with respect to the second pose:

$$J_{2\oplus} = \frac{\partial {}^A x_B \oplus^B x_C}{\partial^B x_C} = \begin{bmatrix} \cos({}^A\psi_B) & -\sin({}^A\psi_B) & 0 \\ \sin({}^A\psi_B) & \cos({}^A\psi_B) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.10}$$

The method returns a numerical matrix containing the evaluation of the Jacobian for the :math:`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. (1.10))

ominus()

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

$${}^B x_A = \ominus {}^A x_B = \begin{bmatrix} -{}^A x_B \cos({}^A\psi_B) -^A y_B \sin({}^A\psi_B) \\ {}^A x_B \sin({}^A\psi_B) -^A y_B \cos({}^A\psi_B) \\ -{}^A\psi_B \end{bmatrix} \tag{1.11}$$

Returns

A-Frame pose expressed in B-Frame coordinates (eq. (1.11))

J_ominus()

Jacobian of the inverse pose compounding operation ((1.8)) with respect the pose *AxB* (the self object):

$$J_{\ominus} = \frac{\partial \ominus^A x_B}{\partial^A x_B} = \begin{bmatrix} -\cos({}^A\psi_B) & -\sin({}^A\psi_B) & {}^A x_B \sin({}^A\psi_B) -^A y_B \cos({}^A\psi_B) \\ \sin({}^A\psi_B) & -\cos({}^A\psi_B) & {}^A x_B \cos({}^A\psi_B) +^A y_B \sin({}^A\psi_B) \\ 0 & 0 & -1 \end{bmatrix} \tag{1.12}$$

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. (1.12))

1.1.3 Pose 4DOF

class Pose.**Pose4D**(*input_array=array([[0.], [0.], [0.], [0.]])*)

Bases: `Pose`

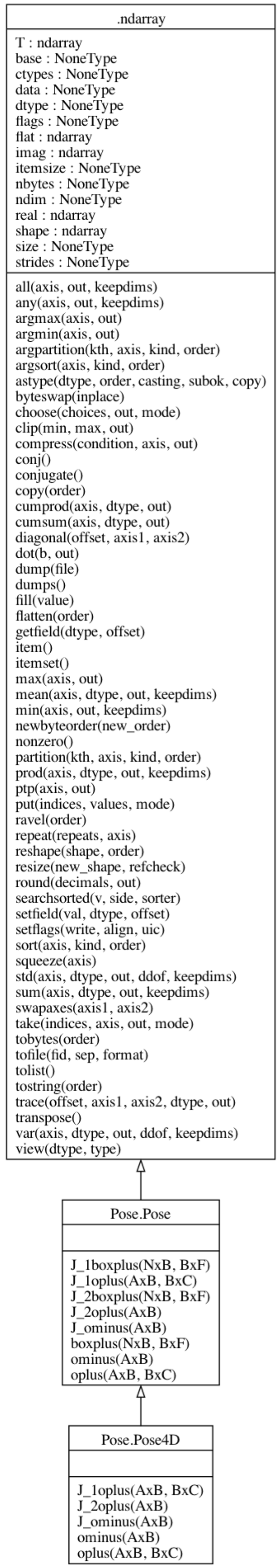
Definition of a robot pose in 4 DOF (x, y, yaw). The class inherits from a ndarray. This class extends the ndarray with the $oplus$ and :math:`ominus operators and the corresponding Jacobians.

__init__(*input_array=array([[0.], [0.], [0.], [0.]])*)

oplus(*BxC*)

Given a Pose3D object *AxB* (the self object) and a Pose3D object *BxC*, it returns the Pose4D object *AxC*.

$$\begin{aligned} {}^A x_B &= \begin{bmatrix} {}^A x_B & {}^A y_B & {}^A z_B & {}^A\psi_B \end{bmatrix}^T \\ {}^B x_C &= \begin{bmatrix} {}^B x_C & {}^B y_C & {}^B z_C & {}^B\psi_C \end{bmatrix}^T \\ {}^A x_C &=^A x_B \oplus^B x_C = \begin{bmatrix} {}^A x_B +^B x_C \cos({}^A\psi_B) -^B y_C \sin({}^A\psi_B) \\ {}^A y_B +^B x_C \sin({}^A\psi_B) +^B y_C \cos({}^A\psi_B) \\ {}^A z_B +^B z_C \\ {}^A\psi_B +^B \psi_C \end{bmatrix} \end{aligned} \tag{1.13}$$


.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)
Pose.Pose
J_1boxplus(NxB, BxF)
J_1oplus(AxB, BxC)
J_2boxplus(NxB, BxF)
J_2oplus(AxB)
J_ominus(AxB)
boxplus(NxB, BxF)
ominus(AxB)
oplus(AxB, BxC)
Pose.Pose4D
J_1oplus(AxB, BxC)
J_2oplus(AxB)
J_ominus(AxB)
ominus(AxB)
oplus(AxB, BxC)

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. (1.13)) with respect to the first pose:

$$J_{1\oplus} = \frac{\partial {}^A x_B \oplus {}^B x_C}{\partial {}^A x_B} = \begin{bmatrix} 1 & 0 & 0 & -{}^B x_C \sin({}^A\psi_B) - {}^B y_C \cos({}^A\psi_B) \\ 0 & 1 & 0 & {}^B x_C \cos({}^A\psi_B) - {}^B y_C \sin({}^A\psi_B) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.14}$$

Parameters

- **AxB** – first pose
- **BxC** – 2nd pose

Returns
$J_{1\oplus}$ Jacobian of the pose compounding operation with respect to the first pose (eq. (1.14))

`J_2oplus()`

Jacobian of the pose compounding operation ((1.13)) with respect to the second pose:

$$J_{2\oplus} = \frac{\partial {}^A x_B \oplus {}^B x_C}{\partial {}^B x_C} = \begin{bmatrix} \cos({}^A\psi_B) & -\sin({}^A\psi_B) & 0 & 0 \\ \sin({}^A\psi_B) & \cos({}^A\psi_B) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.15}$$

Parameters
AxB – first pose

Returns
$J_{2\oplus}$ Jacobian of the pose compounding operation with respect to the second pose (eq. (1.15))

`ominus()`

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

$${}^B x_A = \ominus {}^A x_B = \begin{bmatrix} -{}^A x_B \cos({}^A\psi_B) - {}^A y_B \sin({}^A\psi_B) \\ {}^A x_B \sin({}^A\psi_B) - {}^A y_B \cos({}^A\psi_B) \\ -{}^A z_B \\ -{}^A\psi_B \end{bmatrix} \tag{1.16}$$

Parameters
AxB – B-Frame pose expressed in A-Frame coordinates

Returns
A-Frame pose expressed in B-Frame coordinates (eq. (1.16)

`J_ominus()`

Jacobian of the inverse pose compounding operation ((1.13)) with respect the pose *AxB* (the self object):

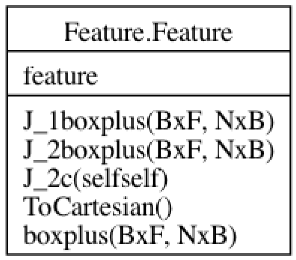
$$J_{\ominus} = \frac{\partial \ominus {}^A x_B}{\partial {}^A x_B} = \begin{bmatrix} -\cos({}^A\psi_B) & -\sin({}^A\psi_B) & 0 & {}^A x_B \sin({}^A\psi_B) - {}^A y_B \cos({}^A\psi_B) \\ \sin({}^A\psi_B) & -\cos({}^A\psi_B) & 0 & {}^A x_B \cos({}^A\psi_B) + {}^A y_B \sin({}^A\psi_B) \\ 0 & 0 & -1 & {}^A z_B \\ 0 & 0 & 0 & -1 \end{bmatrix} \tag{1.17}$$

Parameters
AxB – B-Fram pose expressed in A-Frame coordinates

Returns
$J_{\ominus}$ Jacobian of the inverse pose compounding operation with respect to the pose (eq. (1.17))

1.2 Feature Representation

1.2.1 Feature



class `Feature.`**`Feature`**(*feature*)

Bases: `object`

This class implements the **interface of the pose-feature compounding operation**. This class provides the interface to implement the compounding operation between the robot pose (represented in the N-Frame) and the feature pose (represented in the B-Frame) obtaining the feature representation in the N-Frame. The class also provides the interface to implement the Jacobians of the pose-feature compounding operation.

`__init__`(*feature*)

`boxplus`(*NxB*)

Pose-Feature compounding operation:

$$^{N}x_{F} =^{N} x_{B} \boxplus^{B} x_{F} \tag{1.18}$$

which computes the pose of a feature in the N-Frame given the pose of the robot in the N-Frame and the pose of the feature in the B-Frame. **This is a pure virtual method that must be overwritten by the child class**.

Parameters

- **`NxB`** – Robot pose in the N-Frame ($^{N}x_{B}$)
- **`BxF`** – Feature pose in the B-Frame ($^{B}x_{F}$)

Returns
Feature pose in the N-Frame ($^{N}x_{F}$)

`J_1boxplus`(*NxB*)

Jacobian of the Pose-Feature compounding operation (eq. (1.20)) with respect to the first argument $^{N}x_{B}$.

$$J_{1\boxplus} = \frac{\partial^{N}x_{B} \boxplus^{B} x_{F}}{\partial^{N}x_{B}}. \tag{1.19}$$

To be overriden by the child class.

Parameters

- **`NxB`** – Robot pose in the N-Frame ($^{N}x_{B}$)

- **BxF** – Feature pose in the B-Frame ($^B x_F$)

Returns
Jacobian matrix $J_{1\boxplus}$

`J_2boxplus(NxB)`

Jacobian of the Pose-Feature compounding operation (eq. (1.20)) with respect to the second argument $^B x_F$.

$$J_{2\boxplus} = \frac{\partial^N x_B \boxplus^B x_F}{\partial^B x_F}. \tag{1.20}$$

To be overriden by the child class.

Parameters
NxB – Robot pose in the N-Frame ($^N x_B$)

Returns
Jacobian matrix $J_{2\boxplus}$

`ToCartesian()`

Translates from its internal representation to the representation used for plotting. **To be overriden by the child class**.

Returns
Feature in Cartesian Coordinates

`J_2c()`

Jacobian of the ToCartesian method. Required for plotting non Cartesian features. **To be overriden by the child class**.

Returns
Jacobian of the transformation

1.2.2 Cartesian Feature

`class Feature.CartesianFeature(input_array)`

Bases: `Feature`, `ndarray`

Cartesian feature class. The class inherits from the `Feature` class providing an implementation of its interface for a Cartesian Feature, by implementing the $\boxplus$ operator as well as its Jacobians. The class also inherits from the ndarray numpy class allowing to be operated as a numpy ndarray.

`boxplus(NxB)`

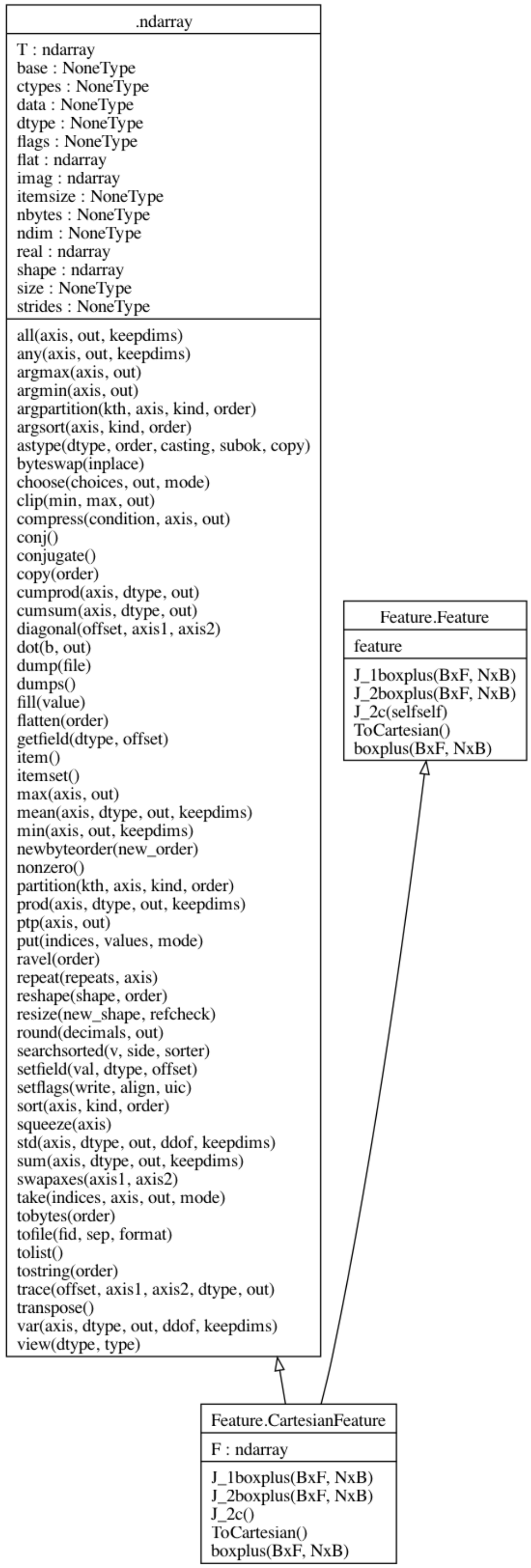
Pose-Cartesian Feature compounding operation:

$$\begin{gathered} F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \\ ^N x_F = ^N x_B \boxplus ^B x_F = F(^N x_B \oplus ^B x_F) \end{gathered} \tag{1.21}$$

which computes the Cartesian position of a feature in the N-Frame given the pose of the robot in the N-Frame and the Cartesian position of the feature in the B-Frame.

Parameters

- **NxB** – Robot pose in the N-Frame ($^N x_B$)
- **BxF** – Cartesian feature pose in the B-Frame ($^B x_F$)


.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)
Feature.Feature
feature
J_1boxplus(BxF, NxB)
J_2boxplus(BxF, NxB)
J_2c(selfself)
ToCartesian()
boxplus(BxF, NxB)
Feature.CartesianFeature
F : ndarray
J_1boxplus(BxF, NxB)
J_2boxplus(BxF, NxB)
J_2c()
ToCartesian()
boxplus(BxF, NxB)

Returns
Feature pose in the N-Frame ($^N x_F$)

`J_1boxplus(`*NxB*`)`

Jacobian of the Pose-Cartesian Feature compounding operation with respect to the robot pose:

$$J_{1\boxplus} = F J_{1\oplus} \tag{1.22}$$

Parameters

- **NxB** – robot pose represented in the N-Frame ($^N x_B$)
- **BxF** – Cartesian feature pose represented in the B-Frame ($^B x_F$)

Returns
Jacobian matrix $J_{boxplus}$ (eq. (1.22)) (eq. (1.22))

`J_2boxplus(`*NxB*`)`

Jacobian of the Pose-Cartesian Feature compounding operation with respect to the feature position:

$$J_{2\boxplus} = F J_{2oplus} \tag{1.23}$$

Parameters

- **NxB** – robot pose represented in the N-Frame ($^N x_B$)
- **BxF** – Cartesian feature pose represented in the B-Frame ($^B x_F$)

Returns
Jacobian matrix $J_{1\boxplus}$ (eq. (1.23))

`ToCartesian()`

Translates from its internal representation to the representation used for plotting.

Returns
Feature in Cartesian Coordinates

`J_2c()`

Jacobian of the ToCartesian method. Required for plotting non Cartesian features. **To be overriden by the child class**.

Returns
Jacobian of the transformation

1.2.3 Polar Feature

class `Feature.PolarFeature(`*input_array*, **args*`)`

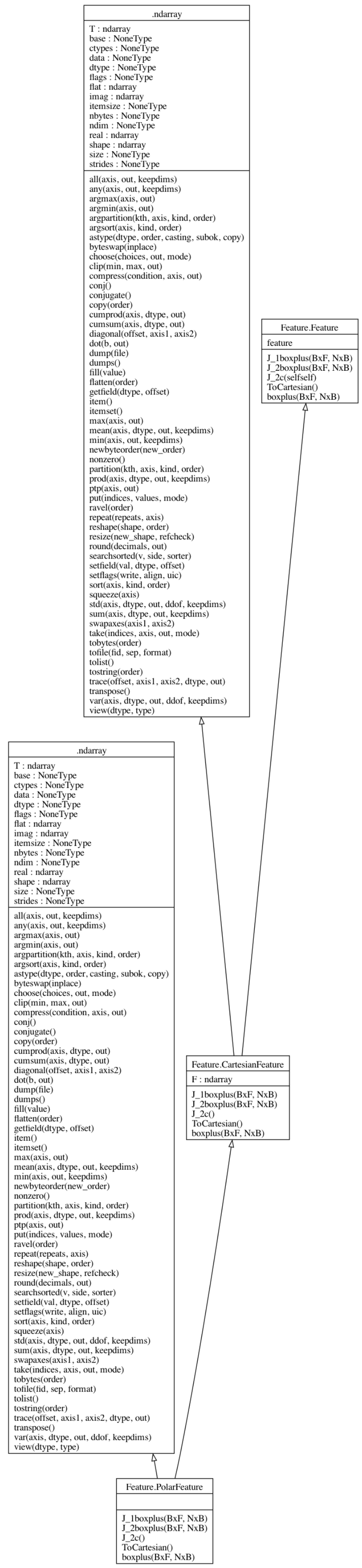
Bases: `CartesianFeature`, `ndarray`

Polar feature class. The class inherits from the `Feature` class providing an implementation of its interface for a Polar Feature, by implementing the ⊞ operator as well as its Jacobians. The class also inherits from the ndarray numpy class allowing to be operated as a numpy ndarray.

`boxplus(`*NxB*`)`

Pose-Polar Feature compounding operation:

$$\begin{gathered} F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \\ {}^N x_F = {}^N x_B \boxplus {}^B x_F = F({}^N x_B \oplus {}^B x_F) \end{gathered} \tag{1.24}$$


.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)
Feature.Feature
feature
J_1boxplus(BxF, NxB)
J_2boxplus(BxF, NxB)
J_2c(selfself)
ToCartesian()
boxplus(BxF, NxB)
.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)
Feature.CartesianFeature
F : ndarray
J_1boxplus(BxF, NxB)
J_2boxplus(BxF, NxB)
J_2c()
ToCartesian()
boxplus(BxF, NxB)
Feature.PolarFeature
J_1boxplus(BxF, NxB)
J_2boxplus(BxF, NxB)
J_2c()
ToCartesian()
boxplus(BxF, NxB)

which computes the Polar pose of a feature in the N-Frame given the pose of the robot in the N-Frame and the Polar pose of the feature in the B-Frame.

Parameters

- **NxB** – Robot pose in the N-Frame ($^N x_B$)
- **BxF** – Polar feature pose in the B-Frame ($^B x_F$)

Returns

Polar feature pose in the N-Frame ($^N x_F$)

`J_1boxplus(NxB)`

Jacobian of the Pose-Polar Feature compounding operation with respect to the robot pose:

$$J_{1\boxplus} = FJ_{1\oplus} \tag{1.25}$$

Parameters

- **NxB** – robot pose represented in the N-Frame ($^N x_B$)
- **BxF** – Polar eature pose represented in the B-Frame ($^B x_F$)

Returns

Jacobian matrix $J_{boxplus}$ (eq. (1.25))

`J_2boxplus(NxB)`

Jacobian of the Pose-Polar Feature compounding operation with respect to the feature pose:

$$J_{2\boxplus} = FJ_{2oplus} \tag{1.26}$$

Parameters

- **NxB** – Robot pose represented in the N-Frame ($^N x_B$)
- **BxF** – Polar feature pose represented in the B-Frame ($^B x_F$)

Returns

Jacobian matrix $J_{1\boxplus}$ (eq. (1.26))

`ToCartesian()`

Translates from its internal representation to the representation used for plotting.

Returns

Feature in Cartesian Coordinates

`J_2c()`

Jacobian of the ToCartesian method. Required for plotting non Cartesian features. **To be overriden by the child class**.

Returns

Jacobian of the transformation

1.3 Coordinate Conversion Functions

This functions are used to convert between different coordinate systems (e.g. from cartesian to polar, from polar to cartesian, etc.). The functions are implemented in a way that they can be used with numpy arrays. or each coordinate conversion its Jacobian is also implemented. This functions are required to convert from the observation space (the coordinates in which the features are observed) to the storage space (the coordinates in which the features are stored in the map). For instance, if the features are observed in polar coordinates, but stored in cartesian coordinates, the conversion from polar to cartesian is required.

1.3.1 Polar To Cartesian

`conversions.p2c(p)`

Converts from a 2D Polar coordinate to its corresponding 2D Cartesian coordinate:

$$
\begin{aligned}
p &= \begin{bmatrix} \rho \\ \theta \end{bmatrix} \\
c &= p2c\left(\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \rho\cos(\theta) \\ \rho\sin(\theta) \end{bmatrix}\right)
\end{aligned}
\tag{1.27}
$$

Parameters
p – point in polar coordinates

Returns
point in cartesian coordinates

`conversions.J_p2c(p)`

Jacobian of the 2D Polar to cartesian conversion:

$$
J_{p2c} = \begin{bmatrix} \frac{\partial x}{\partial \rho} & \frac{\partial x}{\partial \theta} \\ \frac{\partial y}{\partial \rho} & \frac{\partial y}{\partial \theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\rho\sin(\theta) \\ \sin(\theta) & \rho\cos(\theta) \end{bmatrix}
\tag{1.28}
$$

Parameters
p – linearization point in polar coordinates

Returns
Jacobian matrix J_{p2c} (eq. (1.28))

1.3.2 Cartesian To Polar

`conversions.c2p(c)`

2D Cartesian to polar conversion:

$$
\begin{aligned}
c &= \begin{bmatrix} x \\ y \end{bmatrix} \\
p &= c2p\left(\begin{bmatrix} \rho \\ \theta \end{bmatrix} = \begin{bmatrix} \sqrt{x^2 + y^2} \\ atan2(y, x) \end{bmatrix}\right)
\end{aligned}
\tag{1.29}
$$

Parameters
c – point in cartesian coordinates

Returns
point in polar coordinates

`conversions.J_c2p(c)`

Jacobian of the 2D Cartesian to polar conversion:

$$J_{c2p} = \begin{bmatrix} \frac{\partial \rho}{\partial x} & \frac{\partial \rho}{\partial y} \\ \frac{\partial \theta}{\partial x} & \frac{\partial \theta}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} \\ -\frac{y}{x^2+y^2} & \frac{x}{x^2+y^2} \end{bmatrix} \tag{1.30}$$

Parameters

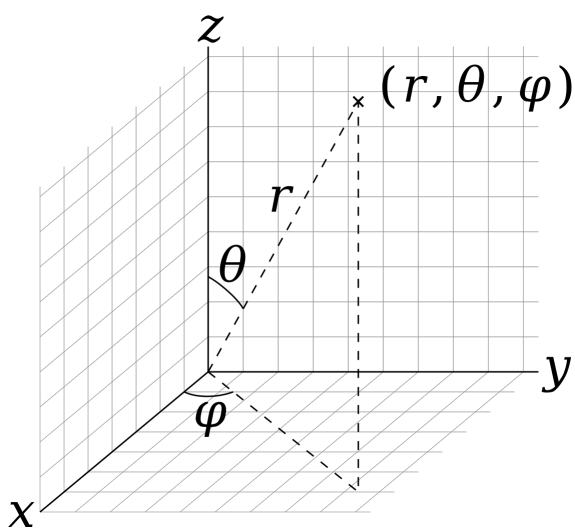
c – point in cartesian coordinates

Returns

Jacobian matrix J_{c2p} (eq. (1.30))

1.3.3 Spherical To Cartesian

`conversions.s2c(s)`



3D Spherical to cartesian conversion:

$$\begin{aligned} s &= \begin{bmatrix} \rho \\ \theta \\ \varphi \end{bmatrix} \\ c &= s2c\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \rho \sin(\theta)\cos(\varphi) \\ \rho \sin(\theta)\sin(\varphi) \\ \rho\cos(\theta) \end{bmatrix}\right) \end{aligned} \tag{1.31}$$

Parameters

s – point in spherical coordinates

Returns

point in cartesian coordinates

`conversions.J_s2c(s)`

Jacobian of the 3D Spherical to cartesian conversion:

$$J_{s2c} = \begin{bmatrix} \frac{\partial x}{\partial \rho} & \frac{\partial x}{\partial \theta} & \frac{\partial x}{\partial \varphi} \\ \frac{\partial y}{\partial \rho} & \frac{\partial y}{\partial \theta} & \frac{\partial y}{\partial \varphi} \\ \frac{\partial z}{\partial \rho} & \frac{\partial z}{\partial \theta} & \frac{\partial z}{\partial \varphi} \end{bmatrix} = \begin{bmatrix} \sin(\theta)\cos(\varphi) & \rho\cos(\theta)\cos(\varphi) & -\rho\sin(\theta)\sin(\varphi) \\ \sin(\theta)\sin(\varphi) & \rho\cos(\theta)\sin(\varphi) & \rho\sin(\theta)\cos(\varphi) \\ \cos(\theta) & -\rho\sin(\theta) & 0 \end{bmatrix} \tag{1.32}$$

Parameters
s – linearization point in spherical coordinates

Returns
Jacobian matrix J_{s2c} (eq. (1.32))

1.3.4 Cartesian To Spherical

`conversions.c2s(c)`

3D Cartesian to spherical conversion:

$$c = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$s = c2s\left(\begin{bmatrix} \rho \\ \theta \\ \varphi \end{bmatrix}\right) = \begin{bmatrix} \sqrt{x^2+y^2+z^2} \\ atan2(\sqrt{x^2+y^2}, z) \\ atan2(y, x) \end{bmatrix} \tag{1.33}$$

Parameters
c – point in cartesian coordinates

Returns
point in spherical coordinates

`conversions.J_c2s(c)`

Jacobian of the 3D Cartesian to spherical conversion:

$$J_{c2s} = \begin{bmatrix} \frac{\partial\rho}{\partial x} & \frac{\partial\rho}{\partial y} & \frac{\partial\rho}{\partial z} \\ \frac{\partial\theta}{\partial x} & \frac{\partial\theta}{\partial y} & \frac{\partial\theta}{\partial z} \\ \frac{\partial\varphi}{\partial x} & \frac{\partial\varphi}{\partial y} & \frac{\partial\varphi}{\partial z} \end{bmatrix} = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2+z^2}} & \frac{y}{\sqrt{x^2+y^2+z^2}} & \frac{z}{\sqrt{x^2+y^2+z^2}} \\ \frac{y}{x^2+y^2} & \frac{x}{x^2+y^2} & 0 \\ \frac{-xz}{(x^2+y^2)\sqrt{x^2+y^2}} & \frac{-yz}{(x^2+y^2)\sqrt{x^2+y^2}} & \frac{\sqrt{x^2+y^2}}{x^2+y^2} \end{bmatrix} \tag{1.34}$$

Parameters
c – linearization point in cartesian coordinates

Returns
Jacobian matrix J_{c2s} (eq. (1.34))

1.3.5 Identity Conversion

`conversions.v2v(v)`

Identity transformation. Returns the same vector.

Parameters
v – input vector

Returns
output vector

`conversions.J_v2v(v)`

Jacobian of the identity transformation. Returns the identity matrix of the same dimensionality as the input vector.

Parameters
v – input vector

Returns
Identity matrix of the same dimensionality as the input vector.

1.4 Robot Simulation

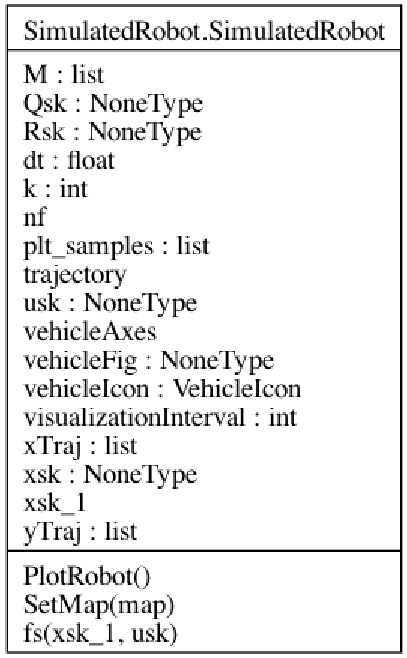


Fig. 1: SimulatedRobot Class Diagram.

`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*, *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

1.4.1 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.

$$Q_k = \begin{bmatrix} \sigma_{\dot{u}}^2 & 0 & 0 \\ 0 & \sigma_{\dot{v}}^2 & 0 \\ 0 & 0 & \sigma_{\dot{r}}^2 \end{bmatrix} \tag{1.35}$$

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

$$\mathbf{u_k} = \begin{bmatrix} u_k & r_k \end{bmatrix}^{\mathbf{T}} \tag{1.36}$$

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

$$x_k = \begin{bmatrix} ^N x_k & ^N y_k & ^N \theta_k & ^B u_k & ^B v_k & ^B r_k \end{bmatrix}^T \tag{1.37}$$

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.

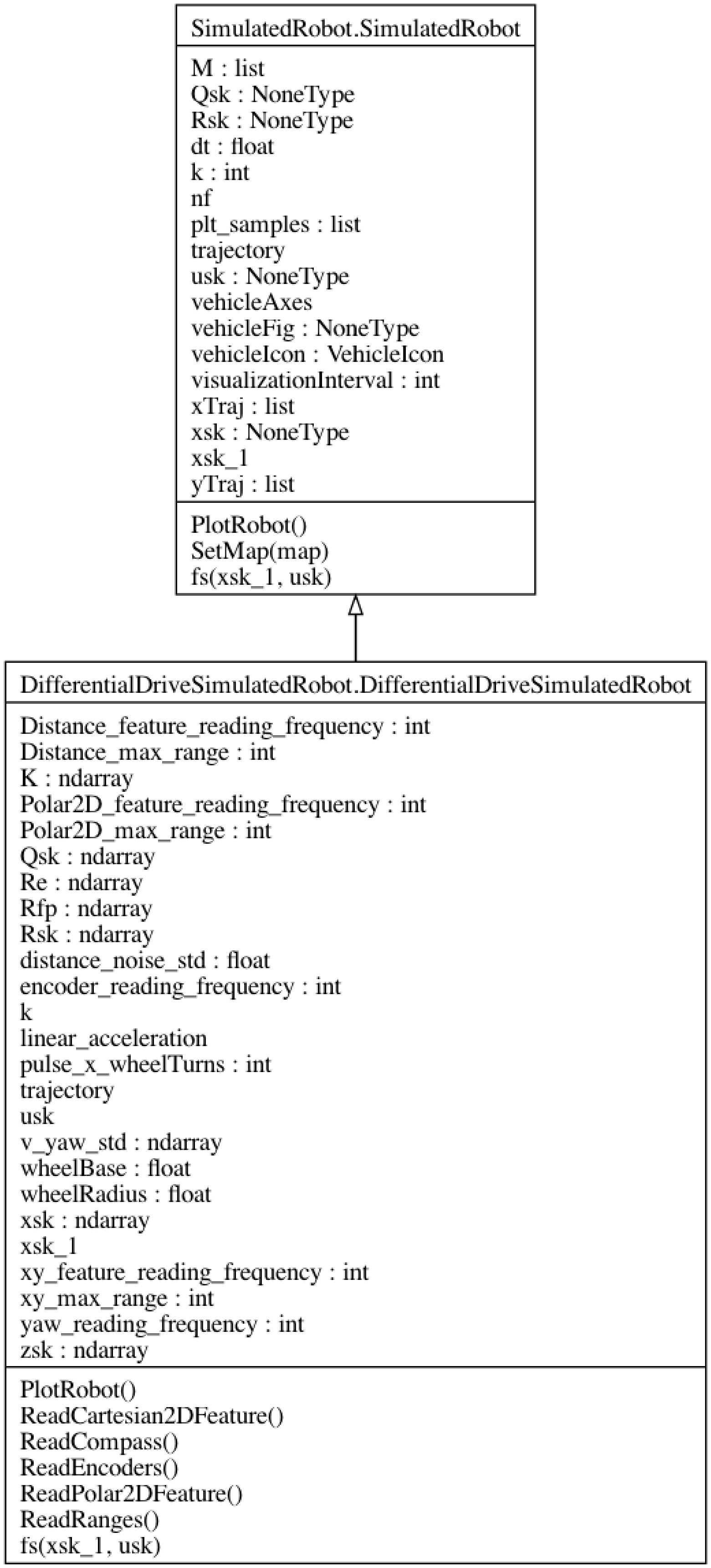


Fig. 2: DifferentialDriveSimulatedRobot Class Diagram.

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:

$$\begin{aligned}
\eta_{s_{k-1}} &= \begin{bmatrix} x_{s_{k-1}} & y_{s_{k-1}} & \theta_{s_{k-1}} \end{bmatrix}^T \\
\nu_{s_{k-1}} &= \begin{bmatrix} u_{s_{k-1}} & v_{s_{k-1}} & r_{s_{k-1}} \end{bmatrix}^T \\
x_{s_{k-1}} &= \begin{bmatrix} \eta^T_{s_{k-1}} & \nu^T_{s_{k-1}} \end{bmatrix}^T \\
u_{s_k} &= \nu_d = \begin{bmatrix} u_d & r_d \end{bmatrix}^T \\
w_{s_k} &= \dot{\nu}_{s_k} \\
x_{s_k} &= f_s(x_{s_{k-1}}, u_{s_k}, w_{s_k}) \\
&= \begin{bmatrix} \eta_{s_{k-1}} \oplus (\nu_{s_{k-1}}\Delta t + \frac{1}{2} w_{s_k} \Delta t^2) \\ \nu_{s_{k-1}} + K(\nu_d - \nu_{s_{k-1}}) + w_{s_k}\Delta t \end{bmatrix} \quad ; \quad K = diag(k_1, k_2, k_3) \quad k_i > 0
\end{aligned} \tag{1.38}$$

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

$zk = [\Delta n_L \ \Delta n_R]^T$ 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.

`ReadPolar2DFeature()`

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

Returns

zsk: [[x1 y1 z1],...,[xn yn zn]]
Cartesian position of the feature observations.

Rsk: block_diag(R_1,...,R_n), where R_i=[[r_xx r_xy r_xz],[r_xy r_yy r_yz],[r_xz r_yz r_yy]] is the
2x2 i-th feature observation covariance. Covariance of the Polar feature observations. Note the features are uncorrelated among them.

`ReadRanges()`

Simulates the reading of distance towards 2D Cartesian 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

1.4.2 4 DOF AUV Robot Simulation

`class AUV4DOFSimulatedRobot.AUV4DOFSimulatedRobot(xs0, map=[], *args)`

Bases: `SimulatedRobot`

This class simulates an AUV equipped with the following sensors: Gyro, Compass, DVL, Depth, direct USBL and inverted USBL.

`dt = 0.1`

class attribute containing sample time of the simulation

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

Constructor.

Parameters

- **xs0** – initial simulated robot pose $x_{s_k} = [x_{s_k}, y_{s_k}, z_{s_k}, \psi_{s_k}]$ in the N-Frame
- **map** – map of the environment

`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

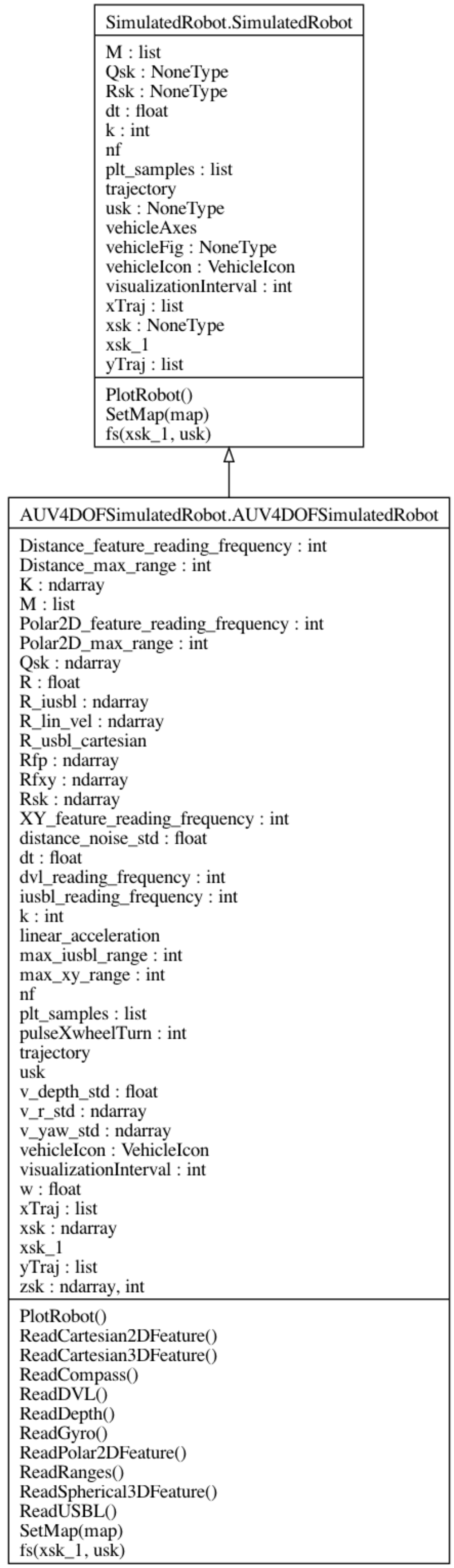


Fig. 3: AUV4DOFSimulatedRobot Class Diagram.

robot state x_{k-1} and the input u_k:

$$
\begin{aligned}
\eta_{s_{k-1}} &= \begin{bmatrix} x_{s_{k-1}} & y_{s_{k-1}} & z_{s_{k-1}} & \psi_{s_{k-1}} \end{bmatrix}^T \\
\nu_{s_{k-1}} &= \begin{bmatrix} u_{s_{k-1}} & v_{s_{k-1}} & w_{s_{k-1}} & r_{s_{k-1}} \end{bmatrix}^T \\
x_{s_{k-1}} &= \begin{bmatrix} \eta^T_{s_{k-1}} & \nu^T_{s_{k-1}} \end{bmatrix}^T \\
u_{s_k} &= \nu_d = \begin{bmatrix} u_d & v_d & w_d & r_d \end{bmatrix}^T \\
w_{s_k} &= \dot{\nu}_{s_k} \\
x_{s_k} &= f_s(x_{s_{k-1}}, u_{s_k}, w_{s_k}) \\
&= \begin{bmatrix} \eta_{s_{k-1}} \oplus (\nu_{s_{k-1}} \Delta t + \frac{1}{2} w_{s_k}) \\ \nu_{s_{k-1}} + K(\nu_d - \nu_{s_{k-1}}) + w_{s_k} \Delta t \end{bmatrix} \quad ; \quad K = diag(k_1, k_2, k_3, k_4) \quad k_i > 0
\end{aligned}
\tag{1.39}
$$

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}

`SetMap(map)`

Initializes the map of the environment.

`ReadGyro()`

Simulates the gyro reading of the robot.

Returns

angular velocity [r] and the covariance of the noise [R_r]

`ReadCompass()`

Simulates the compass reading of the robot.

Returns

yaw and the covariance of its noise *R_yaw*

`ReadDVL()`

Simulates the DVL reading of the robot.

Returns

linear velocity [u v w] and the covariance of the noise [R_lin_vel]

`ReadDepth()`

Simulates the depth reading of the robot.

Returns
depth and the covariance of the noise [R_depth]

`ReadUSBL()`

Simulates the USBL reading of the robot. Assumes that the USBL transceiver is placed at the origin of the N_Frame and the transceiver is placed at the origin of the robot's B-Frame.

Returns
zsk: cartesian position [x y z] Rsk: Covariance of the noise [R_usbl_cartesian]

`ReadSpherical3DFeature()`

Simulates the inverted USBL sensor. In this case the transceiver is mounted at the origin of the robot's B-Frame and n transponders are layed out on the seafloor. The Cartesian position of the transponders is stored in the map in Cartesian coordinates.

Returns
zsk=[[r_0 theta_0 varphi_0],…,[r_i theta_i varphi_i],…[r_n theta_n varphi_n]], Rsk=block_diag(R_0,…,R_i,…,R_n)

where [r_i theta_i phi_i] is the spherical position of the i transponder, being, r_i the distance, theta_i the elevation angle and varphi_i the azimuth angle. The covariance is a block diagonal matrix (since the transponder positions are uncorrelated) where each block is a 3x3 matrix corresponding to the covariance of the noise of each transponder within the range of the sensor.

`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.

`ReadPolar2DFeature()`

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

Returns

zsk: [[x1 y1 z1],…,[xn yn zn]]
Cartesian position of the feature observations.

Rsk: block_diag(R_1,…,R_n), where R_i=[[r_xx r_xy r_xz],[r_xy r_yy r_yz],[r_xz r_yz r_yy]] is the
2x2 i-th feature observation covariance. Covariance of the Polar feature observations. Note the features are uncorrelated among them.

`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.

`ReadCartesian3DFeature()`

Simulates the reading of 3D cartesian features. The features are placed in the map in cartesian coordinates.

Returns

zsk: [[x1 y1 z1],…,[xn yn zn]]
Cartesian position of the feature observations.

Rsk: block_diag(R_1,…,R_n), where R_i=[[r_xx r_xy r_xz],[r_xy r_yy r_yz],[r_xz r_yz r_yy]] is the
3x3 i-th feature observation covariance. Covariance of the Cartesian feature observations. Note the features are uncorrelated among them.

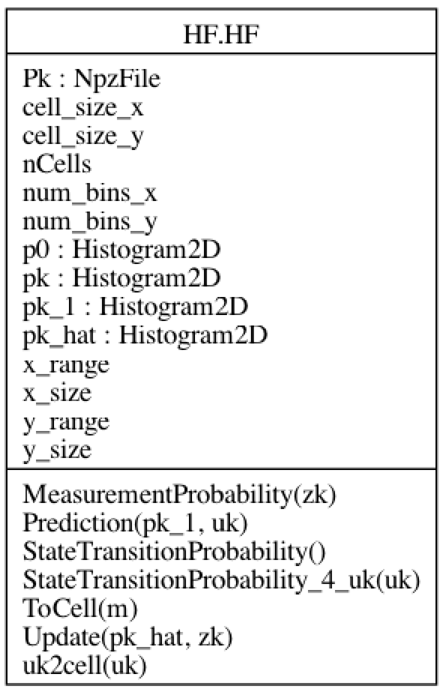
`_PlotSample`(*x*, *P*, *n*)

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

1.5 Filters

1.5.1 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*. It creates the following attributes:

- self.p0: the initial belief histogram
- self.pk_1: the previous belief histogram, initially initialized from *p0*
- self.pk_hat: the prior belief histogram, after applying the Total Probability theorem

- self.pk: the posterior belief histogram, after applying the Bayes Rule

that will be updated during the execution of the filter. This method also initializes the state transition probability matrix *self.Pk*. If the file *StateTransitionProbability.npy* exists, the matrix will be loaded from it. Otherwise, it is computed by the derived class through the pure virtual method *StateTransitionProbability(·)* and stored in the file for posterior uses. This is done to avoid recomputing the matrix every since this is a time-consuming operation. The state transition probability matrix is used in the `Prediction()` step and the measurement probability matrix is used in the `Update()` one.

Parameters
p0 – initial probability histogram

`ToCell(m)`

Converts a metric value to a cell displacement.

Parameters
m – value in meters

Returns
value 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.

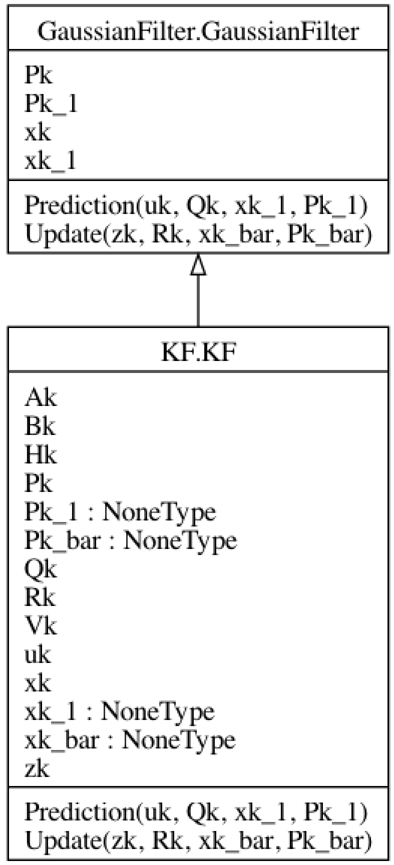
1.5.2 Particle Filter

Particle Filter

To be completed…

1.5.3 Kalman Filter

Kalman Filter



class `KF.KF`(*Ak*, *Bk*, *Hk*, *Vk*, *x0*, *P0*, **args*)

Bases: `GaussianFilter`

Kalman Filter class. Implements the `GaussianFilter` interface for the particular case of the Kalman Filter.

`__init__`(*Ak*, *Bk*, *Hk*, *Vk*, *x0*, *P0*, **args*)

Constructor of the KF class.

Parameters

- **Ak** – Transition matrix of the motion model
- **Bk** – Input matrix of the motion model
- **Hk** – Observation matrix of the observation model
- **Vk** – Noise projection matrix of the motion model
- **x0** – initial mean of the state vector
- **P0** – initial covariance matrix
- **args** – arguments to be passed to the parent class

`Prediction`(*uk*, *Qk*, *xk_1=None*, *Pk_1=None*)

Prediction step of the Kalman Filter.

Parameters

- **uk** – input vector
- **Qk** – covariance matrix of the motion model noise
- **xk_1** – previous mean state vector
- **Pk_1** – previous covariance matrix

Return xk_bar, Pk_bar
current mean state vector and covariance matrix

Update(*zk*, *Rk*, *xk_bar=None*, *Pk_bar=None*)

Update step of the Kalman Filter.

Parameters

- **zk** – observation vector
- **Rk** – covariance of the observation model noise
- **xk_bar** – predicted mean state vector
- **Pk_bar** – predicted covariance matrix

Return xk.Pk
current mean state vector and covariance matrix

1.5.4 Extended Kalman Filter

Extended Kalman Filter

class EKF.**EKF**(*x0*, *P0*, **args*)

Bases: `GaussianFilter`

Extended Kalman Filter class. Implements the `GaussianFilter` interface for the particular case of the Extended Kalman Filter.

__init__(*x0*, *P0*, **args*)

Constructor of the EKF class.

Parameters

- **x0** – initial mean state vector
- **P0** – initial covariance matrix
- **args** – arguments to be passed to the parent class

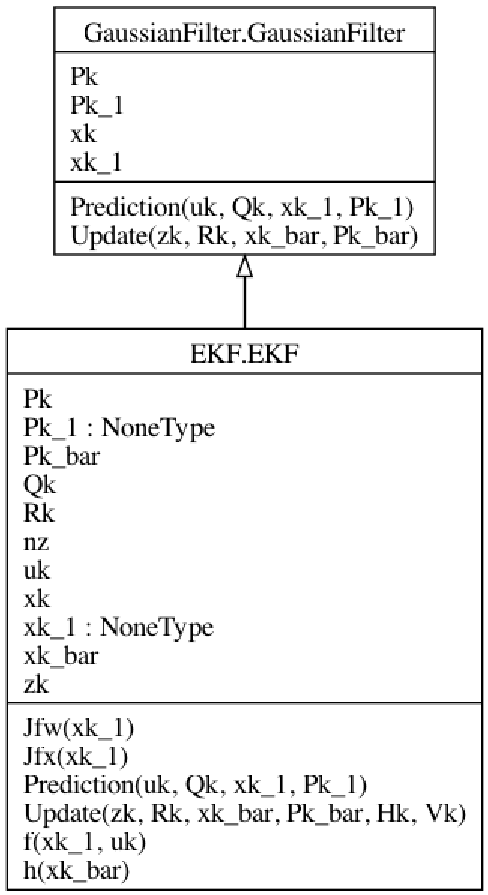
f(*xk_1=None*, *uk=None*)

” Motion model of the EKF to be overwritten by the child class.

Parameters

- **xk_1** – previous mean state vector
- **uk** – input vector

Return xk_bar, Pk_bar
predicted mean state vector and its covariance matrix


GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
EKF.EKF
Pk
Pk_1 : NoneType
Pk_bar
Qk
Rk
nz
uk
xk
xk_1 : NoneType
xk_bar
zk
Jfw(xk_1)
Jfx(xk_1)
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar, Hk, Vk)
f(xk_1, uk)
h(xk_bar)

Jfx(*xk_1=None*)

Jacobian of the motion model with respect to the state vector. *Method to be overwritten by the child class.*

Parameters
xk_1 – Linearization point. By default the linearization point is the previous state vector taken from a class attribute.

Returns
Jacobian matrix

Jfw(*xk_1=None*)

Jacobian of the motion model with respect to the noise vector. *Method to be overwritten by the child class.*

Parameters
xk_1 – Linearization point. By default the linearization point is the previous state vector taken from a class attribute.

Returns
Jacobian matrix

h(*xk_bar=None*)

Observation model of the EKF. We differentiate two types of observations: 1. **Measurements**: observations that are directly measured by the sensors. For example, the position of the robot, its heading, its speed, etc. 2. **Features**: observations of map features. For example, the position of a landmark.

This method calls the *EKF.hm()* which implements the measurements observation equation. To implement a standard EKF, the *EKF.hm* method should be overwritten by the child class to be used as the observation equation for measurements.

Parameters
xk_bar – mean of the predicted state vector. By default it is taken from the class attribute.

Returns
expected observation vector

hm(*xk_bar=None*)

Observation model related to the measurements observations of the EKF to be overwritten by the child class.

Parameters
xk_bar – mean of the predicted state vector. By default it is taken from the class attribute.

Returns
expected observation vector

Prediction(*uk*, *Qk*, *xk_1=None*, *Pk_1=None*)

Prediction step of the EKF. It calls the motion model and its Jacobians to predict the state vector and its covariance matrix.

Parameters

- **uk** – input vector
- **Qk** – covariance matrix of the noise vector
- **xk_1** – previous mean state vector. By default it is taken from the class attribute. Otherwise it updates the class attribute.
- **Pk_1** – covariance matrix of the previous state vector. By default it is taken from the class attribute. Otherwise it updates the class attribute.

Return xk_bar, Pk_bar
predicted mean state vector and its covariance matrix. Also updated in the class attributes.

Update(*zk*, *Rk*, *xk_bar*, *Pk_bar*, *Hk*, *Vk*)

Update step of the EKF. It calls the observation model and its Jacobians to update the state vector and its covariance matrix.

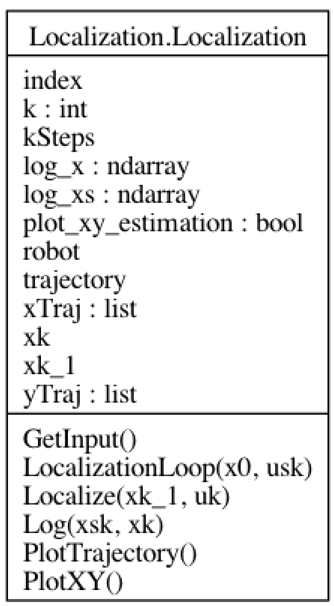
Parameters

- **zk** – observation vector
- **Rk** – covariance matrix of the noise vector
- **xk_bar** – predicted mean state vector. By default it is taken from the class attribute. Otherwise it updates the class attribute.
- **Pk_bar** – covariance matrix of the predicted state vector. By default it is taken from the class attribute. Otherwise it updates the class attribute.

Return xk,Pk
updated mean state vector and its covariance matrix. Also updated in the class attributes.

1.6 Localization

1.6.1 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 (`Index`)
- **kSteps** – Number of time steps to simulate
- **robot** – Simulation robot object (`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.

Returns

- **uk:** input variable

`Localize(xk_1, uk)`

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

Parameters
xk_1 – previous robot pose

Returns
xk current robot pose

`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

`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.

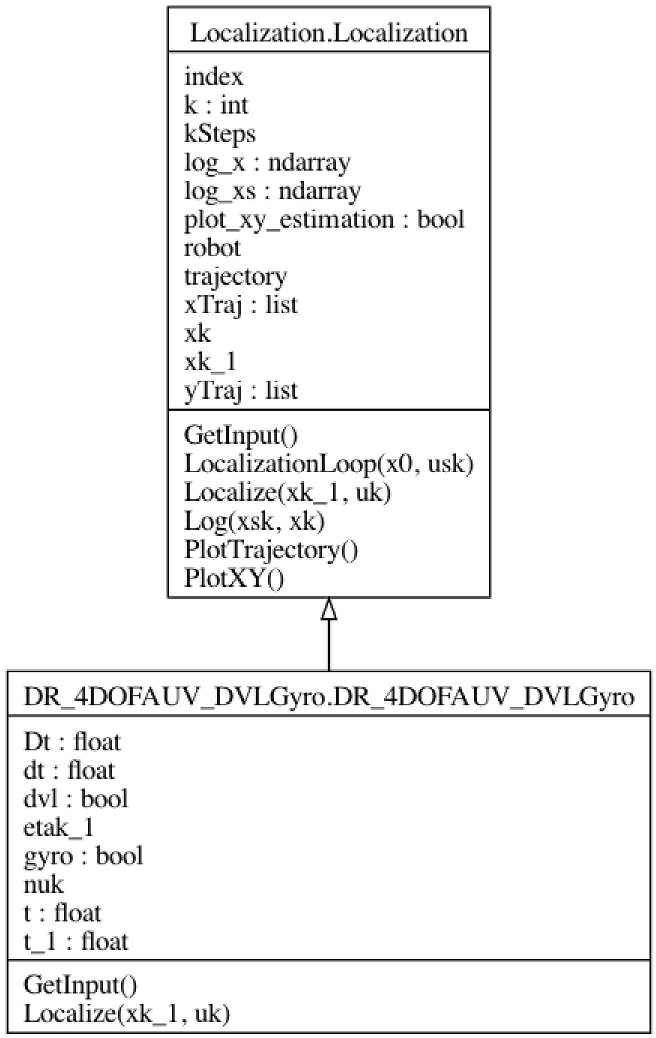
1.6.2 Dead Reckoning

4 DOF AUV Dead REckoning using DVL and Gyro

`class DR_4DOFAUV_DVLGyro.DR_4DOFAUV_DVLGyro(index, kSteps, robot, x0, *args)`

Bases: `Localization`

Dead Reckoning Localization for a 4DOF AUV with DVL and Gyro sensors


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_4DOFAUV_DVLGyro.DR_4DOFAUV_DVLGyro
Dt : float
dt : float
dvl : bool
etak_1
gyro : bool
nuk
t : float
t_1 : float
GetInput()
Localize(xk_1, uk)

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

Constructor of the DR_4DOFAUV_DVLGyro class.

Parameters

args – Rest of arguments to be passed to the parent constructor

Localize(*xk_1*, *uk*)

Motion model for the 4DOF ($[x_k \; y_k \; z_k \; \psi_k]^T$) AUV using as input the lineal velocity read from the DVL senosr and angular velocity read from the Gyro sensor

Parameters

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

Return xk

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

GetInput()

Gets the input vector and the input noise covariance matrix from the robot. The input vector contains the linear read from the DVL and angular velocity read from the Gyro of the robot.

$$
\begin{aligned}
u_k &= \begin{bmatrix} \nu_{DVL}^T & r_{Gyro}^T \end{bmatrix}^T = \begin{bmatrix} u_{DVL} & v_{DVL} & w_{DVL} & r_{Gyro} \end{bmatrix}^T \\
Q_k &= \begin{bmatrix} Q_{DVL} & 0 \\ 0 & \sigma_{Gyro}^2 \end{bmatrix} \quad \text{where} \quad Q_{DVL} = \begin{bmatrix} \sigma_{u_{DVL}}^2 & \sigma_{uv_{DVL}} & \sigma_{uw_{DVL}} \\ \sigma_{vu_{DVL}} & \sigma_{v_{DVL}}^2 & \sigma_{vw_{DVL}} \\ \sigma_{wu_{DVL}} & \sigma_{wv_{DVL}} & \sigma_{w_{DVL}}^2 \end{bmatrix}
\end{aligned}
\tag{1.40}
$$

Returns

input vector u_k and input noise covariance matrix Q_k defined in eq. (1.40).

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 `DR_3DOFDifferentialDrive` class.

Parameters

args – Rest of arguments to be passed to the parent constructor

Localize(*xk_1*, *uk*)

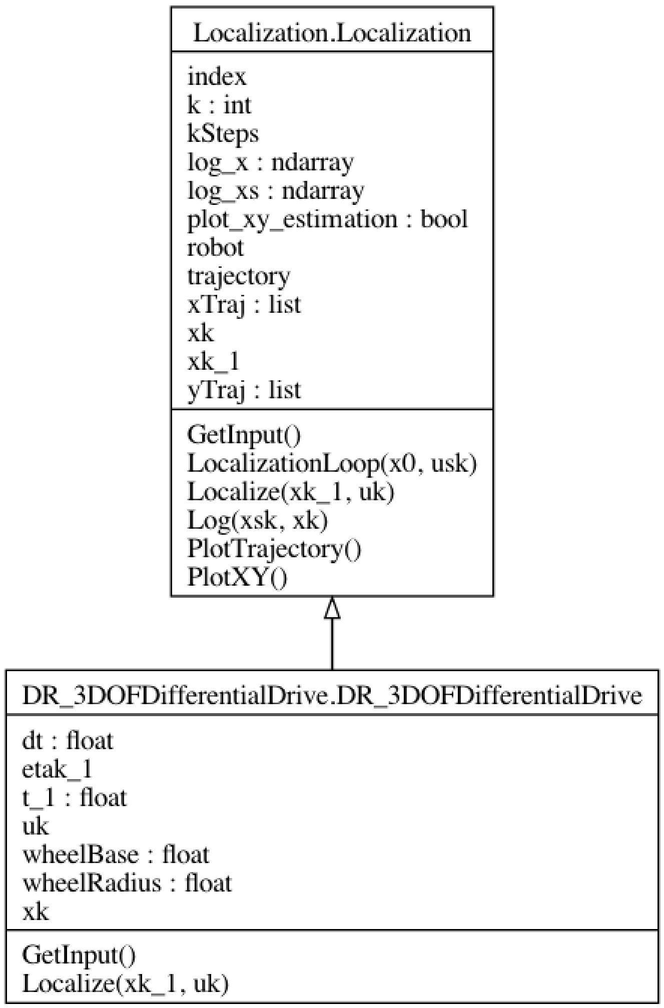
Motion model for the 3DOF ($x_k = [x_k \; y_k \; \psi_k]^T$) Differential Drive Mobile robot using as input the readings of the wheel encoders ($u_k = [n_L \; n_R]^T$).

Parameters

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

Returns

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


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
etak_1
t_1 : float
uk
wheelBase : float
wheelRadius : float
xk
GetInput()
Localize(xk_1, uk)

GetInput()

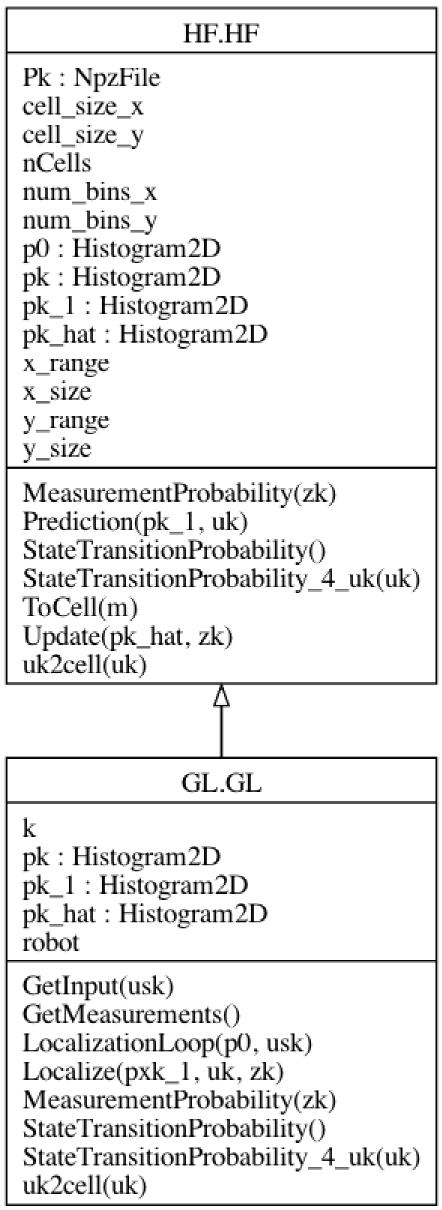
Get the input for the motion model. In this case, the input is the robot displacement computed from the left and right wheel encoders pulses using.

Returns

- **uk:** input vector ($u_k = {}^B[\Delta x \; \Delta y]^T$)

1.6.3 Grid Localization

Grid Localization



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

Bases: *HF*

Grid Localization.

`__init__`(*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.

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. This is a pure virtual method that must be implemented by the derived class.

`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

`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. 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

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

`uk2cell`(*uk*)

Parameters
uk –

Returns

`LocalizationLoop`(*p0*, *usk*)

Given an initial robot pose x_0 and the input to the `AUV4DOFSimulatedRobot.SimulatedRobot` this method calls iteratively `GL_4DOFAUV.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*)

Overrides the parent method `DR_4DOFAUV_DVLGyro.Localize()`.

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

4 DOF AUV

AUV Grid Localization Example

class `GL_4DOFAUV.GL_4DOFAUV`(*dx_max*, *dy_max*, *range_dx*, *range_dy*, *p0*, *index*, *kSteps*, *robot*, *x0*, **args*)

Bases: *GL*, *DR_4DOFAUV_DVLGyro*

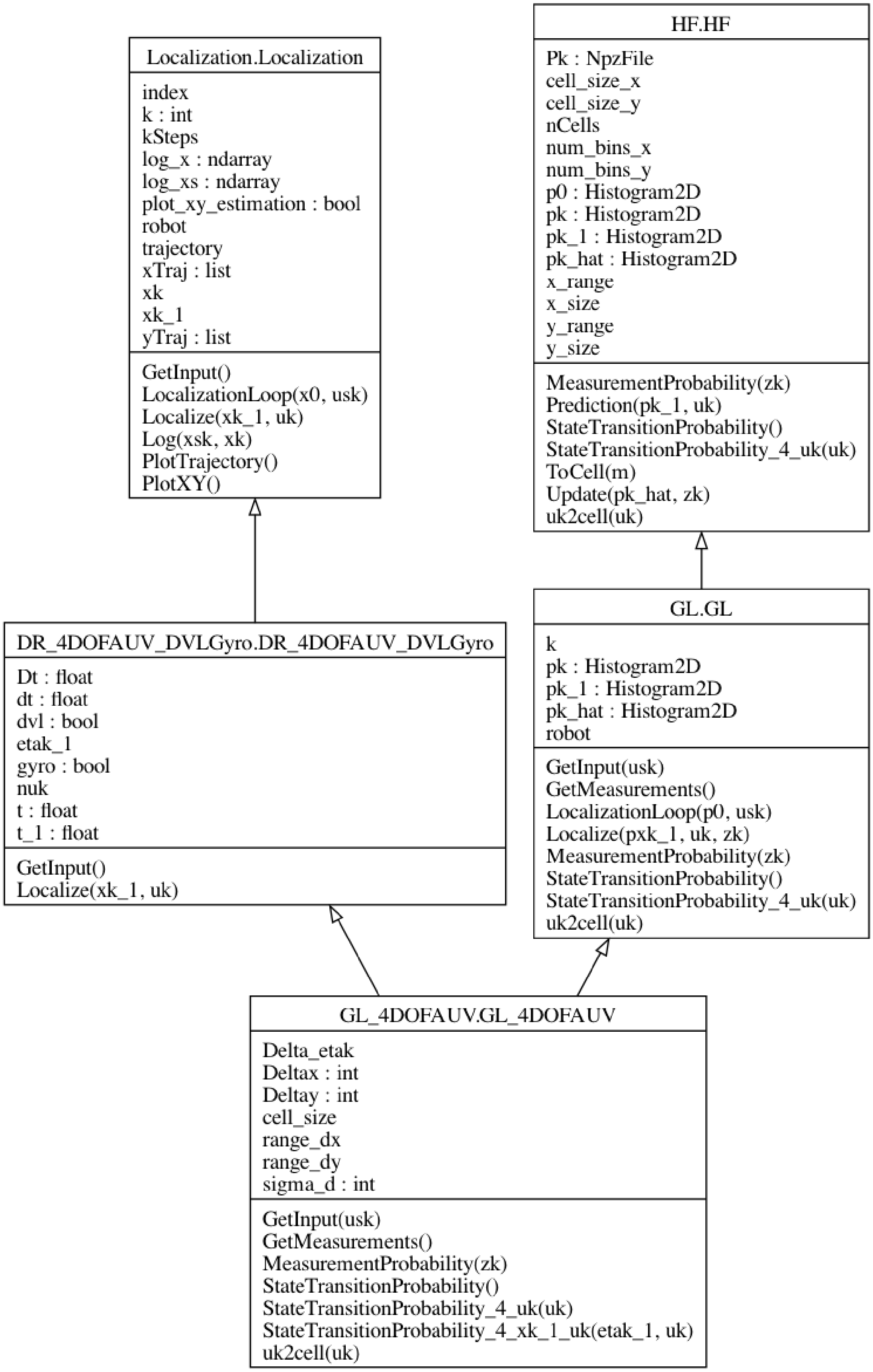
Grid Reckoning Localization for a 4 DOF AUV.

`__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.

Parameters

- **dx_max** – maximum x displacement in meters
- **dy_max** – maximum y displacement in meters
- **range_dx** – range of x displacements in meters


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()
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
MeasurementProbability(zk)
Prediction(pk_1, uk)
StateTransitionProbability()
StateTransitionProbability_4_uk(uk)
ToCell(m)
Update(pk_hat, zk)
uk2cell(uk)
DR_4DOFAUV_DVLGyro.DR_4DOFAUV_DVLGyro
Dt : float
dt : float
dvl : bool
etak_1
gyro : bool
nuk
t : float
t_1 : float
GetInput()
Localize(xk_1, uk)
GL.GL
k
pk : Histogram2D
pk_1 : Histogram2D
pk_hat : Histogram2D
robot
GetInput(usk)
GetMeasurements()
LocalizationLoop(p0, usk)
Localize(pxk_1, uk, zk)
MeasurementProbability(zk)
StateTransitionProbability()
StateTransitionProbability_4_uk(uk)
uk2cell(uk)
GL_4DOFAUV.GL_4DOFAUV
Delta_etak
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)

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

`StateTransitionProbability()`

Computes the complete state transition probability matrix. The matrix is a $n_u imes m_u imes n^2$ 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)`

Parameters
uk –

Returns

MeasurementProbability(*zk*)

Computes the measurement probability histogram given the robot pose η_k and the measurement z_k. In this case the 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_4DOFAUV.LocalizationLoop()` method of the `GL_4DOFAUV.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

3 DOF Differential Drive Mobile Robot

Differential Drive 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*

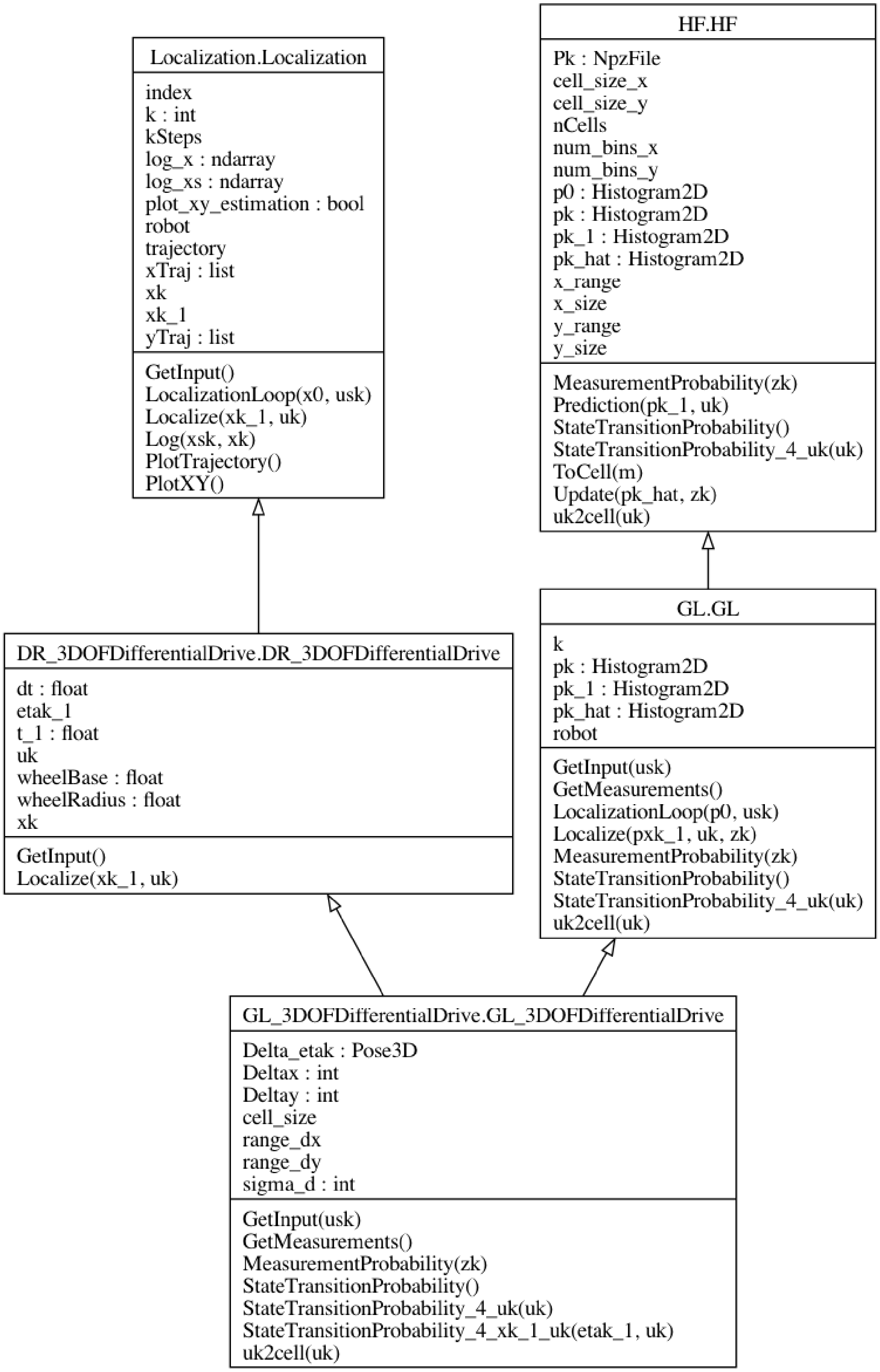
Grid Reckoning Localization for a 4 DOF AUV.

__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.

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


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()
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
MeasurementProbability(zk)
Prediction(pk_1, uk)
StateTransitionProbability()
StateTransitionProbability_4_uk(uk)
ToCell(m)
Update(pk_hat, zk)
uk2cell(uk)
GL.GL
k
pk : Histogram2D
pk_1 : Histogram2D
pk_hat : Histogram2D
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
etak_1
t_1 : float
uk
wheelBase : float
wheelRadius : float
xk
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)

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

`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)`

Parameters
uk –

Returns

`MeasurementProbability(zk)`

Computes the measurement probability histogram given the robot pose η_k and the measurement z_k. In this case the 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_4DOFAUV.LocalizationLoop()` method of the `GL_4DOFAUV.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

1.6.4 Montecarlo Localization

Montecarlo Localization

To be completed…

1.6.5 Gaussian Filter Localization

Gaussian Filter Localization

class GFLocalization.**GFLocalization**(*index*, *kSteps*, *robot*, *x0*, *P0*, **args*)

Bases: *Localization*, GaussianFilter

Map-less localization using KF and EKF filters.

__init__(*index*, *kSteps*, *robot*, *x0*, *P0*, **args*)

Constructor.

Parameters

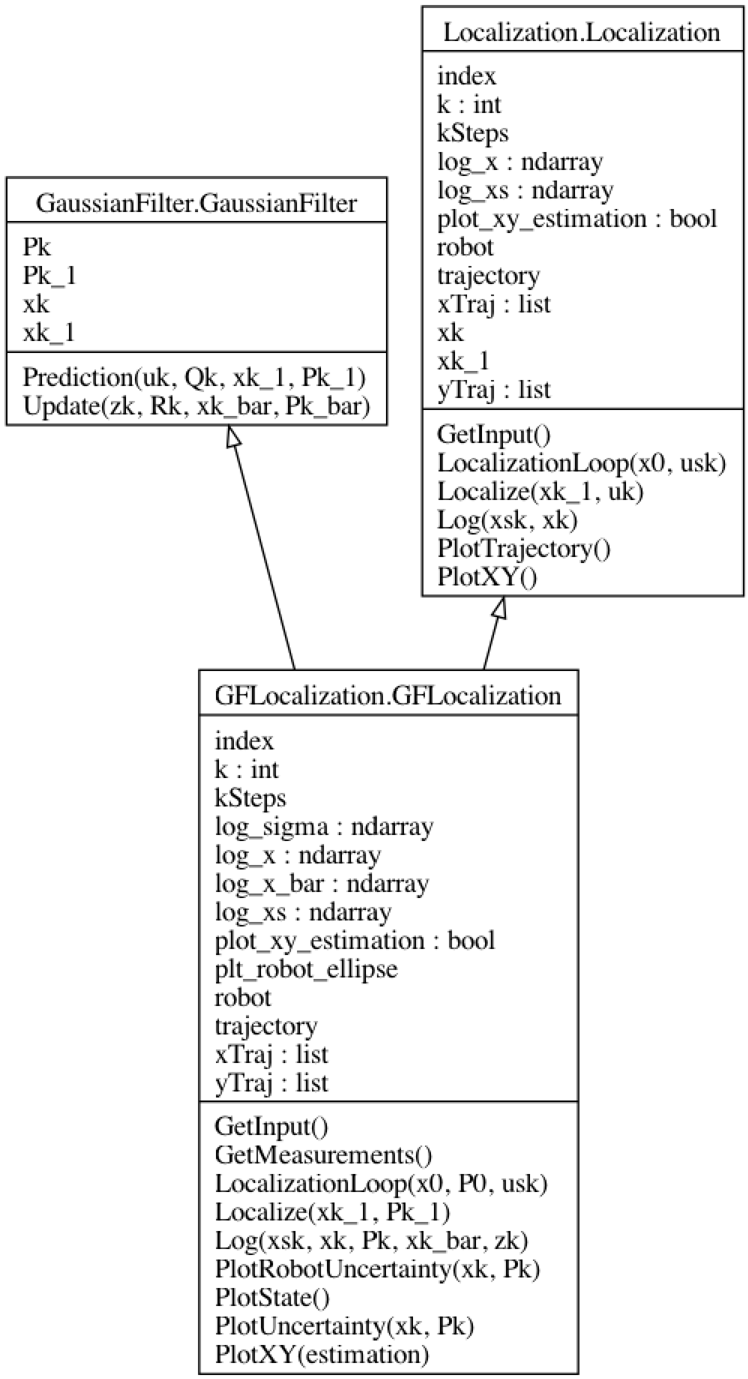
- **x0** – initial state
- **P0** – initial covariance
- **index** – Named tuple used to map the state vector, the simulation vector and the observation vector (`prpy.IndexStruct`)
- **kSteps** – simulation time steps
- **robot** – Simulated Robot object
- **args** – arguments to be passed to the parent constructor

GetInput()

Get the input from the robot. Relates to the motion model as follows:

$$\begin{aligned} x_k &= f(x_{k-1}, u_k, w_k) \\ w_k &= N(0, Q_k) \end{aligned} \tag{1.41}$$

To be overidden by the child class .


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)

Return uk, Qk
input and covariance of the motion model

`GetMeasurements()`

Get the measurements from the robot. Corresponds to the observation model:

$$\begin{aligned} z_k &= h(x_k, v_k) \\ v_k &= N(0, R_k) \end{aligned} \tag{1.42}$$

To be overidden by the child class .

Return zk, Rk
observation vector and covariance of the observation noise.

`Localize`(*xk_1*, *Pk_1*)

Localization iteration. Reads the input of the motion model, performs the prediction step, reads the measurements, performs the update step and logs the results. The method also plots the uncertainty ellipse of the robot pose.

Parameters

- **xk_1** – previous state vector
- **Pk_1** – previous covariance matrix

Return xk, Pk
updated state vector and covariance matrix

`LocalizationLoop`(*x0*, *P0*, *usk*)

Localization loop. During *self.kSteps* it calls the `Localize()` method for each time step.

Parameters

- **x0** – initial state vector
- **P0** – initial covariance matrix

`Log`(*xsk*, *xk*, *Pk*, *xk_bar*, *zk*)

Logs the results for later plotting.

Parameters

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

`PlotState()`

Plot the results of the localization For each state DOF s -si[s] is the corresponding simulated stated -x1[s] is the corresponding observation

`PlotXY`(*estimation=True*)

Plot the x-y trajectory of the robot simulation: True if the simulated XY robot trajectory is available

`PlotRobotUncertainty`(*xk*, *Pk*)

`PlotUncertainty`(*xk*, *Pk*)

4 DOF AUV

4 DOF AUV EKF Localization using an Input Velocity Motion Model with Depht, Yaw and Linear Velocity Measurements

class EKF_4DOFAUV_InputVelocityMM_DVLDepthYawOM.**EKF_4DOFAUV_InputVelocityMM_DVLDepthYawOM**(*kSteps*, *robot*, **args*)

Bases: `GFLocalization`, `DR_4DOFAUV_DVLGyro`, `EKF`

This class implements an EKF localization filter for a 4 DOF AUV using an input velocity motion model incorporating DVL linear velocity measurements, a gyro angular speed measurement, as well as depth and yaw measurements. Inherits from GFLocalization because it is a Localization method using Gaussian filtering, and from EKF because it uses an EKF. It also inherits from DR_4DOFAUV_DVLGyro to reuse its motion model `solved_prlab.DR_4DOFAUV_DVLGyro.Localize()` and the model input `solved_prlab.DR_4DOFAUV_DVLGyro.GetInput()`.

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

Constructor.

Parameters

args – arguments to be passed to the base class constructor

f(*xk_1*, *uk*)

Non-linear motion model using as input the DVL linear velocity and the gyro angular speed:

$$\begin{aligned} x_k &= f(x_{k-1}, u_k, w_k) = x_{k-1} \oplus (u_k + w_k)\Delta t \\ x_{k-1} &= [x_{k_1}^T, y_{k_1}^T, z_{k_1}^T, \psi_{k_1}^T]^T \\ u_k &= [u_k, v_k, w_k, r_k]^T \end{aligned}$$

Parameters

- **xk_1** – previous mean state vector ($x_{k-1} = [x_{k-1}^T, y_{k-1}^T, z_{k-1}^T, \psi_{k-1}^T]^T$) containing the robot position and heading in the N-Frame
- **uk** – input vector $u_k = [u_k^T, v_k^T, w_k^T, r_k^T]^T$ containing the DVL linear velocity and the gyro angular speed, both referenced in the B-Frame

Returns

current mean state vector containing the current robot position and heading ($x_k = [x_k^T, y_k^T, z_k^T, \psi_k^T]^T$) represented in the N-Frame

Jfx(*xk_1*)

Jacobian of the motion model with respect to the state vector:

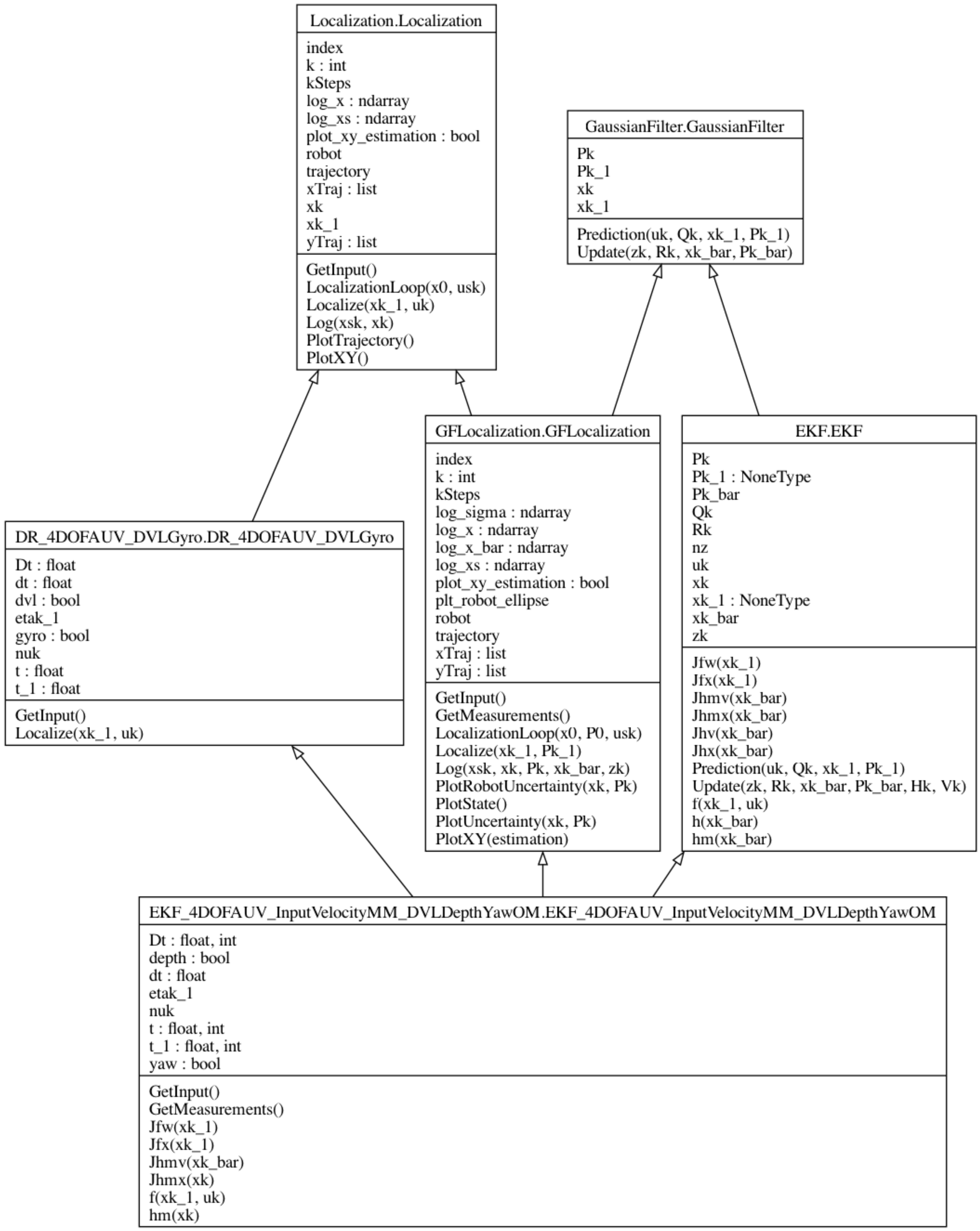
$$J_{fx} = \frac{\partial f(x_{k-1}, u_k, w_k)}{\partial x_{k-1}} = \frac{\partial x_{k-1} \oplus (u_k + w_k)}{\partial x_{k-1}} = J_{1\oplus} \tag{1.43}$$

Parameters

xk_1 – Linearization point. By default the linearization point is the previous state vector taken from a class attribute.

Returns

Jacobian matrix


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)
EKF.EKF
Pk
Pk_1 : NoneType
Pk_bar
Qk
Rk
nz
uk
xk
xk_1 : NoneType
xk_bar
zk
Jfw(xk_1)
Jfx(xk_1)
Jhmv(xk_bar)
Jhmx(xk_bar)
Jhv(xk_bar)
Jhx(xk_bar)
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar, Hk, Vk)
f(xk_1, uk)
h(xk_bar)
hm(xk_bar)
DR_4DOFAUV_DVLGyro.DR_4DOFAUV_DVLGyro
Dt : float
dt : float
dvl : bool
etak_1
gyro : bool
nuk
t : float
t_1 : float
GetInput()
Localize(xk_1, uk)
EKF_4DOFAUV_InputVelocityMM_DVLDepthYawOM.EKF_4DOFAUV_InputVelocityMM_DVLDepthYawOM
Dt : float, int
depth : bool
dt : float
etak_1
nuk
t : float, int
t_1 : float, int
yaw : bool
GetInput()
GetMeasurements()
Jfw(xk_1)
Jfx(xk_1)
Jhmv(xk_bar)
Jhmx(xk)
f(xk_1, uk)
hm(xk)

`Jfw(xk_1)`

Jacobian of the motion model with respect to the motion model noise vector:

$$J_{fx} = \frac{\partial f(x_{k-1}, u_k, w_k)}{\partial w_k} = \frac{\partial x_{k-1} \oplus (u_k + w_k)}{\partial w_k} = J_{2\oplus} \tag{1.44}$$

Parameters

xk_1 – Linearization point. By default the linearization point is the previous state vector taken from a class attribute.

Returns

Jacobian matrix

`hm(xk_bar)`

Observation model related to the measurements observations of the EKF to be overwritten by the child class.

Parameters

xk_bar – mean of the predicted state vector. By default it is taken from the class attribute.

Returns

expected observation vector

`Jhmx(xk)`

Jacobian of the measurement model with respect to the state vector:

$$J_{hmx} = H_{m_k} = \frac{\partial h_m(x_k, v_k)}{\partial x_k} = \frac{\partial [z_{depth}^T, \psi_{compass}^T]^T}{\partial x_k} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.45}$$

Parameters

xk – mean state vector containing the robot position and heading ($x_k = [x_k^T, y_k^T, z_k^T, \psi_k^T]^T$) represented in the N-Frame

Returns

observation matrix (Jacobian) matrix eq. (1.45).

`Jhmv(xk_bar)`

Jacobian of the measurement model with respect to the measurement noise vector:

$$J_{hmv} = V_{m_k} = \frac{\partial h_m(x_k, v_k)}{\partial v_k} = I_{2\times 2} \tag{1.46}$$

Parameters

xk – mean state vector containing the robot position and heading ($x_k = [x_k^T, y_k^T, z_k^T, \psi_k^T]^T$) represented in the N-Frame

Returns

observation noise (Jacobian) matrix eq. (1.46).

`GetInput()`

Get the input from the robot. Relates to the motion model as follows:

$$\begin{aligned} x_k &= f(x_{k-1}, u_k, w_k) \\ w_k &= N(0, Q_k) \end{aligned} \tag{1.47}$$

To be overidden by the child class .

Return uk, Qk

input and covariance of the motion model

GetMeasurements()

Get the measurements from the robot. Corresponds to the observation model:

$$\begin{aligned} z_k &= h(x_k, v_k) \\ v_k &= N(0, R_k) \end{aligned} \tag{1.48}$$

To be overidden by the child class .

Return zk, Rk
observation vector and covariance of the observation noise.

4 DOF AUV EKF Localization using a Constant Velocity Motion Model with Depht, Yaw and Linear Velocity Measurements

class EKF_4DOFAUV_CtVelocityMM_DVLDepthYawOM.**EKF_4DOFAUV_CtVelocityMM_DVLDepthYawOM**(*kSteps*, *robot*, **args*)

Bases: *GFLocalization*, *DR_4DOFAUV_DVLGyro*, *EKF*

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

Constructor.

Parameters

- **x0** – initial state
- **P0** – initial covariance
- **index** – Named tuple used to map the state vector, the simulation vector and the observation vector (prpy.IndexStruct)
- **kSteps** – simulation time steps
- **robot** – Simulated Robot object
- **args** – arguments to be passed to the parent constructor

f(*xk_1*, *uk*)

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

Parameters

- **xk_1** – previous mean state vector
- **uk** – input vector

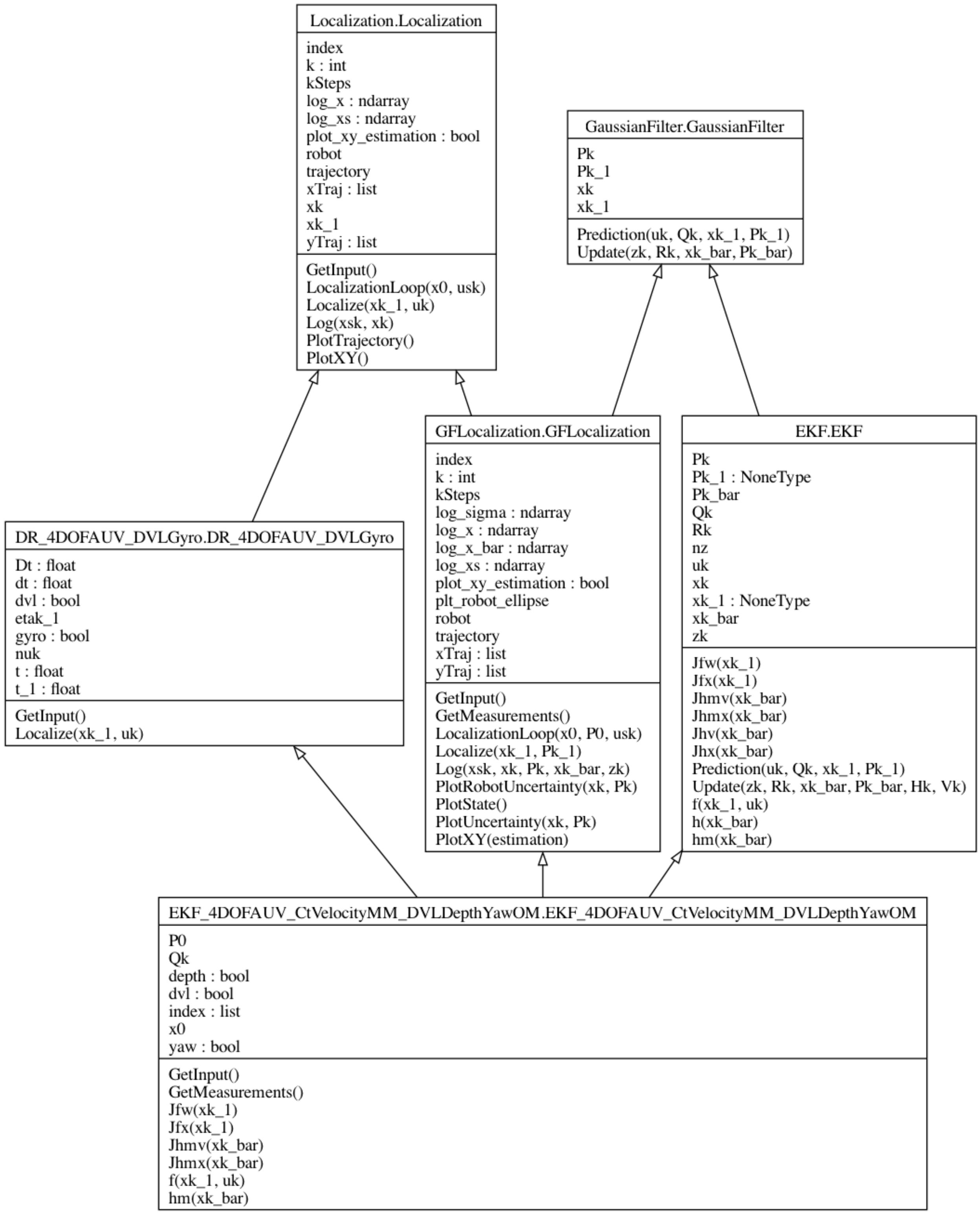
Return xk_bar, Pk_bar
predicted mean state vector and its covariance matrix

Jfx(*xk_1*)

Jacobian of the motion model with respect to the state vector. *Method to be overwritten by the child class*.

Parameters
xk_1 – Linearization point. By default the linearization point is the previous state vector taken from a class attribute.

Returns
Jacobian matrix


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)
EKF.EKF
Pk
Pk_1 : NoneType
Pk_bar
Qk
Rk
nz
uk
xk
xk_1 : NoneType
xk_bar
zk
Jfw(xk_1)
Jfx(xk_1)
Jhmv(xk_bar)
Jhmx(xk_bar)
Jhv(xk_bar)
Jhx(xk_bar)
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar, Hk, Vk)
f(xk_1, uk)
h(xk_bar)
hm(xk_bar)
DR_4DOFAUV_DVLGyro.DR_4DOFAUV_DVLGyro
Dt : float
dt : float
dvl : bool
etak_1
gyro : bool
nuk
t : float
t_1 : float
GetInput()
Localize(xk_1, uk)
EKF_4DOFAUV_CtVelocityMM_DVLDepthYawOM.EKF_4DOFAUV_CtVelocityMM_DVLDepthYawOM
P0
Qk
depth : bool
dvl : bool
index : list
x0
yaw : bool
GetInput()
GetMeasurements()
Jfw(xk_1)
Jfx(xk_1)
Jhmv(xk_bar)
Jhmx(xk_bar)
f(xk_1, uk)
hm(xk_bar)

Jfw(*xk_1*)

Jacobian of the motion model with respect to the noise vector. *Method to be overwritten by the child class*.

Parameters

xk_1 – Linearization point. By default the linearization point is the previous state vector taken from a class attribute.

Returns

Jacobian matrix

hm(*xk_bar*)

Observation model related to the measurements observations of the EKF to be overwritten by the child class.

Parameters

xk_bar – mean of the predicted state vector. By default it is taken from the class attribute.

Returns

expected observation vector

Jhmx(*xk_bar*)

Jhmv(*xk_bar*)

GetInput()

Get the input from the robot. Relates to the motion model as follows:

$$
\begin{aligned}
x_k &= f(x_{k-1}, u_k, w_k) \\
w_k &= N(0, Q_k)
\end{aligned}
\tag{1.49}
$$

To be overidden by the child class .

Return uk, Qk

input and covariance of the motion model

GetMeasurements()

Get the measurements from the robot. Corresponds to the observation model:

$$
\begin{aligned}
z_k &= h(x_k, v_k) \\
v_k &= N(0, R_k)
\end{aligned}
\tag{1.50}
$$

To be overidden by the child class .

Return zk, Rk

observation vector and covariance of the observation noise.

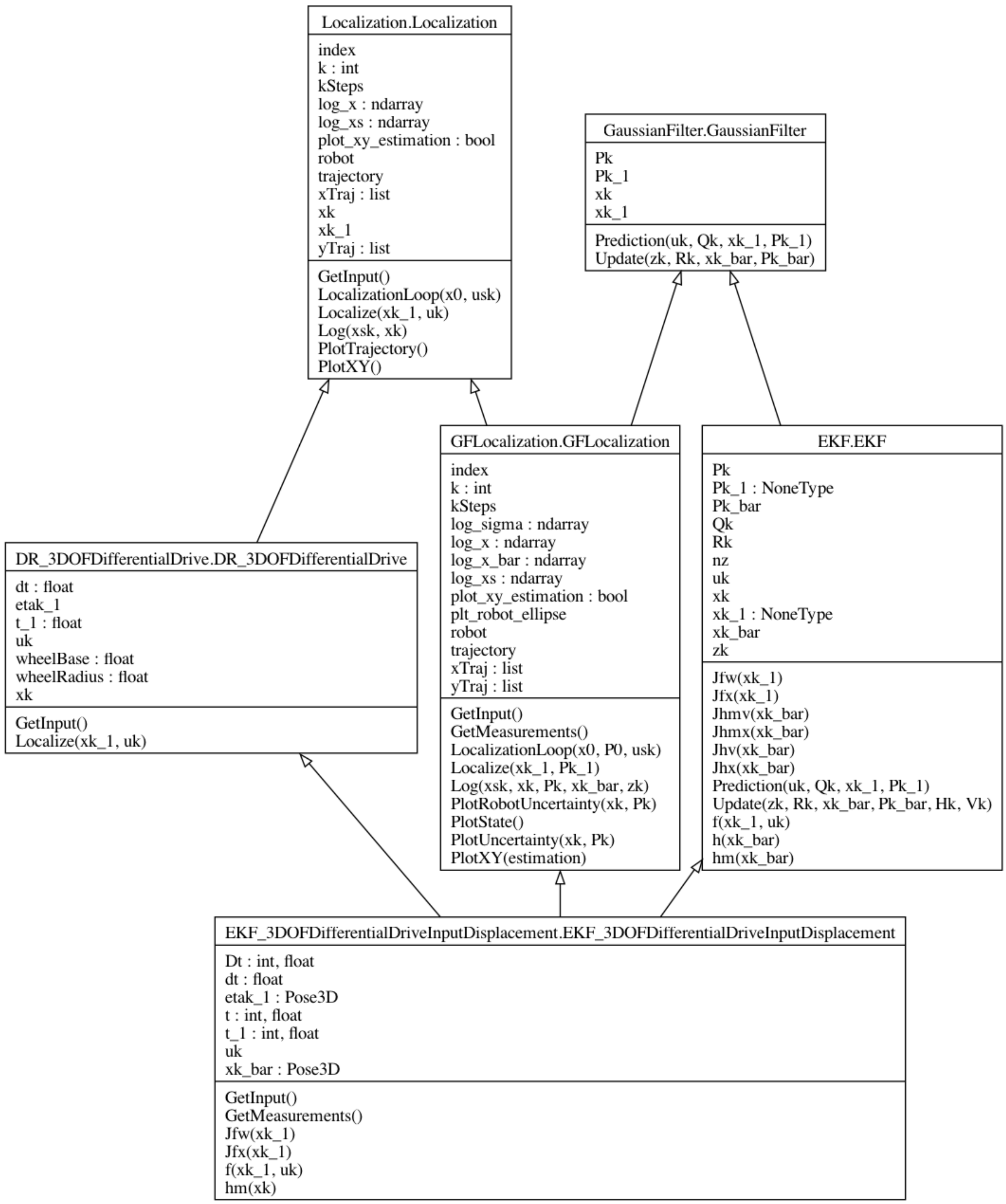
3 DOF Differential Drive Mobile Robot

Differential Drive Grid EKF Using an Input displacement Motion Model

class EKF_3DOFDifferentialDriveInputDisplacement.**EKF_3DOFDifferentialDriveInputDisplacement**(*kSteps*, *robot*, **args*)

Bases: *GFLocalization*, *DR_3DOFDifferentialDrive*, *EKF*

This class implements an EKF localization filter for a 4 DOF AUV using an input velocity motion model incorporating DVL linear velocity mesurements, a gyro angular speed measurement, as well as depth and yaw measurements. Inherit from `PoseCompounding4DOF` first, to ensure it uses the overridden $\oplus$ and $\ominus$ methods.


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)
EKF.EKF
Pk
Pk_1 : NoneType
Pk_bar
Qk
Rk
nz
uk
xk
xk_1 : NoneType
xk_bar
zk
Jfw(xk_1)
Jfx(xk_1)
Jhmv(xk_bar)
Jhmx(xk_bar)
Jhv(xk_bar)
Jhx(xk_bar)
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar, Hk, Vk)
f(xk_1, uk)
h(xk_bar)
hm(xk_bar)
DR_3DOFDifferentialDrive.DR_3DOFDifferentialDrive
dt : float
etak_1
t_1 : float
uk
wheelBase : float
wheelRadius : float
xk
GetInput()
Localize(xk_1, uk)
EKF_3DOFDifferentialDriveInputDisplacement.EKF_3DOFDifferentialDriveInputDisplacement
Dt : int, float
dt : float
etak_1 : Pose3D
t : int, float
t_1 : int, float
uk
xk_bar : Pose3D
GetInput()
GetMeasurements()
Jfw(xk_1)
Jfx(xk_1)
f(xk_1, uk)
hm(xk)

Then, it inherits from `GFLocalization` to implement a localization filter and, finally, it inherits from `EKF` to use the EKF Gaussian filter implementation for the localization.

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

Constructor.

Parameters

args – arguments to be passed to the base class constructor

`GetInput`()

Get the input from the robot. Relates to the motion model as follows:

$$\begin{aligned} x_k &= f(x_{k-1}, u_k, w_k) \\ w_k &= N(0, Q_k) \end{aligned} \tag{1.51}$$

To be overidden by the child class .

Return uk, Qk

input and covariance of the motion model

`f`(*xk_1*, *uk*)

Non-linear motion model using as input the DVL linear velocity and the gyro angular speed:

$$\begin{aligned} x_k &= f(x_{k-1}, u_k, w_k) = x_{k-1} \oplus (u_k + w_k)\Delta t \\ x_{k-1} &= [x_{k_1}^T, y_{k_1}^T, z_{k_1}^T, \psi_{k_1}^T]^T \\ u_k &= [u_k, v_k, w_k, r_k]^T \end{aligned}$$

Parameters

- **xk_1** – previous mean state vector ($x_{k-1} = [x_{k-1}^T, y_{k-1}^T, z_{k-1}^T, \psi_{k-1}^T]^T$) containing the robot position and heading in the N-Frame
- **uk** – input vector $u_k = [u_k^T, v_k^T, w_k^T, r_k^T]^T$ containing the DVL linear velocity and the gyro angular speed, both referenced in the B-Frame

Returns

current mean state vector containing the current robot position and heading ($x_k = [x_k^T, y_k^T, z_k^T, \psi_k^T]^T$) represented in the N-Frame

`Jfx`(*xk_1*)

Jacobian of the motion model with respect to the state vector:

$$J_{fx} = \frac{\partial f(x_{k-1}, u_k, w_k)}{\partial x_{k-1}} = \frac{\partial x_{k-1} \oplus (u_k + w_k)}{\partial x_{k-1}} = J_{1\oplus} \tag{1.52}$$

Parameters

xk_1 – Linearization point. By default the linearization point is the previous state vector taken from a class attribute.

Returns

Jacobian matrix

`Jfw`(*xk_1*)

Jacobian of the motion model with respect to the motion model noise vector:

$$J_{fx} = \frac{\partial f(x_{k-1}, u_k, w_k)}{\partial w_k} = \frac{\partial x_{k-1} \oplus (u_k + w_k)}{\partial w_k} = J_{2\oplus} \tag{1.53}$$

Parameters

xk_1 – Linearization point. By default the linearization point is the previous state vector taken from a class attribute.

Returns
Jacobian matrix

hm(*xk*)

Observation model related to the measurements observations of the EKF to be overwritten by the child class.

Parameters
xk_bar – mean of the predicted state vector. By default it is taken from the class attribute.

Returns
expected observation vector

GetMeasurements()

Gets the measurement vector and the measurement noise covariance matrix from the robot. The measurement vector contains the depth read from the depth sensor and the heading read from the compass sensor.

Returns
observation vector z_k and observation noise covariance matrix R_k defined in eq. `eq-zk-EKF_3DOFDifferentialDriveInputDisplacement`.

Differential Drive Grid EKF Using a Constant Velocity Motion Model

class `EKF_3DOFDifferentialDriveCtVelocity.`**EKF_3DOFDifferentialDriveCtVelocity**(*kSteps*, *robot*, **args*)

Bases: *GFLocalization*, *DR_3DOFDifferentialDrive*, *EKF*

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

Constructor.

Parameters

- **x0** – initial state
- **P0** – initial covariance
- **index** – Named tuple used to map the state vector, the simulation vector and the observation vector (`prpy.IndexStruct`)
- **kSteps** – simulation time steps
- **robot** – Simulated Robot object
- **args** – arguments to be passed to the parent constructor

f(*xk_1*, *uk*)

” Motion model of the EKF to be overwritten by the child class.

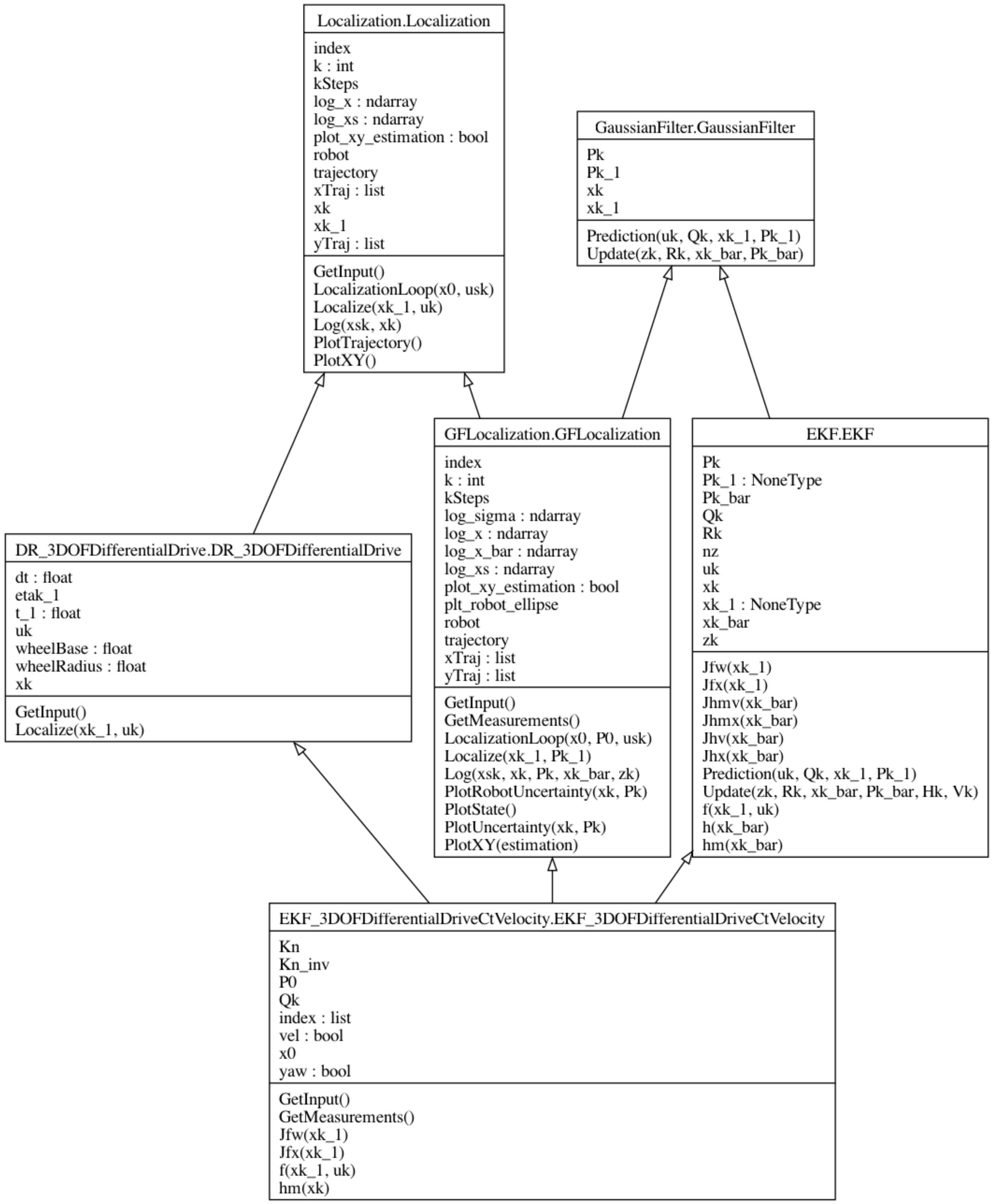
Parameters

- **xk_1** – previous mean state vector
- **uk** – input vector

Return xk_bar, Pk_bar
predicted mean state vector and its covariance matrix

Jfx(*xk_1*)

Jacobian of the motion model with respect to the state vector. *Method to be overwritten by the child class.*


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)
EKF.EKF
Pk
Pk_1 : NoneType
Pk_bar
Qk
Rk
nz
uk
xk
xk_1 : NoneType
xk_bar
zk
Jfw(xk_1)
Jfx(xk_1)
Jhmv(xk_bar)
Jhmx(xk_bar)
Jhv(xk_bar)
Jhx(xk_bar)
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar, Hk, Vk)
f(xk_1, uk)
h(xk_bar)
hm(xk_bar)
DR_3DOFDifferentialDrive.DR_3DOFDifferentialDrive
dt : float
etak_1
t_1 : float
uk
wheelBase : float
wheelRadius : float
xk
GetInput()
Localize(xk_1, uk)
EKF_3DOFDifferentialDriveCtVelocity.EKF_3DOFDifferentialDriveCtVelocity
Kn
Kn_inv
P0
Qk
index : list
vel : bool
x0
yaw : bool
GetInput()
GetMeasurements()
Jfw(xk_1)
Jfx(xk_1)
f(xk_1, uk)
hm(xk)

Parameters
`xk_1` – Linearization point. By default the linearization point is the previous state vector taken from a class attribute.

Returns
Jacobian matrix

`Jfw`(*xk_1*)

Jacobian of the motion model with respect to the noise vector. *Method to be overwritten by the child class*.

Parameters
`xk_1` – Linearization point. By default the linearization point is the previous state vector taken from a class attribute.

Returns
Jacobian matrix

`hm`(*xk*)

Observation model related to the measurements observations of the EKF to be overwritten by the child class.

Parameters
`xk_bar` – mean of the predicted state vector. By default it is taken from the class attribute.

Returns
expected observation vector

`GetInput()`

Get the input from the robot. Relates to the motion model as follows:

$$\begin{aligned} x_k &= f(x_{k-1}, u_k, w_k) \\ w_k &= N(0, Q_k) \end{aligned} \tag{1.54}$$

To be overidden by the child class .

Return uk, Qk
input and covariance of the motion model

`GetMeasurements()`

Get the measurements from the robot. Corresponds to the observation model:

$$\begin{aligned} z_k &= h(x_k, v_k) \\ v_k &= N(0, R_k) \end{aligned} \tag{1.55}$$

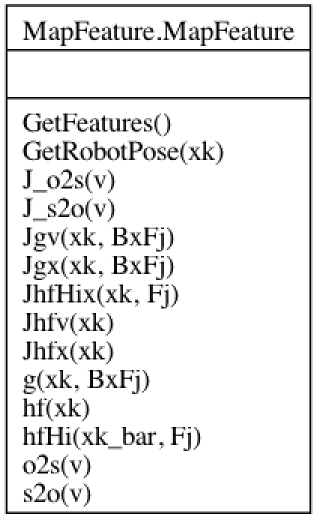
To be overidden by the child class .

Return zk, Rk
observation vector and covariance of the observation noise.

1.6.6 Feature Map based EKF Localzation

Feature Map based EKF Locallzation

Map Feature



class `MapFeature.`**`MapFeature`**(**args*)

Bases: `object`

This class provides the functionality required to use Map Features for Robo Localization. It has methods for reading the feature pose using the robot the sensors (`GetFeatures()`), as well as for computing its:

- observation model (`hf()`),
- inverse observation model (`g()`)
- all the required Jacobians (`Jhfx()`, `Jhfv()`, `Jgx()` and `Jgv()`).

When mapped, a feature may involve 2 different representations:

- The observation representation, which is the representation used by the sensors to observe the feature.
- The storage representation, which is the representation used to store the feature within the map within the state vector.

For instance, a feature may be observed in polar coordinates but stored in Cartesian coordinates. In this case, the observation representation is the polar coordinates and the storage representation is the Cartesian coordinates. The class provides method to convert from one representation to the other (`s2o()` and `o2s()`) and their corresponding Jacobians (`J_s2o()` and `J_o2s()`). By default, the observation representation is the same as the storage representation, but this behaviour may be overriden in child classes.

`__init__`(**args*)

`GetFeatures`()

Reads the Feature observations from the sensors. For all features within the field of view of the sensor, the method returns the list of robot-related poses, the covariance of their corresponding observation noise, the corresponding observation matrix and the noise Jacobian matrix. **This is a pure virtual method that must be overriden in child classes**.

Returns

vector of features observations in the B-Frame and the covariance of their corresponding noise.

- $z_k = [{}^B x_{F_i}^T \cdots {}^B x_{F_j}^T \cdots {}^B x_{F_k}^T]^T$
- $R_k = block\_diag([R_{F_i} \cdots R_{F_j} \cdots R_{F_k}])$
- $H_k = block\_diag([H_{F_i} \cdots H_{F_j} \cdots H_{F_k}])$

- $V_K = I_{z_{nf} \times z_{nf}}$

s2o(v)

Conversion function from the storage representation to the observation representation. By default, it returns the same vector as the one provided as input, assuming that the observation representation is the same as the storage representation. In case it is not, this method must be overriden in the child class.

Parameters
v – vector in the storage representation

Returns
vector in the observation representation

o2s(v)

Conversion function from the observation representation to the storage representation. By default, it returns the same vector as the one provided as input, assuming that the observation representation is the same as the storage representation. In case it is not, this method must be overriden in child classes.

Parameters
v – vector in the observation representation

Returns
vector in the storage representation

J_s2o(v)

Jacobian of the conversion function from the storage representation to the observation representation. By default, it returns the identity matrix, assuming that the observation representation is the same as the storage representation. In case it is not, this method must be overriden in the derived class.

Parameters
v – vector in the storage representation

Returns
Jacobian of the conversion function from the storage representation to the observation representation

J_o2s(v)

Jacobian of the conversion function from the observation representation to the storage representation. By default, it returns the identity matrix, assuming that the observation representation is the same as the storage representation. In case it is not, this method must be overriden in the derived class.

Parameters
v – vector in the observation representation

Returns
Jacobian of the conversion function from the observation representation to the storage representation

hf(xk)

This is the direct observation model, implementing the feature observation function for the data association hypothesis $H = [H_1 \; \cdots \; H_i \; \cdots \; H_{n_{zf}}]$ stored in the attibute `FEKFMBL.FEKFMBL.H`. Given the obsevation vector $z_f = [z_{f_1}^T \; \cdots \; z_{f_i}^T \; \cdots \; z_{f_{n_{zf}}}^T]^T$, the state vector $x_k = [{}^N x_B^T \; x_{rest}^T]^T$ and the observation noise $v_k = [v_{f1_k}^T \; \cdots \; v_{fi_k}^T \; \cdots \; v_{fn_{zf_k}}^T]^T$ the observation equation is given by:

$$z_f = h_f(x_k, v_k) \tag{1.56}$$

which may be expanded as follows:

$$\begin{bmatrix} z_{f_1} \\ \vdots \\ z_{f_i} \\ \vdots \\ z_{n_{zf}} \end{bmatrix} = \begin{bmatrix} h_{f_{H_1}}(x_k, v_k) \\ \vdots \\ h_{f_{H_i}}(x_k, v_k) \\ \vdots \\ h_{f_{H_{n_{zf}}}}(x_k, v_k) \end{bmatrix} = \begin{bmatrix} s2o(\ominus^N x_B \boxplus^N x_{F_{H_1}}) + v_{f1_k} \\ \vdots \\ s2o(\ominus^N x_B \boxplus^N x_{F_{H_i}}) + v_{fi_k} \\ \vdots \\ s2o(\ominus^N x_B \boxplus^N x_{F_{H_{n_{zf}}}}) + v_{fn_{zf}} \end{bmatrix} \tag{1.57}$$

being `hfHi()` the observation function (eq. (1.59)) for the data association hypothesis $H_i \Rightarrow z_{fi} \rightarrow^N x_{F_{H_i}}$, and `s2o()` the conversion function from the storage representation to the observation one.

The method computes the expected observation h_f for the z_f observation. To do it, it iterates over each feature observation z_{f_i} calling the method `hfHi()` to compute the expected observation $h_{f_{H_i}}$ for each feature observation z_{f_i}, collecting all them in the returned vector.

Parameters
xk – state vector mean $\hat{x}_k$.

Returns
vector of expected features observations corresponding to the vector of observed features z_f.

Jhfx(*xk*)

Computes the Jacobian of the feature observation function `hf()` (eq. (1.55)), with respect to the state vector $\bar{x}_k$:

$$J_{hfx} = \frac{\partial h_f(x_k, v_k)}{\partial x_k} = \begin{bmatrix} \frac{\partial h_{f_{H_1}}(x_k,v_k)}{\partial x_k} \\ \vdots \\ \frac{\partial h_{f_{H_i}}(x_k,v_k)}{\partial x_k} \\ \vdots \\ \frac{\partial h_{f_{H_{n_{zf}}}}(x_k,v_k)}{\partial x_k} \end{bmatrix} = \begin{bmatrix} J_{hfH1x} \\ \vdots \\ J_{hfH2x} \\ \vdots \\ J_{hfHn_{zf}x} \end{bmatrix} \tag{1.58}$$

where J_{hfHix} is the Jacobian of the observation function `hfHi()` (eq. (1.60)) for the feature observation z_{f_i}. To do it, given a vector of observations $z_f = [z_{f_1} \;\cdots\; z_{f_i} \;\cdots\; z_{f_{n_{zf}}}]$ this method iterates over each feature observation z_{f_i} calling the method `JhfHix()` to compute the Jacobian of the observation function for each feature observation (J_{hfHix}), collecting all them in the returned Jacobian matrix J_{hfx}.

Parameters
xk – state vector mean $\hat{x}_k$.

Returns
Jacobian of the observation function `hf()` with respect ro the robot pose $J_{hfx} = \frac{\partial h_f(\bar{x}_k, v_{f_k})}{\bar{x}_k}$

Jhfv(*xk*)

Computes the Jacobian of the observation function `hf()` with respect to the observation noise v_k. Normally, the observation noise in the observation B-Frame is linear (see eq. (1.56)) so the Jacobian is the identity matrix.

$$J_{hfv} = \frac{\partial h_f(x_k, v_k)}{\partial v_k}$$
$$= \begin{bmatrix} \frac{\partial h_{f_{H_1}}(x_k,v_k)}{\partial v_k} \\ \vdots \\ \frac{\partial h_{f_{H_i}}(x_k,v_k)}{\partial v_k} \\ \vdots \\ \frac{\partial h_{f_{H_{n_{zf}}}}(x_k,v_k)}{\partial v_k} \end{bmatrix} = \begin{bmatrix} \frac{\partial h_{f_{H_1}}(x_k,v_k)}{\partial v_{f1_k}} & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \frac{\partial h_{f_{H_1}}(x_k,v_k)}{\partial v_{fi_k}} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \frac{\partial h_{f_{H_1}}(x_k,v_k)}{\partial v_{fn_{zf_k}}} \end{bmatrix} = I_{n_{zf} \times n_{zf}} \tag{1.59}$$

If it is not the case, this method must be overriden.

Parameters

xk – state vector mean $\hat{x}_k$.

Returns

Jacobian of the observation function *hf()* with respect ro the observation noise v_k $J_{hfv} = I_{n_{zf} \times n_{zf}}$

hfHi(*xk_bar*, *Fj*)

This is the direct observation model for a single feature observation z_{f_i} , so it implements its related observation function (see eq. (1.59)). For a single feature observation z_{f_i} of the feature ${}^N x_{F_{H_i}}$ the method computes its expected observation from the current robot pose ${}^N x_B$. This function uses a generic implementation through the following equation:

$$z_{f_i} = h_{f_{H_i}}(x_k, v_k) = s2o(\ominus^N x_B \boxplus^N x_{F_{H_i}}) + v_{fi_k} \tag{1.60}$$

Wwere ${}^N x_B$ is the robot pose included within the state vector ($x_k = [{}^N x_B^T \; x_{rest}^T]^T$) and *s2o()* is a conversion function from the store representation to the observation representation.

The method is called by *hf()* to compute the expected observation for each feature observation contained in the observation vector $z_f = [z_{f_1}^T \; \cdots \; z_{1_i}^T \; \cdots \; z_{f_{n_{zf}}}^T]^T$.

Parameters

- **xk_bar** – mean of the predicted state vector
- **Fj** – map index of the observed feature: ${}^N x_{F_j} = self.M[Fj]$

Returns

expected observation of the feature ${}^N x_{F_j}$

JhfHix(*xk*, *Fj*)

Jacobian of the single feature direct observation model *hfHi()* (eq. (1.59)) with respect to the state vector $\bar{x}_k$:

$$\begin{aligned} x_k &= [{}^N x_B^T \; x_{rest}^T]^T \\ {}^N x_B &= F \cdot x_k; F = \begin{bmatrix} I_{p \times p} & 0_{p \times np} \end{bmatrix} \\ J_{hfHix} &= \frac{\partial h_{f_{zfi}}(\bar{x}_k, {}^N x_{F_j}, v_k)}{\partial x_k} = \frac{\partial s2o(\ominus F \cdot x_k \boxplus^N x_{F_j}) + v_{fi_k}}{\partial x_k} \\ &= J_{s2o}(\ominus^N x_B \boxplus^N x_{F_{H_i}}) J_{1\boxplus}(\ominus^N x_B, {}^N x_{F_{H_i}}) J_{\ominus}({}^N x_B) F \end{aligned} \tag{1.61}$$

where p is the dimension of the robot pose ${}^N x_B$ and np is the dimension of the rest of the state vector x_{rest}.

Parameters

- **xk** – state vector mean
- **Fj** – map index of the observed feature

Returns

Jacobian matrix defined in eq. (1.60)

g(*xk*, *BxFj*)

This method provides a generic implementation of the inverse observation model. It computes the feature pose in the N-Frame ${}^N x_{F_j}$ by compounding the robot pose ${}^N x_B$, from where the observation was taken, with the B-Frame referenced feature observation $B x_{F_j}$:

$$ {}^N x_{F_j} = ^N x_B \boxplus o2s({}^B x_{F_j} + v_k) \tag{1.62} $$

In this case, `o2s()` is the conversion function converting from the observation space to the representation one. It is worth noting that the robot pose ${}^N x_B$ is included within the state vector x_k but might not be the whole state vector. For instance, in some cases the state vector may include as well the robot velocity $x_k = [{}^N x_B^T \; {}^B \nu_k^T]^T$. Note that the `g()` works with a single feature observation, instead than with a vector of feature observations.

Parameters

- **xk** – mean state vector containing the robot pose ${}^N x_B$ from where the observation was taken
- **BxFj** – feature observation in the B-Frame ${}^B x_{F_j}$

Returns

mean feature pose in the N-Frame ${}^N x_{F_j}$

Jgx(*xk*, *BxFj*)

Jacobian of the inverse observation model `g()`, with respect to the state vector x_k. According to the generic implementation of the inverse observation model `g()` eq. (1.61), the Jacobian is computed as follows:

$$ J_{gx} = \frac{\partial g({}^N x_B, ^B x_{F_j})}{x_k} = \begin{bmatrix} J_{1\boxplus}({}^N x_B, o2s({}^B x_{F_j})) & 0 \end{bmatrix} \tag{1.63} $$

The zero submatrix, if present, corresponds to the derivate with respect to the non-positional elements of the state vector, for instance the robot velocity ${}^B \nu_k$, in case this was included within the state vector. If the state vector only containes the pose, then the 0 submatix vanishes.

Parameters

xk_bar – predicted state vector

Returns

Jacobian of the inverse observation model `g()` with respect to the state vector (eq. (1.62))

Jgv(*xk*, *BxFj*)

Jacobian of the inverse observation model `g()`, with respect to the observation noise v_k. According to the generic implementation of the inverse observation model `g()` eq. (1.61), the Jacobian is computed as follows:

$$ J_{gv} = \frac{\partial g({}^N x_k, ^B x_{F_j}), v_k}{v_k} = J_{2\boxplus}({}^N x_B, o2s({}^B x_{F_j})) J_{o2s}({}^B x_{F_j}) \tag{1.64} $$

Parameters

- **xk** – state vector containing the robot pose ${}^N x_B$ from where the observation was taken

- **BxFj** – feature observation in the B-Frame ${}^B x_{F_j}$

Returns
Jacobian of the inverse observation model `g()` with respect to the observation noise J_{gv} (see eq. (1.63)).

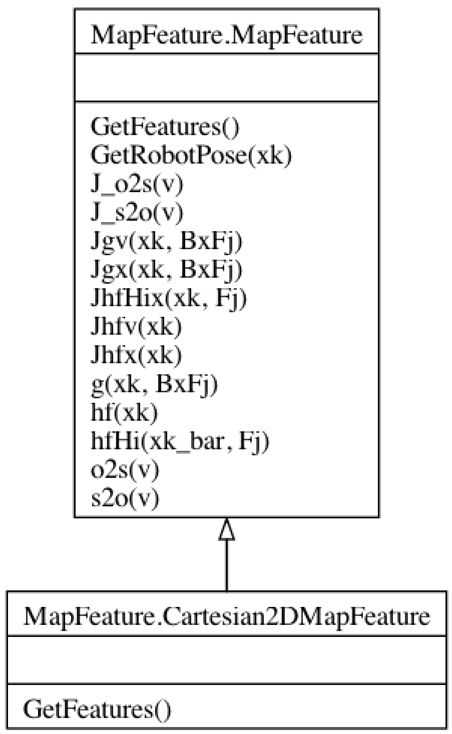
`GetRobotPose(xk)`

Extracts the robot pose from the state vector.

Parameters
xk – mean of the state vector:math:*x_k*

Returns
mean robot pose x_{B_k}

Cartesian Map Feature



`class MapFeature.Cartesian2DMapFeature(*args)`

Bases: `MapFeature`

This class inherits from the `MapFeature` and implements a 2D Cartesian feature model for the MBL problem. The Cartesian coordinates are used for both, observing the feature and for its storage within the map. This class overrides the `GetFeatures()` method to read the 2D Cartesian Features from the robot.

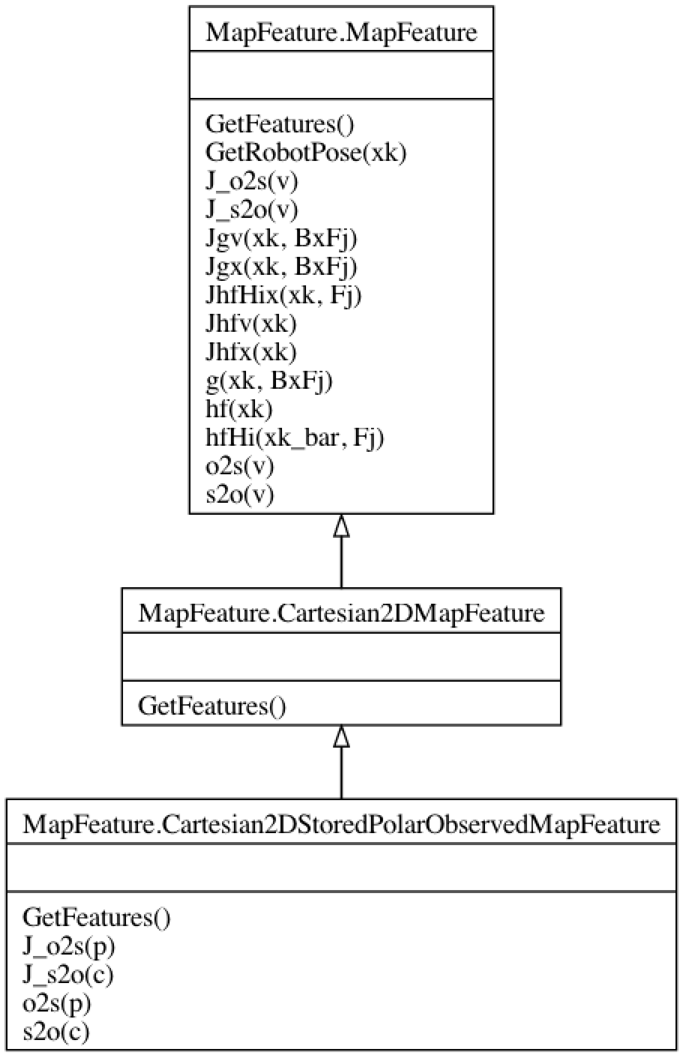
`GetFeatures()`

Reads the Features observations from the sensors. For all features within the field of view of the sensor, the method returns the list of robot-related poses and the covariance of their corresponding observation noise in **2D Cartesian coordinates**.

Return zk, Rk
list of Cartesian features observations in the B-Frame and the covariance of their

corresponding observation noise: * $z_k = [{}^Bx_{F_i}^T \cdots {}^Bx_{F_j}^T \cdots {}^Bx_{F_k}^T]^T$ * $R_k = block\_diag([R_{F_i} \cdots R_{F_j} \cdots R_{F_k}])$

Cartesian Map Feature Observed in Polar Coordinates



class `MapFeature.`**`Cartesian2DStoredPolarObservedMapFeature`**(**args*)

Bases: `Cartesian2DMapFeature`

This class implements the `MapFeature` interface for landmarks observed in polar coordinates and stored in Cartesian coordinates. It inherits from `Cartesian2DMapFeature` which provides the `Cartesian2DMapFeature.GetFeatures()` in charge of reading the 2D Cartesian Features from the robot and overrides the `o2s()` and `s2o()` methods to allow the conversion between the observation and storage representations.

`GetFeatures`()

Reads the Features observations from the sensors. For all features within the field of view of the sensor, the method returns the list of robot-related poses and the covariance of their corresponding observation noise in **2D Cartesian coordinates**.

Return zk, Rk

list of features observations in the B-Frame and the covariance of their corresponding observation noise: * $z_k = [{}^B x_{F_i}^T \cdots {}^B x_{F_j}^T \cdots {}^B x_{F_k}^T]^T$ * $R_k = block\_diag([R_{F_i} \cdots R_{F_j} \cdots R_{F_k}])$

o2s(*p*)

Converts the feature from the observation frame to the sensor frame.

Parameters
p – feature in the observation frame

Returns
feature in the sensor frame

s2o(*c*)

Converts the feature from the sensor frame to the observation frame.

Parameters
c – feature in the sensor frame

Returns
feature in the observation frame

J_o2s(*p*)

Jacobian of the `o2s()` function.

Parameters
p – feature in the observation frame

Returns
Jacobian of the `o2s()` function

J_s2o(*c*)

Jacobian of the `s2o()` function.

Parameters
c – feature in the sensor frame

Returns
Jacobian of the `s2o()` function

Feature Map based EKF Localization

class FEKFMBL.**FEKFMBL**(*M*, *alpha*, **args*)

Bases: `GFLocalization`, `MapFeature`

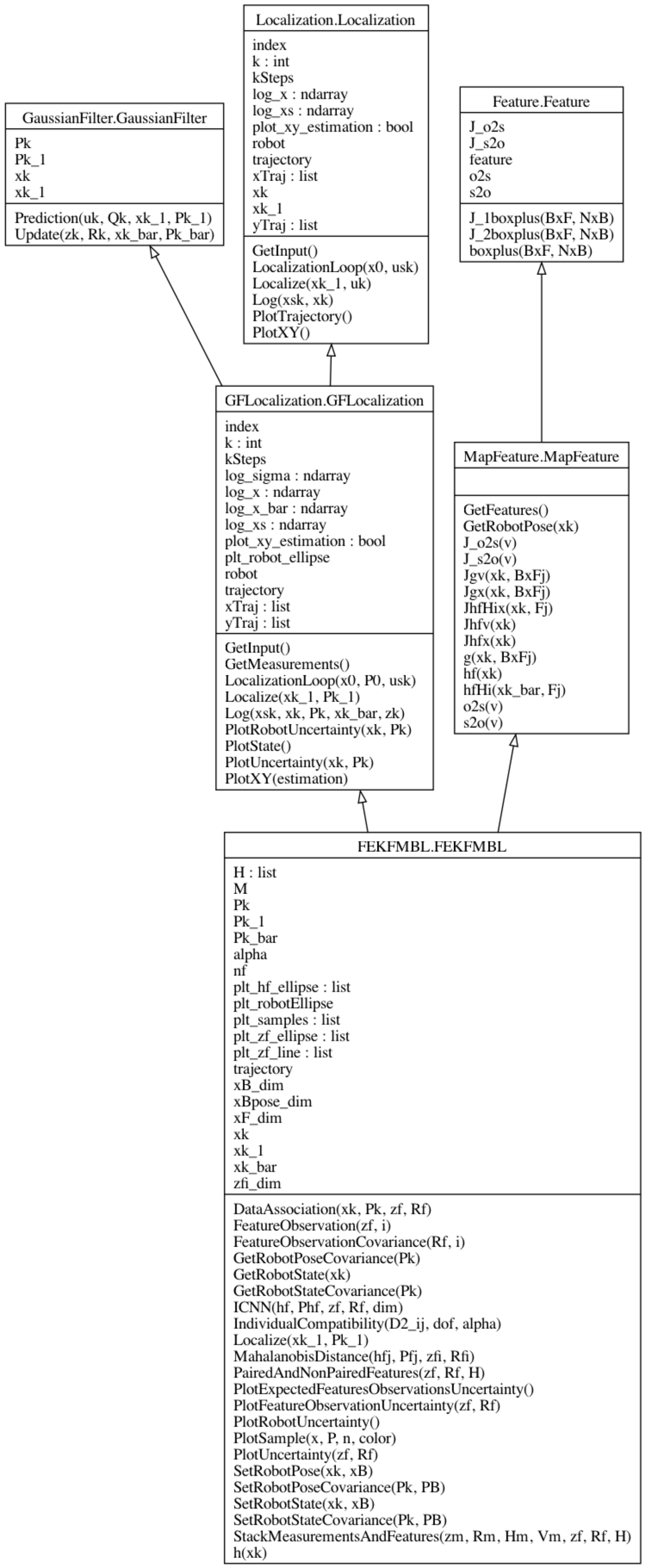
Feature Extended Kalman Filter Map based Localization class. Inherits from `GFLocalization.GFLocalization` and `MapFeature.MapFeature`. The first one provides the basic functionality of a localization algorithm, while the second one provides the basic functionality required to use features. `FEKFMBL.FEKFMBL` extends those classes by adding functionality to use a map based on features.

__init__(*M*, *alpha*, **args*)

Constructor of the FEKFMBL class.

Parameters

- **xBpose_dim** – dimensionality of the robot pose within the state vector
- **xB_dim** – dimensionality of the state vector
- **xF_dim** – dimentsionality of a feature
- **zfi_dim** – dimensionality of a single feature observation


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
Feature.Feature
J_o2s
J_s2o
feature
o2s
s2o
J_1boxplus(BxF, NxB)
J_2boxplus(BxF, NxB)
boxplus(BxF, NxB)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)
MapFeature.MapFeature
GetFeatures()
GetRobotPose(xk)
J_o2s(v)
J_s2o(v)
Jgv(xk, BxFj)
Jgx(xk, BxFj)
JhfHix(xk, Fj)
Jhfv(xk)
Jhfx(xk)
g(xk, BxFj)
hf(xk)
hfHi(xk_bar, Fj)
o2s(v)
s2o(v)
FEKFMBL.FEKFMBL
H : list
M
Pk
Pk_1
Pk_bar
alpha
nf
plt_hf_ellipse : list
plt_robotEllipse
plt_samples : list
plt_zf_ellipse : list
plt_zf_line : list
trajectory
xB_dim
xBpose_dim
xF_dim
xk
xk_1
xk_bar
zfi_dim
DataAssociation(xk, Pk, zf, Rf)
FeatureObservation(zf, i)
FeatureObservationCovariance(Rf, i)
GetRobotPoseCovariance(Pk)
GetRobotState(xk)
GetRobotStateCovariance(Pk)
ICNN(hf, Phf, zf, Rf, dim)
IndividualCompatibility(D2_ij, dof, alpha)
Localize(xk_1, Pk_1)
MahalanobisDistance(hfj, Pfj, zfi, Rfi)
PairedAndNonPairedFeatures(zf, Rf, H)
PlotExpectedFeaturesObservationsUncertainty()
PlotFeatureObservationUncertainty(zf, Rf)
PlotRobotUncertainty()
PlotSample(x, P, n, color)
PlotUncertainty(zf, Rf)
SetRobotPose(xk, xB)
SetRobotPoseCovariance(Pk, PB)
SetRobotState(xk, xB)
SetRobotStateCovariance(Pk, PB)
StackMeasurementsAndFeatures(zm, Rm, Hm, Vm, zf, Rf, H)
h(xk)

- **M** – Feature Based Map $M = [^N x_{F_1}^T \; ... \; ^N x_{F_{n_f}}^T]^T$
- **alpha** – Chi2 tail probability. Confidence interaval of the individual compatibility test
- **args** – arguments to be passed to the EKFLocalization constructor

h(*xk*)

Observation model for the joint measurements and feature observations:

$$z_k = h(x_k, v_k) \Rightarrow \begin{bmatrix} z_m \\ z_f \end{bmatrix} = \begin{bmatrix} h_m(x_k, v_m) \\ h_f(x_k, v_f) \end{bmatrix} \; ; \; v_k = [v_m^T \; v_f^T]^T \tag{1.65}$$

This method calls `EKF.EKF.hm()` and `MapFeature.MapFeature.hf()` to obtain the expected sensor measurements and the expected feature observations respectively. The method returns an stacked vector of expected measurements and feature observations.

Parameters

xk – mean state vector used as linearization point

Returns

Joint stacked vector of the expected mesurement and feature observations

SquaredMahalanobisDistance(*hfj*, *Pfj*, *zfi*, *Rfi*)

Computes the squared Mahalanobis distance between the expected feature observation hf_j and the feature observation z_{f_i}.

Parameters

- **hfj** – expected feature observation
- **Pfj** – expected feature observation covariance
- **zfi** – feature observation
- **Rfi** – feature observation covariance

Returns

Squared Mahalanobis distance between the expected feature observation hf_j and the feature observation z_{f_i}

IndividualCompatibility(*D2_ij*, *dof*, *alpha*)

Computes the individual compatibility test for the squared Mahalanobis distance D^2_{ij}. The test is performed using the Chi-Square distribution with dof degrees of freedom and a significance level α.

Parameters

- **D2_ij** – squared Mahalanobis distance
- **dof** – number of degrees of freedom
- **alpha** – confidence level

Returns

bolean value indicating if the Mahalanobis distance is smaller than the threshold defined by the confidence level

ICNN(*hf*, *Phf*, *zf*, *Rf*, *dim*)

Individual Compatibility Nearest Neighbor (ICNN) data association algorithm. Given a set of expected feature observations h_f and a set of feature observations z_f, the algorithm returns a pairing hypothesis H that associates each feature observation z_{f_i} with the expected feature observation h_{f_j} that minimizes the Mahalanobis distance D^2_{ij}.

Parameters

- **hf** – vector of expected feature observations
- **Phf** – Covariance matrix of the expected feature observations
- **zf** – vector of feature observations
- **Rf** – Covariance matrix of the feature observations
- **dim** – feature dimensionality

Returns
The vector of asociation hypothesiss

`DataAssociation`(*xk*, *Pk*, *zf*, *Rf*)

Data association algorithm. Given state vector (x_k and P_k) including the robot pose and a set of feature observations z_f and its covariance matrices R_f, the algorithm computes the expected feature observations h_f and its covariance matrices P_f. Then it calls an association algorithms like `ICNN()` (JCBB, etc.) to build a pairing hypothesis associating the observed features z_f with the expected features observations h_f.

The vector of association hypothesis H is stored in the `H` attribute and its dimension is the number of observed features within z_f. Given the j^{th} feature observation z_{f_j}, *self.H[j]=i* means that z_{f_j} has been associated with the i^{th} feature . If *self.H[j]=None* means that z_{f_j} has not been associated either because it is a new observed feature or because it is an outlier.

Parameters

- **xk** – mean state vector including the robot pose
- **Pk** – covariance matrix of the state vector
- **zf** – vector of feature observations
- **Rf** – Covariance matrix of the feature observations

Returns
The vector of asociation hypothesiss

`Localize`(*xk_1*, *Pk_1*)

Localization iteration. Reads the input of the motion model, performs the prediction step (`EKF.EKF.Prediction()`), reads the measurements and the features, solves the data association calling `DataAssociation()` and the performs the update step (`EKF.EKF.Update()`) and logs the results. The method also plots the uncertainty ellipse (`PlotUncertainty()`) of the robot pose, the feature observations and the expected feature observations.

Parameters

- **xk_1** – previous state vector
- **Pk_1** – previous covariance matrix

Return xk, Pk
updated state vector and covariance matrix

`StackMeasurementsAndFeatures`(*zm*, *Rm*, *Hm*, *Vm*, *zf*, *Rf*, *H*)

Given the vector of measurements observations z_m together with their covariance matrix R_m, the vector of feature observations z_f together with their covariance matrix R_f, The measurement observation matrix H_m, the measurement observation noise matrix V_m and the vector of feature associations H, this method returns the joint observation vector z_k, its related covariance matrix R_k, the stacked Observation matrix H_k, the stacked noise observation matrix V_k, the vector of non-paired features z_{np} and its noise covariance matrix R_{np}. It is assumed that the measurements and the features observations are independent, therefore the covariance matrix of the joint observation vector is a block diagonal matrix.

Parameters

- **zm** – measurement observations vector
- **Rm** – covariance matrix of the measurement observations
- **Hm** – measurement observation matrix
- **Vm** – measurement observation noise matrix
- **zf** – feature observations vector
- **Rf** – covariance matrix of the feature observations
- **H** – features associations vector

Returns

vector of joint measurement and feature observations z_k and its covariance matrix R_k

SplitFeatures(*zf*, *Rf*, *H*)

Given the vector of feature observations z_f and their covariance matrix R_f, and the vector of feature associations H, this function returns the vector of paired feature observations z_p together with its covariance matrix R_p, and the vector of non-paired feature observations z_{np} together with its covariance matrix R_{np}. The paired observations will be used to update the filter, while the non-paired ones will be considered as outliers. In the case of SLAM, they become new feature candidates.

Parameters

- **zf** – vector of feature observations
- **Rf** – covariance matrix of feature observations
- **H** – hypothesis of feature associations

Returns

vector of paired feature observations z_p, covariance matrix of paired feature observations R_p, vector of non-paired feature observations z_{np}, covariance matrix of non-paired feature observations R_{np}.

PlotFeatureObservationUncertainty(*zf*, *Rf*)

Plots the uncertainty ellipse of the feature observations. This method is called by *FEKFMBL.PlotUncertainty()*.

Parameters

- **zf** – vector of feature observations
- **Rf** – covariance matrix of the feature observations

PlotExpectedFeaturesObservationsUncertainty()

For all features in the map, this method plots the uncertainty ellipse of the expected feature observations. This method is called by *FEKFMBL.PlotUncertainty()*.

PlotSampleObservationSpace(*NxB*, *BxFj*, *BPFj*, *n*, *color='r.'*)

Plots n samples from a Gaussian distribution with mean x and covariance P. This method is called by *FEKFMBL.PlotUncertainty()*. This is a method for testing. It can be used to compare the uncertainty ellipse with the samples.

Parameters

- **x** – mean of the Gaussian distribution
- **P** – covariance of the Gaussian distribution
- **n** – number of samples
- **color** – color of the samples

PlotSample(*x*, *P*, *n*, *color='r.'*)

Plots n samples from a Gaussian distribution with mean x and covariance P. This method is called by *FEKFMBL.PlotUncertainty()*. This is a method for testing. It can be used to compare the uncertainty ellipse with the samples.

Parameters

- **x** – mean of the Gaussian distribution
- **P** – covariance of the Gaussian distribution
- **n** – number of samples
- **color** – color of the samples

PlotRobotUncertainty()

Plots the robot trajectory and its uncertainty ellipse. This method is called by *FEKFMBL.PlotUncertainty()*.

PlotUncertainty(*zf*, *Rf*)

Plots the uncertainty ellipses of the robot pose (*PlotRobotUncertainty()*), the feature observations (*PlotFeatureObservationUncertainty()*) and the expected feature observations (*PlotExpectedFeaturesObservationsUncertainty()*). This method is called by *FEKFMBL.Localize()* at the end of a localization iteration in order to update the online visualization.

Parameters

- **zf** – vector of feature observations
- **Rf** – covariance matrix of the feature observations

GetRobotState(*xk*)

Returns the robot state from the state vector.

Parameters

xk – mean of the state vector:math:*x_k*

Returns

mean robot state x_{B_k}

SetRobotState(*xk*, *xB*)

Updates the robot state within the state vector.

Parameters

- **xk** – mean of the state vector:math:*x_k*
- **xB** – mean robot state x_{B_k}

Returns

updatd mean state vector x_k

GetRobotStateCovariance(*Pk*)

Returns the robot covariance from the state covariance matrix.

Parameters

Pk – state vector covariance matrix P_k

Returns

robot state covariance P_{B_k}

SetRobotStateCovariance(*Pk*, *PB*)

Updates the robot covariance from the state covariance matrix.

Parameters

- **Pk** – state vector covariance matrix P_k
- **PB** – robot state covariance P_{B_k}

Returns

updatd state covariance matrix P_k

SetRobotPose(*xk*, *xB*)

Updates the robot pose within the state vector.

Parameters

- **xk** – mean of the state vector:math:*x_k*
- **xB** – mean robot pose x_{B_k}

Returns

updatd mean state vector x_k

GetRobotPoseCovariance(*Pk*)

Returns the robot pose covariance from the state covariance matrix.

Parameters

Pk – state vector covariance matrix P_k

Returns

robot pose covariance P_{B_k}

SetRobotPoseCovariance(*Pk*, *PB*)

Updates the robot pose covariance from the state covariance matrix.

Parameters

- **Pk** – state vector covariance matrix P_k
- **PB** – robot pose covariance P_{B_k}

Returns

updated state covariance matrix P_k

4 DOF AUV

4 DOF AUV Map based EKF Localization using an Input Velocity Motion Model with Depht, Yaw and Linear Velocity Measurements, and a 2D Cartesian Feature Observation Model

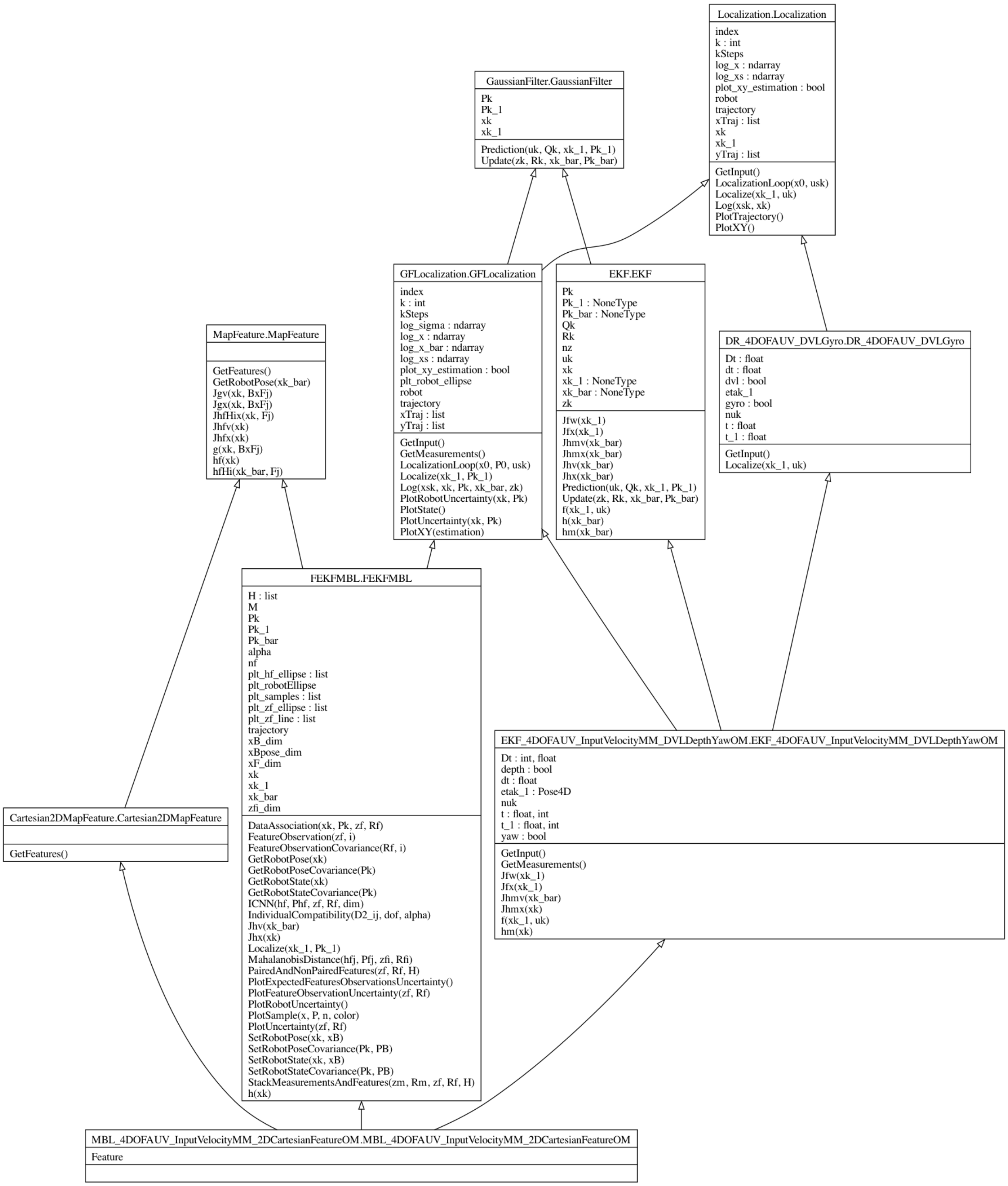
3 DOF Differential Drive Mobile Robot

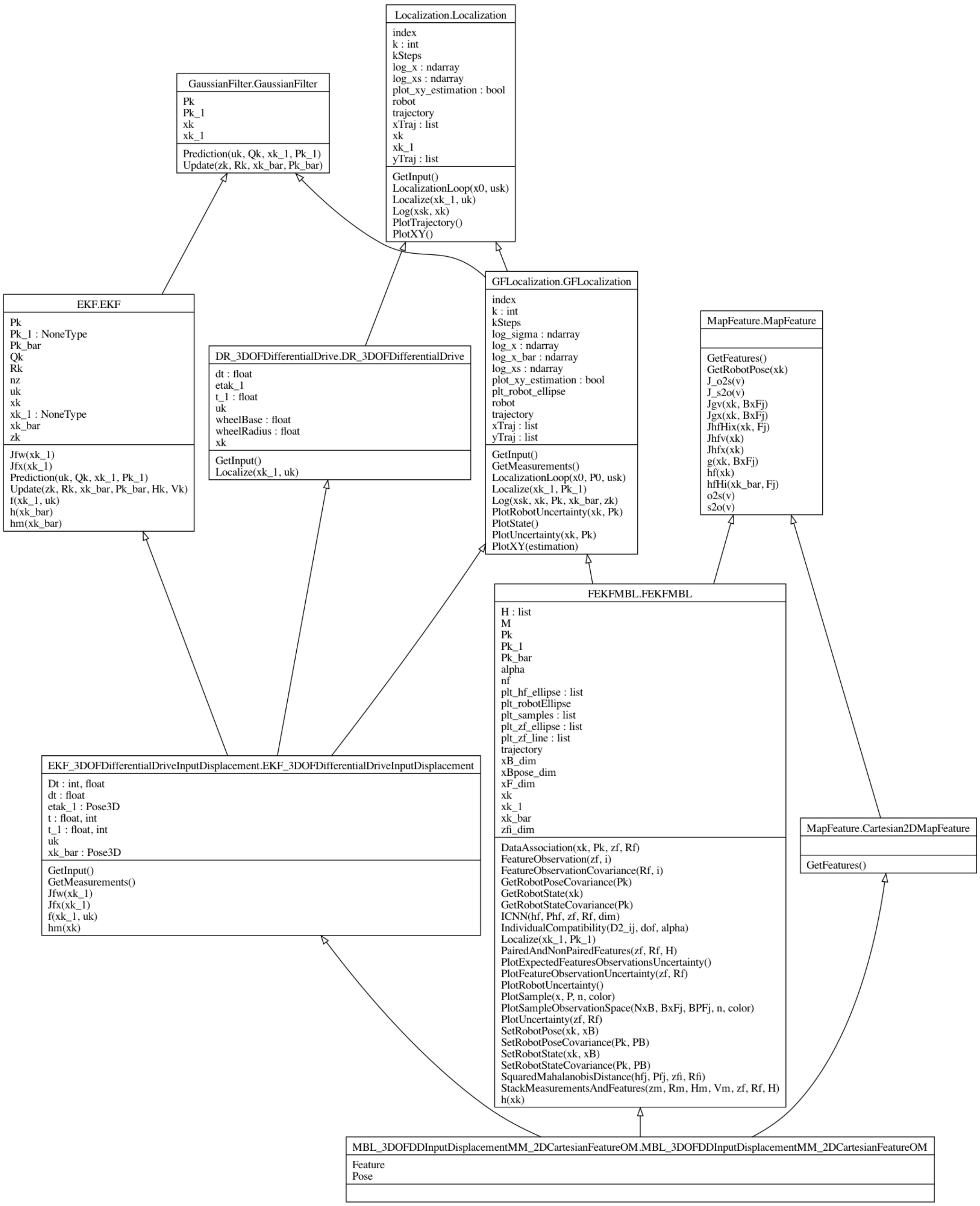
Differential Drive EKF Map Based Localization EKF Using an Input displacement Motion Model and 2D Cartesian Feature Observation Model

class MBL_3DOFDDInputDisplacementMM_2DCartesianFeatureOM.**MBL_3DOFDDInputDisplacementMM_2DCartesianFeatu**

Bases: *Cartesian2DMapFeature*, *FEKFMBL*, *EKF_3DOFDifferentialDriveInputDisplacement*

Feature EKF Map based Localization of a 3 DOF Differential Drive Mobile Robot ($x_k = [^N x_{B_k} \; ^N y_{B_k} \; ^N \psi_{B_k}]^T$) using a 2D Cartesian feature map ($M = [[^N x_{F_1} \; ^N y_{F_1}] \; [x_{F_2} \; ^N y_{F_2}] \; ... \; [^N x_{F_n} \; ^N y_{F_n}]]^T$), and an input displacement motion model (:math:u_k=[^BDeltax_k ^BDelta y_k ^BDelta x psi_k]^T`). The class inherits from


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)
EKF.EKF
Pk
Pk_1 : NoneType
Pk_bar : NoneType
Qk
Rk
nz
uk
xk
xk_1 : NoneType
xk_bar : NoneType
zk
Jfw(xk_1)
Jfx(xk_1)
Jhmv(xk_bar)
Jhmx(xk_bar)
Jhv(xk_bar)
Jhx(xk_bar)
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
f(xk_1, uk)
h(xk_bar)
hm(xk_bar)
MapFeature.MapFeature
GetFeatures()
GetRobotPose(xk_bar)
Jgv(xk, BxFj)
Jgx(xk, BxFj)
JhfHix(xk, Fj)
Jhfv(xk)
Jhfx(xk)
g(xk, BxFj)
hf(xk)
hfHi(xk_bar, Fj)
DR_4DOFAUV_DVLGyro.DR_4DOFAUV_DVLGyro
Dt : float
dt : float
dvl : bool
etak_1
gyro : bool
nuk
t : float
t_1 : float
GetInput()
Localize(xk_1, uk)
FEKFMBL.FEKFMBL
H : list
M
Pk
Pk_1
Pk_bar
alpha
nf
plt_hf_ellipse : list
plt_robotEllipse
plt_samples : list
plt_zf_ellipse : list
plt_zf_line : list
trajectory
xB_dim
xBpose_dim
xF_dim
xk
xk_1
xk_bar
zfi_dim
DataAssociation(xk, Pk, zf, Rf)
FeatureObservation(zf, i)
FeatureObservationCovariance(Rf, i)
GetRobotPose(xk)
GetRobotPoseCovariance(Pk)
GetRobotState(xk)
GetRobotStateCovariance(Pk)
ICNN(hf, Phf, zf, Rf, dim)
IndividualCompatibility(D2_ij, dof, alpha)
Jhv(xk_bar)
Jhx(xk)
Localize(xk_1, Pk_1)
MahalanobisDistance(hfj, Pfj, zfi, Rfi)
PairedAndNonPairedFeatures(zf, Rf, H)
PlotExpectedFeaturesObservationsUncertainty()
PlotFeatureObservationUncertainty(zf, Rf)
PlotRobotUncertainty()
PlotSample(x, P, n, color)
PlotUncertainty(zf, Rf)
SetRobotPose(xk, xB)
SetRobotPoseCovariance(Pk, PB)
SetRobotState(xk, xB)
SetRobotStateCovariance(Pk, PB)
StackMeasurementsAndFeatures(zm, Rm, zf, Rf, H)
h(xk)
EKF_4DOFAUV_InputVelocityMM_DVLDepthYawOM.EKF_4DOFAUV_InputVelocityMM_DVLDepthYawOM
Dt : int, float
depth : bool
dt : float
etak_1 : Pose4D
nuk
t : float, int
t_1 : float, int
yaw : bool
GetInput()
GetMeasurements()
Jfw(xk_1)
Jfx(xk_1)
Jhmv(xk_bar)
Jhmx(xk)
f(xk_1, uk)
hm(xk)
Cartesian2DMapFeature.Cartesian2DMapFeature
GetFeatures()
MBL_4DOFAUV_InputVelocityMM_2DCartesianFeatureOM.MBL_4DOFAUV_InputVelocityMM_2DCartesianFeatureOM
Feature


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
EKF.EKF
Pk
Pk_1 : NoneType
Pk_bar
Qk
Rk
nz
uk
xk
xk_1 : NoneType
xk_bar
zk
Jfw(xk_1)
Jfx(xk_1)
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar, Hk, Vk)
f(xk_1, uk)
h(xk_bar)
hm(xk_bar)
DR_3DOFDifferentialDrive.DR_3DOFDifferentialDrive
dt : float
etak_1
t_1 : float
uk
wheelBase : float
wheelRadius : float
xk
GetInput()
Localize(xk_1, uk)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)
MapFeature.MapFeature
GetFeatures()
GetRobotPose(xk)
J_o2s(v)
J_s2o(v)
Jgv(xk, BxFj)
Jgx(xk, BxFj)
JhfHix(xk, Fj)
Jhfv(xk)
Jhfx(xk)
g(xk, BxFj)
hf(xk)
hfHi(xk_bar, Fj)
o2s(v)
s2o(v)
FEKFMBL.FEKFMBL
H : list
M
Pk
Pk_1
Pk_bar
alpha
nf
plt_hf_ellipse : list
plt_robotEllipse
plt_samples : list
plt_zf_ellipse : list
plt_zf_line : list
trajectory
xB_dim
xBpose_dim
xF_dim
xk
xk_1
xk_bar
zfi_dim
DataAssociation(xk, Pk, zf, Rf)
FeatureObservation(zf, i)
FeatureObservationCovariance(Rf, i)
GetRobotPoseCovariance(Pk)
GetRobotState(xk)
GetRobotStateCovariance(Pk)
ICNN(hf, Phf, zf, Rf, dim)
IndividualCompatibility(D2_ij, dof, alpha)
Localize(xk_1, Pk_1)
PairedAndNonPairedFeatures(zf, Rf, H)
PlotExpectedFeaturesObservationsUncertainty()
PlotFeatureObservationUncertainty(zf, Rf)
PlotRobotUncertainty()
PlotSample(x, P, n, color)
PlotSampleObservationSpace(NxB, BxFj, BPFj, n, color)
PlotUncertainty(zf, Rf)
SetRobotPose(xk, xB)
SetRobotPoseCovariance(Pk, PB)
SetRobotState(xk, xB)
SetRobotStateCovariance(Pk, PB)
SquaredMahalanobisDistance(hfj, Pfj, zfi, Rfi)
StackMeasurementsAndFeatures(zm, Rm, Hm, Vm, zf, Rf, H)
h(xk)
EKF_3DOFDifferentialDriveInputDisplacement.EKF_3DOFDifferentialDriveInputDisplacement
Dt : int, float
dt : float
etak_1 : Pose3D
t : float, int
t_1 : float, int
uk
xk_bar : Pose3D
GetInput()
GetMeasurements()
Jfw(xk_1)
Jfx(xk_1)
f(xk_1, uk)
hm(xk)
MapFeature.Cartesian2DMapFeature
GetFeatures()
MBL_3DOFDDInputDisplacementMM_2DCartesianFeatureOM.MBL_3DOFDDInputDisplacementMM_2DCartesianFeatureOM
Feature
Pose

the following classes: * `Cartesian2DMapFeature`: 2D Cartesian MapFeature using the Catesian coordinates for both, storage and landmark observations. * `FEKFMBL`: Feature EKF Map based Localization class. * `EKF_3DOFDifferentialDriveInputDisplacement`: EKF for 3 DOF Differential Drive Mobile Robot with input displacement motion model.

`__init__`(**args*)

Constructor of the FEKFMBL class.

Parameters

- **`xBpose_dim`** – dimensionality of the robot pose within the state vector
- **`xB_dim`** – dimensionality of the state vector
- **`xF_dim`** – dimentsionality of a feature
- **`zfi_dim`** – dimensionality of a single feature observation
- **`M`** – Feature Based Map $M = [^Nx_{F_1}^T \ ... \ ^Nx_{F_{n_f}}^T]^T$
- **`alpha`** – Chi2 tail probability. Confidence interaval of the individual compatibility test
- **`args`** – arguments to be passed to the EKFLocalization constructor

Differential Drive EKF Map Based Localization EKF Using an Input displacement Motion Model and 2D Feature Store in Cartesina and Observed in Polar Coordinates

class `MBL_3DOFDDInputDisplacementMM_2DCartesianFeatureOM.`**`MBL_3DOFDDInputDisplacementMM_2DCartesianFeatu`**

Bases: *`Cartesian2DMapFeature`*, *`FEKFMBL`*, *`EKF_3DOFDifferentialDriveInputDisplacement`*

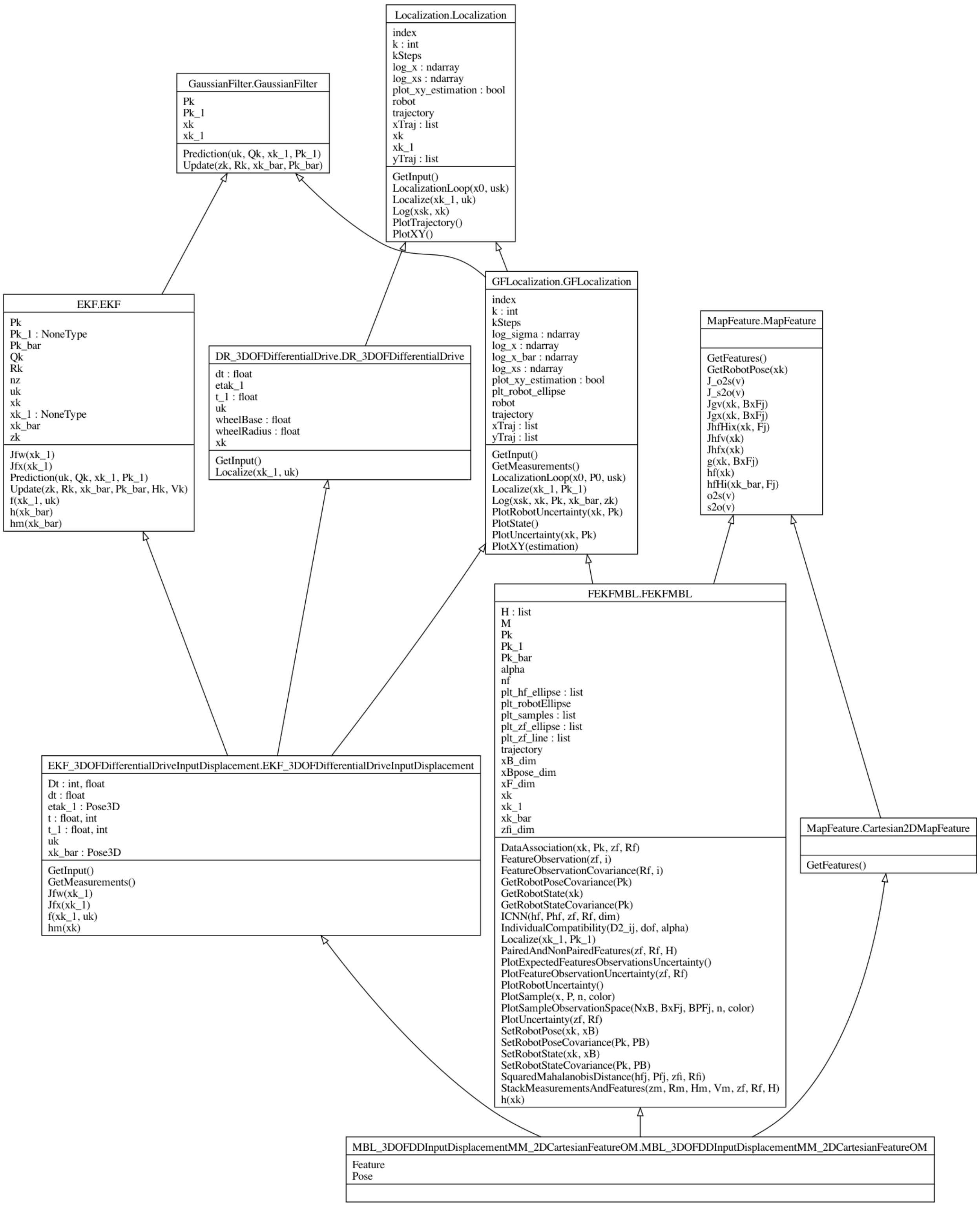
Feature EKF Map based Localization of a 3 DOF Differential Drive Mobile Robot ($x_k = [^Nx_{B_k} \ ^Ny_{B_k} \ ^N\psi_{B_k}]^T$) using a 2D Cartesian feature map ($M = [[^Nx_{F_1} \ ^Ny_{F_1}] \ [x_{F_2} \ ^Ny_{F_2}] \ ... \ [^Nx_{F_n} \ ^Ny_{F_n}]]^T$), and an input displacement motion model (:math:u_k=[^BDeltax_k ^BDelta y_k ^BDelta x psi_k]^T`). The class inherits from the following classes: * `Cartesian2DMapFeature`: 2D Cartesian MapFeature using the Catesian coordinates for both, storage and landmark observations. * `FEKFMBL`: Feature EKF Map based Localization class. * `EKF_3DOFDifferentialDriveInputDisplacement`: EKF for 3 DOF Differential Drive Mobile Robot with input displacement motion model.

`__init__`(**args*)

Constructor of the FEKFMBL class.

Parameters

- **`xBpose_dim`** – dimensionality of the robot pose within the state vector
- **`xB_dim`** – dimensionality of the state vector
- **`xF_dim`** – dimentsionality of a feature
- **`zfi_dim`** – dimensionality of a single feature observation
- **`M`** – Feature Based Map $M = [^Nx_{F_1}^T \ ... \ ^Nx_{F_{n_f}}^T]^T$
- **`alpha`** – Chi2 tail probability. Confidence interaval of the individual compatibility test
- **`args`** – arguments to be passed to the EKFLocalization constructor


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
EKF.EKF
Pk
Pk_1 : NoneType
Pk_bar
Qk
Rk
nz
uk
xk
xk_1 : NoneType
xk_bar
zk
Jfw(xk_1)
Jfx(xk_1)
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar, Hk, Vk)
f(xk_1, uk)
h(xk_bar)
hm(xk_bar)
DR_3DOFDifferentialDrive.DR_3DOFDifferentialDrive
dt : float
etak_1
t_1 : float
uk
wheelBase : float
wheelRadius : float
xk
GetInput()
Localize(xk_1, uk)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)
MapFeature.MapFeature
GetFeatures()
GetRobotPose(xk)
J_o2s(v)
J_s2o(v)
Jgv(xk, BxFj)
Jgx(xk, BxFj)
JhfHix(xk, Fj)
Jhfv(xk)
Jhfx(xk)
g(xk, BxFj)
hf(xk)
hfHi(xk_bar, Fj)
o2s(v)
s2o(v)
FEKFMBL.FEKFMBL
H : list
M
Pk
Pk_1
Pk_bar
alpha
nf
plt_hf_ellipse : list
plt_robotEllipse
plt_samples : list
plt_zf_ellipse : list
plt_zf_line : list
trajectory
xB_dim
xBpose_dim
xF_dim
xk
xk_1
xk_bar
zfi_dim
DataAssociation(xk, Pk, zf, Rf)
FeatureObservation(zf, i)
FeatureObservationCovariance(Rf, i)
GetRobotPoseCovariance(Pk)
GetRobotState(xk)
GetRobotStateCovariance(Pk)
ICNN(hf, Phf, zf, Rf, dim)
IndividualCompatibility(D2_ij, dof, alpha)
Localize(xk_1, Pk_1)
PairedAndNonPairedFeatures(zf, Rf, H)
PlotExpectedFeaturesObservationsUncertainty()
PlotFeatureObservationUncertainty(zf, Rf)
PlotRobotUncertainty()
PlotSample(x, P, n, color)
PlotSampleObservationSpace(NxB, BxFj, BPFj, n, color)
PlotUncertainty(zf, Rf)
SetRobotPose(xk, xB)
SetRobotPoseCovariance(Pk, PB)
SetRobotState(xk, xB)
SetRobotStateCovariance(Pk, PB)
SquaredMahalanobisDistance(hfj, Pfj, zfi, Rfi)
StackMeasurementsAndFeatures(zm, Rm, Hm, Vm, zf, Rf, H)
h(xk)
EKF_3DOFDifferentialDriveInputDisplacement.EKF_3DOFDifferentialDriveInputDisplacement
Dt : int, float
dt : float
etak_1 : Pose3D
t : float, int
t_1 : float, int
uk
xk_bar : Pose3D
GetInput()
GetMeasurements()
Jfw(xk_1)
Jfx(xk_1)
f(xk_1, uk)
hm(xk)
MapFeature.Cartesian2DMapFeature
GetFeatures()
MBL_3DOFDDInputDisplacementMM_2DCartesianFeatureOM.MBL_3DOFDDInputDisplacementMM_2DCartesianFeatureOM
Feature
Pose

Differential Drive EKF Map Based Localization EKF Using an Input displacement Motion Model and 2D Polar Feature Observation Model

class MBL_3DOFDDInputDisplacementMM_2DPolarFeatureOM.**MBL_3DOFDDInputDisplacementMM_2DPolarFeatureOM**(*arg

Bases: PolarMapFeature, *FEKFMBL*, *EKF_3DOFDifferentialDriveInputDisplacement*

Feature EKF Map based Localization of a 3 DOF Differential Drive Mobile Robot ($x_k = [^N x_{B_k}\ ^N y_{B_k}\ ^N \psi_{B_k}\ ]^T$) using a 2D Cartesian feature map ($M = [[^N x_{F_1}\ ^N y_{F_1}]\ [x_{F_2}\ ^N y_{F_2}]\ ...\ [^N x_{F_n}\ ^N y_{F_n}]]^T$), and an input displacement motion model (:math:u_k=[^BDeltax_k ^BDelta y_k ^BDelta x psi_k]^T`). The class inherits from the following classes: * Cartesian2DMapFeature: 2D Cartesian MapFeature using the Catesian coordinates for both, storage and landmark observations. * FEKFMBL: Feature EKF Map based Localization class. * EKF_3DOFDifferentialDriveInputDisplacement: EKF for 3 DOF Differential Drive Mobile Robot with input displacement motion model.

__init__(**args*)

Constructor of the FEKFMBL class.

Parameters

- **xBpose_dim** – dimensionality of the robot pose within the state vector
- **xB_dim** – dimensionality of the state vector
- **xF_dim** – dimentsionality of a feature
- **zfi_dim** – dimensionality of a single feature observation
- **M** – Feature Based Map $M = [^N x_{F_1}^T\ ...\ ^N x_{F_{n_f}}^T]^T$
- **alpha** – Chi2 tail probability. Confidence interaval of the individual compatibility test
- **args** – arguments to be passed to the EKFLocalization constructor

Differential Drive EKF Map Based Localization EKF Using a Ct Velocity Motion Model and 2D Cartesian Feature Observation Model

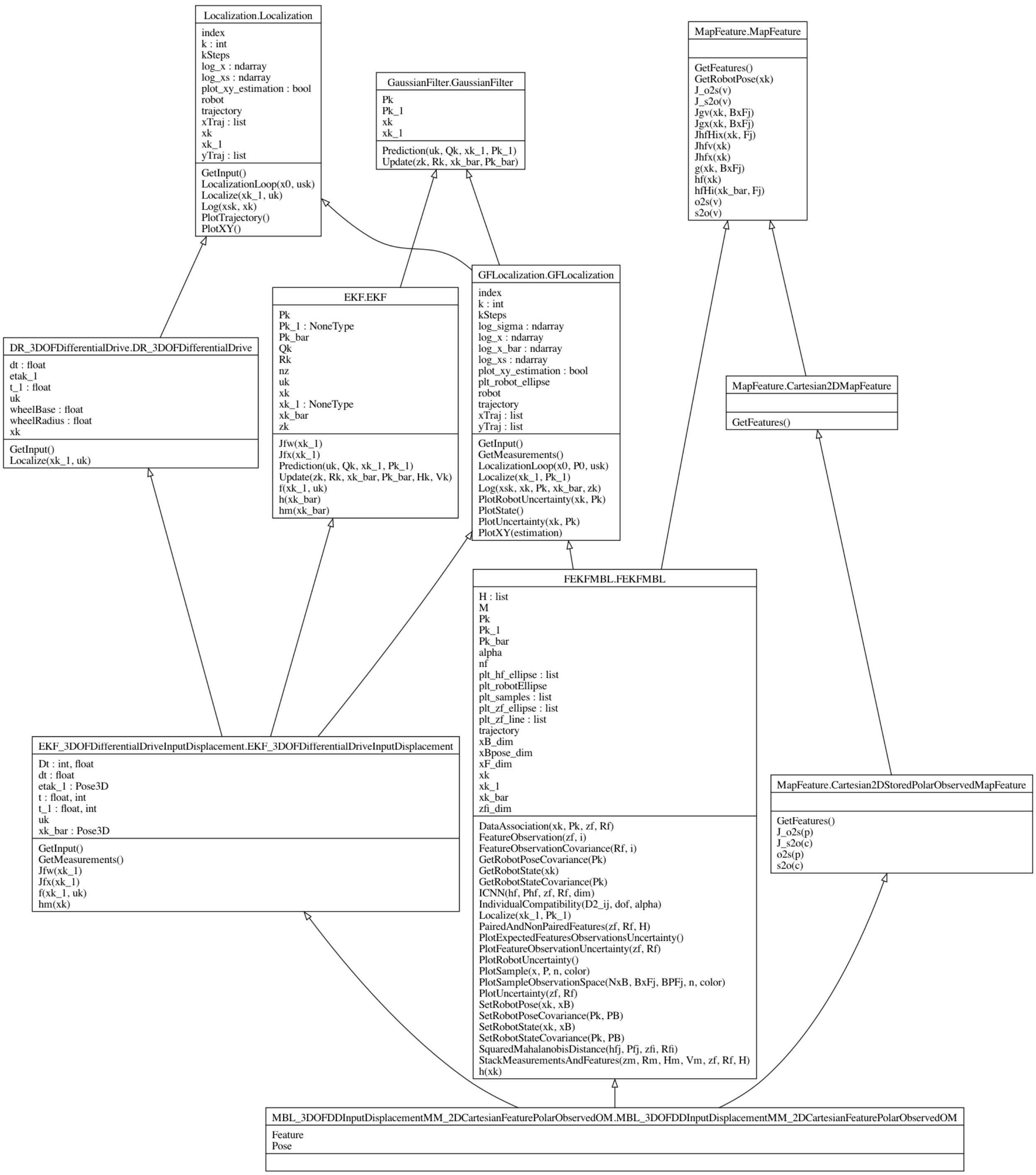
1.7 Simulatenous Localization And Mapping

1.7.1 Feature based EKF Simultaneous Localization And Mapping

Feature based EKF Simultaneous Localization And Mapping

To be completed…


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
EKF.EKF
Pk
Pk_1 : NoneType
Pk_bar
Qk
Rk
nz
uk
xk
xk_1 : NoneType
xk_bar
zk
Jfw(xk_1)
Jfx(xk_1)
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar, Hk, Vk)
f(xk_1, uk)
h(xk_bar)
hm(xk_bar)
DR_3DOFDifferentialDrive.DR_3DOFDifferentialDrive
dt : float
etak_1
t_1 : float
uk
wheelBase : float
wheelRadius : float
xk
GetInput()
Localize(xk_1, uk)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)
MapFeature.MapFeature
GetFeatures()
GetRobotPose(xk)
J_o2s(v)
J_s2o(v)
Jgv(xk, BxFj)
Jgx(xk, BxFj)
JhfHix(xk, Fj)
Jhfv(xk)
Jhfx(xk)
g(xk, BxFj)
hf(xk)
hfHi(xk_bar, Fj)
o2s(v)
s2o(v)
FEKFMBL.FEKFMBL
H : list
M
Pk
Pk_1
Pk_bar
alpha
nf
plt_hf_ellipse : list
plt_robotEllipse
plt_samples : list
plt_zf_ellipse : list
plt_zf_line : list
trajectory
xB_dim
xBpose_dim
xF_dim
xk
xk_1
xk_bar
zfi_dim
DataAssociation(xk, Pk, zf, Rf)
FeatureObservation(zf, i)
FeatureObservationCovariance(Rf, i)
GetRobotPoseCovariance(Pk)
GetRobotState(xk)
GetRobotStateCovariance(Pk)
ICNN(hf, Phf, zf, Rf, dim)
IndividualCompatibility(D2_ij, dof, alpha)
Localize(xk_1, Pk_1)
PairedAndNonPairedFeatures(zf, Rf, H)
PlotExpectedFeaturesObservationsUncertainty()
PlotFeatureObservationUncertainty(zf, Rf)
PlotRobotUncertainty()
PlotSample(x, P, n, color)
PlotSampleObservationSpace(NxB, BxFj, BPFj, n, color)
PlotUncertainty(zf, Rf)
SetRobotPose(xk, xB)
SetRobotPoseCovariance(Pk, PB)
SetRobotState(xk, xB)
SetRobotStateCovariance(Pk, PB)
SquaredMahalanobisDistance(hfj, Pfj, zfi, Rfi)
StackMeasurementsAndFeatures(zm, Rm, Hm, Vm, zf, Rf, H)
h(xk)
EKF_3DOFDifferentialDriveInputDisplacement.EKF_3DOFDifferentialDriveInputDisplacement
Dt : int, float
dt : float
etak_1 : Pose3D
t : int, float
t_1 : int, float
uk
xk_bar : Pose3D
GetInput()
GetMeasurements()
Jfw(xk_1)
Jfx(xk_1)
f(xk_1, uk)
hm(xk)
MapFeature.PolarMapFeature
GetFeatures()
MBL_3DOFDDInputDisplacementMM_2DPolarFeatureOM.MBL_3DOFDDInputDisplacementMM_2DPolarFeatureOM
Feature
Pose


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()
GaussianFilter.GaussianFilter
Pk
Pk_1
xk
xk_1
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar)
MapFeature.MapFeature
GetFeatures()
GetRobotPose(xk)
J_o2s(v)
J_s2o(v)
Jgv(xk, BxFj)
Jgx(xk, BxFj)
JhfHix(xk, Fj)
Jhfv(xk)
Jhfx(xk)
g(xk, BxFj)
hf(xk)
hfHi(xk_bar, Fj)
o2s(v)
s2o(v)
GFLocalization.GFLocalization
index
k : int
kSteps
log_sigma : ndarray
log_x : ndarray
log_x_bar : ndarray
log_xs : ndarray
plot_xy_estimation : bool
plt_robot_ellipse
robot
trajectory
xTraj : list
yTraj : list
GetInput()
GetMeasurements()
LocalizationLoop(x0, P0, usk)
Localize(xk_1, Pk_1)
Log(xsk, xk, Pk, xk_bar, zk)
PlotRobotUncertainty(xk, Pk)
PlotState()
PlotUncertainty(xk, Pk)
PlotXY(estimation)
EKF.EKF
Pk
Pk_1 : NoneType
Pk_bar
Qk
Rk
nz
uk
xk
xk_1 : NoneType
xk_bar
zk
Jfw(xk_1)
Jfx(xk_1)
Prediction(uk, Qk, xk_1, Pk_1)
Update(zk, Rk, xk_bar, Pk_bar, Hk, Vk)
f(xk_1, uk)
h(xk_bar)
hm(xk_bar)
DR_3DOFDifferentialDrive.DR_3DOFDifferentialDrive
dt : float
etak_1
t_1 : float
uk
wheelBase : float
wheelRadius : float
xk
GetInput()
Localize(xk_1, uk)
MapFeature.Cartesian2DMapFeature
GetFeatures()
FEKFMBL.FEKFMBL
H : list
M
Pk
Pk_1
Pk_bar
alpha
nf
plt_hf_ellipse : list
plt_robotEllipse
plt_samples : list
plt_zf_ellipse : list
plt_zf_line : list
trajectory
xB_dim
xBpose_dim
xF_dim
xk
xk_1
xk_bar
zfi_dim
DataAssociation(xk, Pk, zf, Rf)
FeatureObservation(zf, i)
FeatureObservationCovariance(Rf, i)
GetRobotPoseCovariance(Pk)
GetRobotState(xk)
GetRobotStateCovariance(Pk)
ICNN(hf, Phf, zf, Rf, dim)
IndividualCompatibility(D2_ij, dof, alpha)
Localize(xk_1, Pk_1)
PairedAndNonPairedFeatures(zf, Rf, H)
PlotExpectedFeaturesObservationsUncertainty()
PlotFeatureObservationUncertainty(zf, Rf)
PlotRobotUncertainty()
PlotSample(x, P, n, color)
PlotSampleObservationSpace(NxB, BxFj, BPFj, n, color)
PlotUncertainty(zf, Rf)
SetRobotPose(xk, xB)
SetRobotPoseCovariance(Pk, PB)
SetRobotState(xk, xB)
SetRobotStateCovariance(Pk, PB)
SquaredMahalanobisDistance(hfj, Pfj, zfi, Rfi)
StackMeasurementsAndFeatures(zm, Rm, Hm, Vm, zf, Rf, H)
h(xk)
EKF_3DOFDifferentialDriveInputDisplacement.EKF_3DOFDifferentialDriveInputDisplacement
Dt : int, float
dt : float
etak_1 : Pose3D
t : float, int
t_1 : float, int
uk
xk_bar : Pose3D
GetInput()
GetMeasurements()
Jfw(xk_1)
Jfx(xk_1)
f(xk_1, uk)
hm(xk)
MapFeature.Cartesian2DStoredPolarObservedMapFeature
GetFeatures()
J_o2s(p)
J_s2o(c)
o2s(p)
s2o(c)
MBL_3DOFDDInputDisplacementMM_2DCartesianFeaturePolarObservedOM.MBL_3DOFDDInputDisplacementMM_2DCartesianFeaturePolarObservedOM
Feature
Pose

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

D

E

I

J

K

L

M

O

P

R

S

T

U

V