InEKF

Release 0.1.0

Easton Potokar

May 02, 2022

INEKF DOCUMENTATION

InEKF is a C++ library with python bindings that implements the Invariant Extend Kalman Filter (InEKF) in a modular to enable easy application to any system.

CHAPTER

ONE

FEATURES

- Support for Right & Left filters.
- Base classes provide easy extension via inheritance.
- Coded using static Eigen types for efficient structure.
- Fully featured python interface for use in classroom, prototyping, etc.
- C++14 and above supported.
- Fullly templated Lie Groups SO2, SO3, SE2, SE3 to enable additional tracking of Euclidean states and multiple extra columns in SE2/SE3.
- Dynamic Lie Groups types to add columns to SE2/SE3 on the fly (for InEKF SLAM and others).
- Various examples to get started.

1.1 Installation

Installation is straightforward in either language.

C++

Python

Installation of library in C++ requires manually building from source using CMake. The only dependency is Eigen, and if you want to build the python binding, pybind11 as well. If not found on the system, both of these will be pulled from their respective git repos, and built locally.

A simple build will look like:

```
mkdir build
cd build
cmake ..
make
sudo make install
```

Tests, examples, and the python binding can all be enabled with the following options when running cmake. Note all are disabled by default.

```
cmake .. -DTESTS=ON -DPYTHON=ON -DEXAMPLES=ON
```

Note: By default, cmake is set to debug mode which can significantly slow things down, it can be set to release mode by passing `-DCMAKE_BUILD_TYPE=Release` to cmake.

After installation, the library can be linked against using the following lines in cmake

```
find_package(Eigen3 CONFIG REQUIRED)
find_package(InEKF CONFIG REQUIRED)
target_link_libraries(mytarget PUBLIC InEKF::Core InEKF::Inertial InEKF::SE2Models)
```

Alternatively, cmake can be configured to pull InEKF directly from git

```
FetchContent_Declare(
  InEKF
  GIT_REPOSITORY git@bitbucket.org:frostlab/inekf.git
  GIT_TAG        v0.1.0
  )
FetchContent_MakeAvailable(InEKF)
```

The above options can be added into this line as well, for example `option(PYTHON ON)` right above the `FetchContent_MakeAvailable(InEKF)` line. When installed from source, the package can be removed using `sudo make remove`.

Python installation is the easiest:) Just a single line to install from pip

```
pip install inekf
```

1.2 Getting Started

The InEKF library has been designed to be straightforward and simple to use, while still being versatile.

First, we must import the library, and any dependencies.

C++

Python

```
#include <Eigen/Core>
#include <InEKF/Core>
#include <InEKF/SE2Models>
```

```
import numpy as np
import inekf
```

Note: TODO: Insert citation to InEKF tutorial here for more info about things!

1.2.1 Using Lie Groups

The Lie groups in the library are heavily templated to allow for simple usage of any Lie group. The special orthogonal groups *SO* are templated to allow tracking additional Euclidean states alongside the group (defaults to 0), and the special Euclidean groups *SE* are templated to allow the additional Euclidean states along with any number of positional columns (defaults to 1).

We've also repurposed the bracket `[]` in python to allow for a near identical usage across APIs. If using C++17 or python, these can templates can be omitted if using the defaults. We assume throughout C++17 is used, and ommit the empty <>.

C++

Python

```
// Two columns, one Euclidean state
InEKF::SE2<2,1> x1();

// If using C++17
// One column, no Euclidean states
// 0 rotation, 1 for x and y, covariance of identities
InEKF::SE2 x2(0, 1, 1, Eigen::Matrix3d::Identity());

// If using older standard of C++
InEKF::SE2<> x3(0, 1, 1, Eigen::Matrix3d::Identity());
```

```
# Two columns, one Euclidean state
x1 = inekf.SE2[2, 1]()

# One column, no Euclidean states
# 0 rotation, 1 for x and y, covariance of identities
x2 = inekf.SE2(0, 1, 1, np.eye(3))
```

They each have a number of constructors, see C++ *Lie Groups* and Python *Lie Groups* for more details.

Note: If no covariance is passed to the constructor, the state is assumed to be "certain" and no covariance is set. If you wish to have covariance tracked (necessary for use in InEKF), make sure you set this.

Further, each group is equipped with a number of common operations, such as

- Inverse
- Group action (multiplication)
- Wedge ^ operator
- Exp/Log
- Adjoint

Along with these is overloading of `()` to return the state matrix, and `[]` to retrieve a specific column.

C++

Python

```
InEKF::SE2 x(0,1,2);
InEKF::SE2 y(3,4,5);
Eigen::Vector3d xi{1,2,3};

// Group methods
x.inverse();
x*y;
x.log();
x.Ad();

// Static methods
InEKF::SE2::wedge(xi);
InEKF::SE2::exp(xi);
InEKF::SE2::log(x);
InEKF::SE2::Ad(x);

// Getters
x();
x.mat();
// SO2 object
x.R();
// Vector 1,2
x[0];
// Covariance
x.cov();
// Get additional Euclidean states
x.aug();
```

```
x = inekf.SE2(0,1,2)
y = inekf.SE2(3,4,5)
xi = np.array([1,2,3])

# Group methods
x.inverse
~x # Same as above
x*y
x@y # Same as above
x.log
x.Ad

# Static methods
inekf.SE2.wedge(xi)
inekf.SE2.exp(xi)
inekf.SE2.log_(x)
inekf.SE2.Ad_(x)

# Getters
x.mat
# SO2 object
x.R
# Vector 1,2
x[0]
```

(continues on next page)

(continued from previous page)

```
# Covariance
x.cov
# Get additional Euclidean states
x.aug
```

1.2.2 Making Models

Next, process and measurement models must be made. You'll likely need a custom process model done via inheritance, for this see *Custom Models*. You can also customize measurement models (see same link), but the built in is robust enough for most purposes.

For example, here's creation of a simple odometry model in *SE(2)*,

C++

Python

```
// Set the covariance of theta (rad), x, y
InEKF::OdometryProcess pModel(0.001, 0.05, 0.05);
```

```
# Set the covariance of theta (rad), x, y
pModel = inekf.OdometryProcess(0.001, 0.05, 0.05)
```

An invariant measurement model is either a left $Xb + w$ or right $X^{-1}b + w$. The invariant model is then defined by this *b* vector, the covariance of *w*, and whether it's a right or a left measurement. The linearized innovation matrix *H* is then automatically created. For example, we'll set up a GPS sensor in *SE(2)*, which is left invariant,

C++

Python

```
// Make b vector
Eigen::Vector3d b{0, 0, 1};

// Make covariance
Eigen::Matrix2d M = Eigen::Matrix2d::Identity()*.01;

// Make model
InEKF::MeasureModel<InEKF::SE2> gps(b, M, InEKF::ERROR::LEFT);
```

```
# Make b vector
b = np.array([0, 0, 1])

# Make covariance
M = np.eye(2)*.01;

# Make model
gps = inekf.MeasureModel[inekf.SE2](b, M, inekf.ERROR.LEFT)
```

Or similarly, a compass measuring exactly true north, which is right invariant,

C++

Python

```
// Make b vector
Eigen::Vector3d b{1, 0, 0};

// Make covariance
Eigen::Matrix2d M = Eigen::Matrix2d::Identity()*.01;

// Make model
InEKF::MeasureModel<InEKF::SE2> compass(b, M, InEKF::RIGHT);
```

```
# Make b vector
b = np.array([1, 0, 0])

# Make covariance
M = np.eye(2)*.01

# Make model
compass = inekf.MeasureModel[inekf.SE2](b, M, inekf.ERROR.RIGHT)
```

1.2.3 Making & Using the InEKF

Finally, we make the InEKF. The InEKF takes 3 arguments in its constructor: the process model, an initial estimate, and whether to run a right or left InEKF.

C++

Python

```
// Make initial estimate
Eigen::Matrix3d cov = Eigen::Matrix3d::Identity()*.1;
InEKF::SE2 x0(0, 0, 0, cov);

// Make Right InEKF
InEKF::InEKF iekf(&pModel, x0, InEKF::RIGHT);
iekf.addMeasureModel("gps", &gps);
iekf.addMeasureModel("compass", &compass);
```

```
# Make initial estimate
cov = np.eye(3)*.1
x0 = inekf.SE2(0, 0, 0, cov)

# Make Right InEKF
iekf = inekf.InEKF(pModel, x0, inekf.ERROR.RIGHT)
iekf.addMeasureModel("gps", gps)
iekf.addMeasureModel("compass", compass)
```

Using the predict and update steps is just as easy (we make fake data here to use). While technically an invariant measurement will have extra ones or zeros on the end, the `MeasureModel` class will take care of appending these when needed. This steps are generally done in a loop and are executed when data is received. After each step is ran it will return the corresponding state estimate which can also be accessed using `getState` in C++ or the `state` property in python.

C++

Python

```cpp
InEKF::SE2 state;

// Prediction step with some control U
InEKF::SE2 U(.1, .1, .1);
// Predict also takes an optional dt, which may or may not
// be used, depending on the process model (not needed in this case)
state = iekf.predict(U);

// Update gps with location measurement = 1,1
// We include the needed one here as well
Eigen::Vector3d z_gps{1, 1, 1};
// Updates with name we put in before
state = iekf.update("gps", z_gps);

// Update compass with measurement = 1, 0
// Model appends the extra 0 is this case
Eigen::Vector2d z_compass{1, 0};
state = iekf.update("compass", z_compass);

// Get most recent state
state = iekf.getState();
```

```python
# Prediction step with some control U
U = inekf.SE2(.1, .1, .1)
# Predict also takes an optional dt, which may or may not
# be used, depending on the process model (not needed in this case)
state = iekf.predict(U)

# Update gps with location measurement = 1,1
# We include the needed one here as well
z_gps = np.array([1, 1, 1])
# Updates with name we put in before
state = iekf.update("gps", z_gps)

# Update compass with measurement = 1, 0
# Model appends the extra 0 is this case
z_compass = np.array([1, 0])
state = iekf.update("compass", z_compass)

# Get most recent state
state = iekf.state
```

More examples can be found in the bitbucket repository, both in C++ and python.

1.3 Custom Models

InEKF is set up so your process/measure models will be an easy extension and continue to function with `InEKF` and `LieGroups` if defined properly. Note this can be done in python or C++. The following is what must be defined/done to successfully do this. The following methods/variables for each base class must be implemented/set

1.3.1 MeasureModel

All methods are already implemented in the `MeasureModel` class, so the base class can be used in most scenarios. This means when you do want to make a custom measurement class, which methods to override can be decided on a case by case basis. The `MeasureModel` constructor takes in the vector `b`, covariance `M`, and type of error and from these makes `H` accordingly. It is also templated by the type of group that it is defined on.

If you decide to override, make sure you call the base class constructor and set the error, or set the first 4 values of the following, otherwise they default to all zeros.

Method	Use
*error_*	Type of invariant measurement, of type *InEKF::ERROR*.
*M_*	Noise parameter. A default should be set in the constructor, and possible a method made to set it
*b_*	b vector in invariant measurement model. Can be used to set *H_* through *setHandb()* and if the first few elements are nonzero, is needed in *calcV()*
*H_*	Linearized innovation matrix `H`. Can be set manually or from Will be hit with adjoint depending on type of filter.
processZ()	Any preprocessing that needs to be done on z should be done here. This could include adding 0s and 1s on the end, change of frames, etc. Returns z.
makeHError()	Shifts *H_* by the adjoint, and saves it in *H_error_* and returns it. Likely will not need to be overriden.
calcV()	Accepts an exact size of z, and calculates/returns the innovation. Likely will not need to be overriden.
calcSInverse()	Calculates and returns S^{-1}, the inverse of the measurement covariance. Also likely won't need to be overriden. Use *H_error_* here.

Building a custom `SE(2)` measure model in C++ and python will look something like the following.

C++

Python

```
class MySensor : public InEKF::MeasureModel<InEKF::SE2<1,0>> {}
```

```
class MySensor(inekf.MeasureModel[inekf.SE2[1,0]]):
    pass
```

And then override functions as needed. For examples see the Inertial Models in C++ and the Underwater Inertial from scratch script in python.

Note: In python `error_`, `M_`, and `H_` are named `error`, `M`, and `H`, respectively. Further note, due to how the python bindings function, you *can not* modify `M` and `H` in place, they must be written as a whole.

As a reference, here's what these functions will be used for in update step of the InEKF.

```
// Do any preprocessing on z (fill it up, frame changes, etc)
VectorB z_ = m_model->processZ(z, state_);;
```

(continues on next page)

(continued from previous page)

```
// Change H via adjoint if necessary
MatrixH H = m_model->makeHError(state_, error_);

// Use measurement model to make Sinv and V
VectorV V = m_model->calcV(z_, state_);
MatrixS Sinv = m_model->calcSInverse(state_);

// Caculate K + dX
MatrixK K = state_.cov() * (H.transpose() * Sinv);
TangentVector K_V = K * V;
```

1.3.2 ProcessModel

In contrast, the process model implements a few things that MUST be overriden. It is templated by both the group it is defined on, as well as the control input that is taken in.

Method	Use
f()	State process model. Returns the state.
makePhi()	Creates exp(A*dt) to use. Make sure to check what type of error State is and make A accordingly
*Q_*	Noise parameter. A default should be set in the constructor, and possible a method made to set it

Building a custom `SE(2)` process model with a 3-vector as controls in C++ and python will look something like the following.

C++

Python

```
class MyProcess : public ProcessModel<SE3<1,0>, Eigen::Vector3d> {

    public:
        MyProcess(MatrixCov Q) {Q_ = Q};
        ~MyProcess(){}
        SE3<1,0> f(Eigen::Vector3d u, double dt, SE3<1,0> state) override;
        MatrixCov makePhi(const Eigen::Vector3d& u, double dt, const SE3<1,0>& state,␣
↪ERROR error) override;

};
```

```
class MyProcess(inekf.ProcessModel[inekf.SE2[1,0], "Vec3"]):
    def __init__(self, Q):
        self.Q = Q

    def f(self, u, dt, state):
        # Your implementation here
        pass

    def makePhi(self, u, dt, state, error):
        # Your implementation here
        pass
```

For examples see the Inertial Process model in C++ and the Underwater Inertial from scratch script in python.

Note: Just like in the measure model case, here `Q_` is actually named `Q` on the python side.. Again, due to how the python bindings function, you *can not* modify `Q` in place, it must be written as a whole.

1.4 Changelog

1.4.1 InEKF 0.1.0

5/2/22

First release!

Highlights

- Fully fleshed out template Lie group structure
- Written documentation for C++ and python
- Easily extensible process and measurement models in both C++ and python
- Victoria Park SLAM example

New Features

- Everything! :)

Breaking Changes

- If you were running a dev version previously, a lot of syntax has changed.
- This should be fairly easy to adjust however, with just a few changed names that should be fairly obvious.

Bug Fixes

- N/A

1.5 Core Classes

1.5.1 Error

enum `InEKF::`**`ERROR`**

Type of invariant error. Has options for left or right.

Values:

enumerator **LEFT**

enumerator **RIGHT**

1.5.2 Invariant Extended Kalman Filter

template<class **pM**>

class *InEKF*::**InEKF**

The Invariant Extended Kalman Filter.

Template Parameters **pM** – Process Model. Pulls group and control info from it, can be left out if class template argument deduction (C++17) is used.

Public Functions

inline **InEKF**(*pM* *pModel, Group state, *ERROR* error = *ERROR*::*RIGHT*)

Construct a new *InEKF* object.

Parameters

- **pModel** – Pointer to the process model.
- **state** – Initial state, must be of same group that process model uses, and must be uncertain.
- **error** – Right or left invariant error.

Group **predict**(const U &u, double dt = 1)

Prediction Step.

Parameters

- **u** – Control, must be same as what process model uses.
- **dt** – Delta t. Used sometimes depending on process model. Defaults to 1.

Returns State estimate

Group **update**(std::string name, const Eigen::VectorXd &z)

Update Step.

Parameters

- **name** – Name of measurement model.
- **z** – Measurement. May vary in size depending on how measurement model processes it.

Returns State estimate.

void **addMeasureModel**(std::string name, *MeasureModel*<Group> *m)

Add measurement model to the filter.

Parameters

- **name** – Name of measurement model.
- **m** – A measure model pointer, templated by the used group.

void **addMeasureModels**(std::map<std::string, *MeasureModel*<Group>*> m)

Add multiple measurement models to the filter.

Parameters **m** – Map from model names to model. Can be used passed in as {“name”: model, “another”: diff_model}

inline const Group &**getState**() const

Get the current state estimate.

Returns const Group&

inline void **setState**(const Group &state)

Set the current state estimate.

Parameters **state** – Current state estimate

1.5.3 Measure Model

template<class **Group**>

class InEKF::**MeasureModel**

Base class measure model. Written to be inherited from, but in most cases this class will be sufficient.

Template Parameters **Group** – State’s group that is being tracked.

Public Types

typedef Eigen::Matrix<double, *Group*::rotSize, *Group*::rotSize> **MatrixS**

Size of matrix needed for the measurement model covariance.

typedef Eigen::Matrix<double, *Group*::rotSize, *Group*::N> **MatrixH**

Size of matrix needed for linearized measurement model.

typedef Eigen::Matrix<double, *Group*::rotSize, 1> **VectorV**

Size of vector for truncated innovation.

typedef Eigen::Matrix<double, *Group*::M, 1> **VectorB**

Size of vector for full measurement size.

Public Functions

inline **MeasureModel**()

Construct a new Measure Model object.

inline **MeasureModel**(*VectorB* b, const *MatrixS* &M, *ERROR* error)

Construct a new Measure Model object, automatically creating H. Should be used most of the time.

Parameters

- **b** – b vector from measurement model. Will be used to create H.
- **M** – Measurement covariance.

- **error** – Type of invariant measurement (right or left).

inline virtual *VectorB* **processZ**(const Eigen::VectorXd &z, const *Group* &state)

Process measurement before putting into *InEKF*. Can be used to change frames, convert r/b->x/y, or append 0s. By default is used to append zeros/ones onto it according to b vector set. Called first in update step.

Parameters

- **z** – Measurement
- **state** – Current state estimate.

Returns Processed measurement.

inline virtual *MatrixH* **makeHError**(const *Group* &state, *ERROR* iekfERROR)

Sets and returns H_error_ for settings where filter error type != measurement error type. Done by multiplying H by adjoint of current state estimate. Called second in update step.

Parameters

- **state** – Current state estimate.
- **iekfERROR** – Type of filter error.

Returns H_error_

inline virtual *VectorV* **calcV**(const *VectorB* &z, const *Group* &state)

Computes innovation based on measurement model. Called third in the update step.

Parameters

- **z** – Measurement.
- **state** – Current state estimate.

Returns Truncated innovation.

inline virtual *MatrixS* **calcSInverse**(const *Group* &state)

Calculate inverse of measurement noise S, using H_error_. Called fourth in the update step.

Parameters **state** – Current state estimate.

Returns Inverse of measurement noise.

inline *MatrixH* **getH**()

Gets linearized matrix H.

Returns MatrixH

inline *ERROR* **getError**()

Get the measurement model error type.

Returns ERROR

inline void **setHandb**(*VectorB* b)

Sets measurement vector b and recreates H accordingly. Useful if vector b isn't constant.

Parameters **b** – Measurement model b

Protected Attributes

ERROR **error_**

Type of error of the filter (right/left)

MatrixS **M_** = *MatrixS*::Identity(*Group*::rotSize, *Group*::rotSize)

Measurement covariance.

VectorB **b_** = *VectorB*::Zero(*Group*::m, 1)

b vector used in measure model.

MatrixH **H_** = *MatrixH*::Zero(*Group*::rotSize, *Group*::c)

Linearized H matrix. Will be automatically created from b in constructor unless overriden.

MatrixH **H_error_**

This is the converted H used in *InEKF* if it's a right filter with left measurement or vice versa. Used in calcSInverse if overriden.

1.5.4 Process Model

template<class **Group**, class **U**>

class InEKF::**ProcessModel**

Base class process model.

Template Parameters

- **Group** – State's group that is being tracked.
- **U** – Form of control. Can be either a group, or an Eigen::Matrix<double,n,1>

Public Types

typedef *Group*::MatrixCov **MatrixCov**

The covariance matrix has size NxN, where N is the state dimension.

typedef *Group*::MatrixState **MatrixState**

A group element is a matrix of size MxM.

typedef *Group* **myGroup**

Renaming of Group template, used by the *InEKF*.

typedef *U* **myU**

Renaming of U template, used by the *InEKF*.

Public Functions

inline **ProcessModel**()

Construct a new Process Model object.

inline virtual *Group* **f**(*U* u, double dt, *Group* state)

Propagates state forward one timestep. Must be overriden, has no implementation.

Parameters

- **u** – Control
- **dt** – Delta time
- **state** – Current state

Returns Updated state estimate

inline virtual *MatrixCov* **makePhi**(const *U* &u, double dt, const *Group* &state, *ERROR* error)

Make a discrete time linearized process model matrix, with $\Phi = \exp(A\Delta t)$. Must be overriden, has no implementation.

Parameters

- **u** – Control
- **dt** – Delta time
- **state** – Current state estimate (shouldn't be needed unless doing an "Imperfect InEKF")
- **error** – Right or left error. Function should be implemented to handle both.

Returns Phi

inline *MatrixCov* **getQ**() const

Get process model covariance.

Returns Q

inline void **setQ**(*MatrixCov* Q)

Set process model covariance.

Parameters **Q** –

Protected Attributes

MatrixCov **Q_**

Process model covariance.

1.6 Lie Groups

1.6.1 SO(2)

template<int **A** = 0>

class `InEKF::SO2` : public InEKF::*LieGroup*<*SO2*<*A*>, *calcStateDim*(2, 0, *A*), 2, *A*>

2D rotational states, also known as the 2x2 special orthogonal group, SO(2).

Template Parameters `A` – Number of augmented Euclidean states. Can be Eigen::Dynamic if desired. Defaults to 0.

Public Functions

inline `SO2`(const MatrixState &State = MatrixState::Identity(), const MatrixCov &Cov = MatrixCov::Zero(c, c), const VectorAug &Aug = VectorAug::Zero(a))

Construct *SO2* object with all available options.

Parameters

- `State` – A 2x2 Eigen matrix. Defaults to the identity.
- `Cov` – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.
- `Aug` – Additional euclidean states if A != 0. Defaults to 0s.

inline `SO2`(const *SO2* &State)

Copy constructor. Initialize with another *SO2* object.

Parameters `State` – *SO2* object. The matrix, covariance and augmented state will all be copied from it.

inline `SO2`(double theta, const MatrixCov &Cov = MatrixCov::Zero(c, c), const VectorAug &Aug = VectorAug::Zero(a))

Construct a new *SO2* object using an angle.

Parameters

- `theta` – Angle of rotation matrix in radians.
- `Cov` – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.
- `Aug` – Additional euclidean states if A != 0. Defaults to 0s.

inline `SO2`(const TangentVector &xi, const MatrixCov &Cov = MatrixCov::Zero(c, c))

Construct a new *SO2* object from a tangent vector using the exponential operator.

Parameters

- `xi` – Tangent vector of size (1 + Augmented state size).
- `Cov` – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.

inline `~SO2`()

Destroy the *SO2* object.

inline *SO2* `R`() const

Gets rotational component of the state. In the *SO2* case, this is everything except the augmented Euclidean states and covariance.

Returns SO2<> Rotational component of the state.

void `addAug`(double x, double sigma = 1)

Adds an element to the augmented Euclidean state. Only usable if A = Eigen::Dynamic.

Parameters

- **x** – Variable to add.
- **sigma** – Covariance of element. Only used if state is uncertain.

SO2<*A*> **inverse**() const

Invert state.

Returns Inverted matrix (transpose). Augmented portion and covariance is dropped.

SO2<*A*> **operator***(const *SO2*<*A*> &rhs) const

Combine rotations. Augmented states are summed.

Parameters **rhs** – Right hand element of multiplication.

Returns Combined elements with same augmented size.

Public Static Functions

static MatrixState **wedge**(const TangentVector &xi)

Move element in R^n to the Lie algebra.

Parameters **xi** – Tangent vector

Returns MatrixState Element of Lie algebra

static *SO2* **exp**(const TangentVector &xi)

Move an element from R^n -> algebra -> group.

Parameters **xi** – Tangent vector

Returns Element of *SO2*

static TangentVector **log**(const *SO2* &g)

Move an element from group -> algebra -> R^n.

Parameters **g** – Group element

Returns TangentVector

static MatrixCov **Ad**(const *SO2* &g)

Compute the linear map Adjoint.

Parameters **g** – Element of *SO2*

Returns Matrix of size state dimension x state dimension

Public Static Attributes

static constexpr int **rotSize** = 2

Size of rotational component of group.

static constexpr int **N** = *calcStateDim*(*rotSize*, 0, *A*)

Dimension of group.

static constexpr int **M** = *calcStateMtxSize*(*rotSize*, 0)

State will have matrix of size M x M.

`static constexpr int a = A == Eigen::Dynamic ? 0 : A`

Handles defaults values of augmented sizes when A is Eigen::Dyanmic.

`static constexpr int c = A == Eigen::Dynamic ? 1 : N`

Handles defaults values of tangent vector sizes when A is Eigen::Dyanmic.

static constexpr int **m** = *M*

Handles defaults values of matrix sizes.

1.6.2 SE(2)

template<int **C** = 1, int **A** = 0>

class InEKF::**SE2** : public InEKF::*LieGroup*<*SE2*<*C*, *A*>, *calcStateDim*(2, *C*, *A*), *calcStateMtxSize*(2, *C*), *A*>

2D rigid body transformation, also known as the 3x3 special Euclidean group, SE(2).

Template Parameters

- **C** – Number of Euclideans columns to include. Can be Eigen::Dynamic. Defaults to 1.
- **A** – Number of augmented Euclidean states. Can be Eigen::Dynamic if desired. Defaults to 0.

Public Functions

inline **SE2**(const MatrixState &State = MatrixState::Identity(m, m), const MatrixCov &Cov = MatrixCov::Zero(c, c), const VectorAug &Aug = VectorAug::Zero(a, 1))

Construct *SE2* object with all available options.

Parameters

- **State** – An MxM Eigen matrix. Defaults to the identity.
- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.
- **Aug** – Additional euclidean states if A != 0. Defaults to 0s.

inline **SE2**(const *SE2* &State)

Copy constructor. Initialize with another *SE2* object.

Parameters **State** – *SE2* object. The matrix, covariance and augmented state will all be copied from it.

SE2(const TangentVector &xi, const MatrixCov &Cov = MatrixCov::Zero(c, c))

Construct a new *SE2* object from a tangent vector using the exponential operator.

Parameters

- **xi** – Tangent vector of size (1 + 2*Columns + Augmented state size).
- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.

inline **SE2**(double theta, double x, double y, const MatrixCov &Cov = MatrixCov::Zero(c, c))

Construct a new *SE2* object using an theta, x, y values. Only works if C=1.

Parameters

- **theta** – Angle of rotate in radians.
- **x** – X-distance
- **y** – Y-distance
- **Cov** – Covariance of state. If not input, state is set as “certain” and covariance is not tracked.

inline **~SE2**()

Destroy the *SE2* object.

inline *SO2* **R**() const

Gets rotational component of the state.

Returns SO2<> Rotational component of the state.

inline Eigen::Vector2d **operator[]**(int idx) const

Gets the ith positional column of the group.

Parameters **idx** – Index of column to get, from 0 to C-1.

Returns Eigen::Vector2d

void **addCol**(const Eigen::Vector2d &x, const Eigen::Matrix2d &sigma = Eigen::Matrix2d::Identity())

Adds a column to the matrix state. Only usable if C = Eigen::Dynamic.

Parameters

- **x** – Column to add in.
- **sigma** – Covariance of element. Only used if state is uncertain.

void **addAug**(double x, double sigma = 1)

Adds an element to the augmented Euclidean state. Only usable if A = Eigen::Dynamic.

Parameters

- **x** – Variable to add.
- **sigma** – Covariance of element. Only used if state is uncertain.

SE2 **inverse**() const

Invert state.

Returns Inverted matrix. Augmented portion and covariance is dropped.

SE2 **operator***(const *SE2* &rhs) const

Combine transformations. Augmented states are summed.

Parameters **rhs** – Right hand element of multiplication.

Returns Combined elements with same augmented size.

Public Static Functions

static MatrixState **wedge**(const TangentVector &xi)

Move element in R^n to the Lie algebra.

Parameters **xi** – Tangent vector

Returns MatrixState Element of Lie algebra static *SE2* **exp**(const TangentVector &xi)

Move an element from R^n -> algebra -> group.

Parameters **xi** – Tangent vector

Returns Element of *SE2*

static TangentVector **log**(const *SE2* &g)

Move an element from group -> algebra -> R^n.

Parameters **g** – Group element

Returns TangentVector

static MatrixCov **Ad**(const *SE2* &g)

Compute the linear map Adjoint.

Parameters **g** – Element of *SE2*

Returns Matrix of size state dimension x state dimension

Public Static Attributes

static constexpr int **rotSize** = 2

Size of rotational component of group.

static constexpr int **N** = *calcStateDim*(*rotSize*, *C*, *A*)

Dimension of group.

static constexpr int **M** = *calcStateMtxSize*(*rotSize*, *C*)

State will have matrix of size M x M.

static constexpr int a = A == Eigen::Dynamic ? 0 : A

Handles defaults values of augmented sizes when A is Eigen::Dyanmic.

static constexpr int c = N == Eigen::Dynamic ? 3 : N

Handles defaults values of tangent vector sizes when A is Eigen::Dyanmic.

static constexpr int m = M == Eigen::Dynamic ? 3 : M

Handles defaults values of matrix sizes.

1.6.3 SO(3)

template<int **A** = 0>

class InEKF::**SO3** : public InEKF::*LieGroup*<*SO3*<*A*>, *calcStateDim*(3, 0, *A*), 3, *A*>

3D rotational states, also known as the 3x3 special orthogonal group, SO(3).

Template Parameters **A** – Number of augmented Euclidean states. Can be Eigen::Dynamic if desired. Defaults to 0.

Public Functions

inline **SO3**(const MatrixState &State = MatrixState::Identity(), const MatrixCov &Cov = MatrixCov::Zero(c, c), const VectorAug &Aug = VectorAug::Zero(a))

Construct *SO3* object with all available options.

Parameters

- **State** – A 2x2 Eigen matrix. Defaults to the identity.
- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.
- **Aug** – Additional euclidean states if A != 0. Defaults to 0s.

inline **SO3**(const *SO3* &State)

Copy constructor. Initialize with another *SO3* object.

Parameters **State** – *SO3* object. The matrix, covariance and augmented state will all be copied from it.

SO3(double w1, double w2, double w3, const MatrixCov &Cov = MatrixCov::Zero(c, c), const VectorAug &Aug = VectorAug::Zero(a))

Construct a new *SO3* object using angles and the matrix exponential.

Parameters

- **w1** – Angle 1.
- **w2** – Angle 2.
- **w3** – Angle 3.
- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.
- **Aug** – Additional euclidean states if A != 0. Defaults to 0s.

inline **SO3**(const TangentVector &xi, const MatrixCov &Cov = MatrixCov::Zero(c, c))

Construct a new *SO3* object from a tangent vector using the exponential operator.

Parameters

- **xi** – Tangent vector of size (3 + Augmented state size).
- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.

inline **~SO3**()

Destroy the *SO3* object.

inline *SO3* **R**() const

Gets rotational component of the state. In the *SO3* case, this is everything except the augmented Euclidean states and covariance.

Returns SO3<> Rotational component of the state.

void **addAug**(double x, double sigma = 1)

Adds an element to the augmented Euclidean state. Only usable if A = Eigen::Dynamic.

Parameters

- **x** – Variable to add.
- **sigma** – Covariance of element. Only used if state is uncertain.

SO3<*A*> **inverse**() const

Invert state.

Returns Inverted matrix (transpose). Augmented portion and covariance is dropped.

SO3<*A*> **operator***(const *SO3*<*A*> &rhs) const

Combine rotations. Augmented states are summed.

Parameters **rhs** – Right hand element of multiplication.

Returns Combined elements with same augmented size.

Public Static Functions

static MatrixState **wedge**(const TangentVector &xi)

Move element in R^n to the Lie algebra.

Parameters **xi** – Tangent vector

Returns MatrixState Element of Lie algebra

static *SO3* **exp**(const TangentVector &xi)

Move an element from R^n -> algebra -> group.

Parameters **xi** – Tangent vector

Returns Element of *SO3*

static TangentVector **log**(const *SO3* &g)

Move an element from group -> algebra -> R^n.

Parameters **g** – Group element

Returns TangentVector

static MatrixCov **Ad**(const *SO3* &g)

Compute the linear map Adjoint.

Parameters **g** – Element of *SO3*

Returns Matrix of size state dimension x state dimension

Public Static Attributes

static constexpr int **rotSize** = 3

Size of rotational component of group.

static constexpr int **N** = *calcStateDim*(*rotSize*, 0, *A*)

Dimension of group.

static constexpr int **M** = *calcStateMtxSize*(*rotSize*, 0)

State will have matrix of size M x M.

static constexpr int a = A == Eigen::Dynamic ? 0 : A

Handles defaults values of augmented sizes when A is Eigen::Dyanmic.

static constexpr int c = A == Eigen::Dynamic ? 3 : N

Handles defaults values of tangent vector sizes when A is Eigen::Dyanmic.

static constexpr int **m** = *M*

Handles defaults values of matrix sizes.

1.6.4 SE(3)

template<int **C** = 1, int **A** = 0>

class InEKF::**SE3** : public InEKF::*LieGroup*<*SE3*<*C*, *A*>, *calcStateDim*(3, *C*, *A*), *calcStateMtxSize*(3, *C*), *A*>

3D rigid body transformation, also known as the 4x4 special Euclidean group, SE(3).

Template Parameters

- **C** – Number of Euclideans columns to include. Can be Eigen::Dynamic. Defaults to 1.
- **A** – Number of augmented Euclidean states. Can be Eigen::Dynamic if desired. Defaults to 0.

Public Functions

inline **SE3**(const MatrixState &State = MatrixState::Identity(m, m), const MatrixCov &Cov = MatrixCov::Zero(c, c), const VectorAug &Aug = VectorAug::Zero(a, 1))

Construct *SE2* object with all available options.

Parameters

- **State** – An MxM Eigen matrix. Defaults to the identity.
- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.
- **Aug** – Additional euclidean states if A != 0. Defaults to 0s.

inline **SE3**(const *SE3* &State)

Copy constructor. Initialize with another *SE2* object.

Parameters **State** – *SE2* object. The matrix, covariance and augmented state will all be copied from it.

SE3(const TangentVector &xi, const MatrixCov &Cov = MatrixCov::Zero(c, c))

Construct a new *SE2* object from a tangent vector using the exponential operator.

Parameters

- **xi** – Tangent vector of size (3 + 3*Columns + Augmented state size).
- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.

SE3(const *SO3*<> R, const Eigen::Matrix<double, small_xi, 1> &xi, const MatrixCov &Cov = MatrixCov::Zero(c, c))

Construct a new *SE3* object.

Parameters

- **R** – Rotational portion of the *SE3* object.
- **xi** – Translational columns to input.

- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.

inline **SE3**(double w1, double w2, double w3, double x, double y, double z, const MatrixCov &Cov = MatrixCov::Zero(c, c))

Construct a new *SE3* object using exponential ooperator on angles and putting positions directly in.

Parameters

- **w1** – Rotaional component 1.
- **w2** – Rotaional component 2.
- **w3** – Rotaional component 3.
- **x** – X-position.
- **y** – Y-position.
- **z** – Z-position.
- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.

inline **~SE3**()

Destroy the *SE3* object.

inline *SO3* **R**() const

Gets rotational component of the state.

Returns SO2<> Rotational component of the state.

inline Eigen::Vector3d **operator[]**(int idx) const

Gets the ith positional column of the group.

Parameters **idx** – Index of column to get, from 0 to C-1.

Returns Eigen::Vector2d

void **addCol**(const Eigen::Vector3d &x, const Eigen::Matrix3d &sigma = Eigen::Matrix3d::Identity())

Adds a column to the matrix state. Only usable if C = Eigen::Dynamic.

Parameters

- **x** – Column to add in.
- **sigma** – Covariance of element. Only used if state is uncertain.

void **addAug**(double x, double sigma = 1)

Adds an element to the augmented Euclidean state. Only usable if A = Eigen::Dynamic.

Parameters

- **x** – Variable to add.
- **sigma** – Covariance of element. Only used if state is uncertain.

SE3 **inverse**() const

Invert state.

Returns Inverted matrix. Augmented portion and covariance is dropped.

SE3 **operator***(const *SE3* &rhs) const

Combine transformations. Augmented states are summed.

Parameters **rhs** – Right hand element of multiplication.

Returns Combined elements with same augmented size.

Public Static Functions

static MatrixState **wedge**(const TangentVector &xi)

Move element in R^n to the Lie algebra.

Parameters **xi** – Tangent vector

Returns MatrixState Element of Lie algebra

static *SE3* **exp**(const TangentVector &xi)

Move an element from R^n -> algebra -> group.

Parameters **xi** – Tangent vector

Returns Element of *SE2*

static TangentVector **log**(const *SE3* &g)

Move an element from group -> algebra -> R^n.

Parameters **g** – Group element

Returns TangentVector

static MatrixCov **Ad**(const *SE3* &g)

Compute the linear map Adjoint.

Parameters **g** – Element of *SE2*

Returns Matrix of size state dimension x state dimension

Public Static Attributes

static constexpr int **rotSize** = 3

Size of rotational component of group.

static constexpr int **N** = *calcStateDim*(*rotSize*, *C*, *A*)

Dimension of group.

static constexpr int **M** = *calcStateMtxSize*(*rotSize*, *C*)

State will have matrix of size M x M.

`static constexpr int a = A == Eigen::Dynamic ? 0 : A`

Handles defaults values of augmented sizes when A is Eigen::Dyanmic.

`static constexpr int c = N == Eigen::Dynamic ? 6 : N`

Handles defaults values of tangent vector sizes when A is Eigen::Dyanmic.

`static constexpr int m = M == Eigen::Dynamic ? 4 : M`

Handles defaults values of matrix sizes.

1.6.5 Lie Group Base

template<class **Class**, int **N**, int **M**, int **A**>

class InEKF::**LieGroup**

Base Lie Group Class.

Template Parameters

- **Class** – Class that is inheriting from it. Allows for better polymorphism
- **N** – Group dimension
- **M** – Lie Group matrix size
- **A** – Augmented Euclidean state size

Public Types

typedef Eigen::Matrix<double, *N*, 1> **TangentVector**

A tangent vector has size Nx1, where N is the state dimension.

typedef Eigen::Matrix<double, *N*, *N*> **MatrixCov**

The covariance matrix has size NxN, where N is the state dimension.

typedef Eigen::Matrix<double, *M*, *M*> **MatrixState**

A group element is a matrix of size MxM.

typedef Eigen::Matrix<double, *A*, 1> **VectorAug**

Vector of additional Euclidean states, of size Ax1.

Public Functions

inline **LieGroup**()

Construct a new Lie Group object. Default Contructor.

inline **LieGroup**(*MatrixState* State, *MatrixCov* Cov, *VectorAug* Aug)

Construct a new Lie Group object.

Parameters

- **State** – Group element
- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.
- **Aug** – Additional euclidean states if A != 0. Defaults to 0s.

inline **LieGroup**(*MatrixCov* Cov, *VectorAug* Aug)

Construct a new Lie Group object.

Parameters

- **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.
- **Aug** – Additional euclidean states if A != 0. Defaults to 0s.

inline **LieGroup**(*MatrixCov* Cov)

Construct a new Lie Group object.

Parameters **Cov** – Covariance of state. If not input, state is set as "certain" and covariance is not tracked.

inline virtual **~LieGroup**()

Destroy the Lie Group object.

inline bool **uncertain**() const

Returns whether object is uncertain, ie if it has a covariance.

Returns true

Returns false

inline const *MatrixCov* &**cov**() const

Get covariance of group element.

Returns const MatrixCov&

inline const *VectorAug* &**aug**() const

Get additional Euclidean state of object.

Returns const VectorAug&

inline const *MatrixState* &**mat**() const

Get actual group element.

Returns const MatrixState&

inline const *MatrixState* &**operator()**() const

Get actual group element.

Returns const MatrixState&

inline void **setCov**(const *MatrixCov* &Cov)

Set the state covariance.

Parameters **Cov** – Covariance matrix.

inline void **setAug**(const *VectorAug* &Aug)

Set the additional augmented state.

Parameters **Aug** – Augmented state vector.

inline void **setMat**(const *MatrixState* &State)

Set the group element.

Parameters **State** – matrix Lie group element.

inline const *Class* &**derived**() const

Cast *LieGroup* object to object that is inheriting from it.

Returns const Class&

inline *Class* **inverse**() const

Invert group element.

Returns Inverted group element. Augmented portion and covariance is dropped.

inline *TangentVector* **log**() const

Move this element from group -> algebra -> R^n.

Returns TangentVector

inline *MatrixCov* **Ad**() const

Get adjoint of group element.

Returns MatrixCov

inline *Class* **compose**(const *Class* &g) const

Multiply group elements.

Parameters **g** – Group element.

Returns Class

inline std::string **toString**() const

Convert group element to string. If uncertain print covariance as well. If has augmented state, print that as well.

Returns std::string

Public Static Functions

static inline *MatrixState* **wedge**(const *TangentVector* &xi)

Move element in R^n to the Lie algebra.

Parameters **xi** – Tangent vector

Returns MatrixState Element of Lie algebra

static inline *Class* **exp**(const *TangentVector* &xi)

Move an element from R^n -> algebra -> group.

Parameters **xi** – Tangent vector

Returns Element of *SO3*

static inline *TangentVector* **log**(const *Class* &g)

Move an element from group -> algebra -> R^n.

Parameters **g** – Group element

Returns TangentVector

static inline *MatrixCov* **Ad**(const *Class* &g)

Compute the linear map Adjoint.

Parameters **g** – Element of *SO3*

Returns Matrix of size state dimension x state dimension

1.6.6 Helpers

constexpr int `InEKF::`**`calcStateDim`**`(`int rotMtxSize, int C, int A`)`

Computes total dimension of the state.

Parameters

- **`rotMtxSize`** – Matrix size of the rotational component of the group.
- **`C`** – Number of columns to be included.
- **`A`** – Number of Euclidean states included.

Returns Total dimension of state.

constexpr int `InEKF::`**`calcStateMtxSize`**`(`int rotMtxSize, int C`)`

Compute group matrix size.

Parameters

- **`rotMtxSize`** – Matrix size of the rotational component of the group.
- **`C`** – Number of columns to be included.

Returns Total dimension of state.

1.7 Inertial Models

See the Underwater Inertial example to see these classes in usage.

1.7.1 Inertial Process Model

class `InEKF::`**`InertialProcess`** : public InEKF::*ProcessModel*<*SE3*<2, 6>, Eigen::Matrix<double, 6, 1>>

Inertial process model. Integrates IMU measurements and tracks biases. Requires "Imperfect InEKF" since biases don't fit into Lie group structure.

Public Functions

`InertialProcess``()`

Construct a new Inertial Process object.

inline **`~InertialProcess`**`()`

Destroy the Inertial Process object.

SE3<2, 6> **`f`**`(`Eigen::Vector6d u, double dt, *SE3*<2, 6> state`)` override

Overriden from base class. Integrates IMU measurements.

Parameters

- **`u`** – Control. First 3 are angular velocity, last 3 are linear acceleration.
- **`dt`** – Delta time
- **`state`** – Current state

Returns Integrated state

MatrixCov **makePhi**(const Eigen::Vector6d &u, double dt, const *SE3*<2, 6> &state, *ERROR* error) override

Overriden from base class. Since this is used in an "Imperfect InEKF", both left and right versions are slightly state dependent.

Parameters

- **u** – Control
- **dt** – Delta time
- **state** – Current state estimate (shouldn't be needed unless doing an "Imperfect InEKF")
- **error** – Right or left error. Function should be implemented to handle both.

Returns Phi

void **setGyroNoise**(double std)

Set the gyro noise. Defaults to 0 if not set.

Parameters **std** – Gyroscope standard deviation

void **setAccelNoise**(double std)

Set the accelerometer noise. Defaults to 0 if not set.

Parameters **std** – Accelerometer standard deviation

void **setGyroBiasNoise**(double std)

Set the gryo bias noise. Defaults to 0 if not set.

Parameters **std** – Gyroscope bias standard deviation

void **setAccelBiasNoise**(double std)

Set the accelerometer bias noise. Defaults to 0 if not set.

Parameters **std** – Accelerometer bias standard deviation

1.7.2 Depth Sensor

class InEKF::**DepthSensor** : public InEKF::*MeasureModel*<*SE3*<2, 6>>

Pressure/Depth sensor measurement model for use with inertial process model. Uses pseudo-measurements to fit into a left invariant measurement model.

Public Functions

DepthSensor(double std = 1)

Construct a new Depth Sensor object.

Parameters **std** – The standard deviation of the measurement.

inline **~DepthSensor**()

Destroy the Depth Sensor object.

virtual VectorB **processZ**(const Eigen::VectorXd &z, const *SE3*<2, 6> &state) override

Overriden from the base class. Inserts psuedo measurements for the x and y value to fit the invariant measurement.

Parameters

- **z** – Measurement

- **state** – Current state estimate.

Returns Processed measurement.

virtual MatrixS **calcSInverse**(const *SE3*<2, 6> &state) override

Overriden from base class. Calculate inverse of measurement noise S, using the Woodbury Matrix Identity.

Parameters **state** – Current state estimate.

Returns Inverse of measurement noise.

void **setNoise**(double std)

Set the measurement noise.

Parameters **std** – The standard deviation of the measurement.

1.7.3 Doppler Velocity Log

class InEKF::**DVLSensor** : public InEKF::*MeasureModel*<*SE3*<2, 6>>

DVL sensor measurement model for use with inertial process model.

Public Functions

DVLSensor()

Construct a new *DVLSensor* object. Assumes no rotation or translation between this and IMU frame.

DVLSensor(Eigen::Matrix3d dvlR, Eigen::Vector3d dvlT)

Construct a new *DVLSensor* object with offset from IMU frame.

Parameters

- **dvlR** – 3x3 Rotation matrix encoding rotationf rom DVL to IMU frame.
- **dvlT** – 3x1 Vector of translation from IMU to DVL in IMU frame.

DVLSensor(*SO3*<> dvlR, Eigen::Vector3d dvlT)

Construct a new *DVLSensor* object with offset from IMU frame.

Parameters

- **dvlR** – *SO3* object encoding rotationf rom DVL to IMU frame.
- **dvlT** – 3x1 Vector of translation from IMU to DVL in IMU frame.

DVLSensor(*SE3*<> dvlH)

Construct a new *DVLSensor* object with offset from IMU frame.

Parameters **dvlH** – *SE3* object encoding transformation from DVL to IMU frame.

inline **~DVLSensor**()

Destroy the *DVLSensor* object.

void **setNoise**(double std_dvl, double std_imu)

Set the noise covariances.

Parameters

- **std_dvl** – Standard deviation of DVL measurement.

- **std_imu** – Standard deviation of gyropscope measurement (needed b/c we transform frames).

virtual VectorB **processZ**(const Eigen::VectorXd &z, const *SE3*<2, 6> &state) override

Overriden from base class. Takes in a 6 vector with DVL measurement as first 3 elements and IMU as last three and converts DVL to IMU, then makes it the right size and passes it on.

Parameters

- **z** – DVL/IMU measurement.
- **state** – Current state estimate.

Returns Processed measurement.

1.8 SE2 Models

See the Victoria Park example to see these classes in usage.

1.8.1 Odometry Process Model

class InEKF::**OdometryProcess** : public InEKF::*ProcessModel*<*SE2*<>, *SE2*<>>

Odometry process model with single column.

Public Functions

inline **OdometryProcess**()

Construct a new Odometry Process object.

inline **OdometryProcess**(float theta_cov, float x_cov, float y_cov)

Construct a new Odometry Process object and set corresponding covariances.

Parameters

- **theta_cov** – Standard deviation of rotation between timesteps.
- **x_cov** – Standard deviation of x between timesteps.
- **y_cov** – Standard deviation of y between timesteps.

inline **OdometryProcess**(Eigen::Vector3d q)

Construct a new Odometry Process object. Set Q from vector.

Parameters **q** – Vector that becomes diagonal of Q.

inline **OdometryProcess**(Eigen::Matrix3d q)

Construct a new Odometry Process object. Set Q from matrix.

Parameters **q** – Matrix that is set as Q.

inline **~OdometryProcess**()

Destroy the Odometry Process object.

virtual *SE2* **f**(*SE2*<> u, double dt, *SE2*<> state) override

Overriden from base class. Propagates the model $X_{t+1} = XU$.

Parameters

- **u** – Control
- **dt** – Delta time
- **state** – Current state

Returns Updated state estimate

virtual MatrixCov **makePhi**(const *SE2*<> &u, double dt, const *SE2*<> &state, *ERROR* error) override

Overriden from base class. If right, this is the identity. If left, it's the adjoint of U.

Parameters

- **u** – Control
- **dt** – Delta time
- **state** – Current state estimate (shouldn't be needed unless doing an "Imperfect InEKF")
- **error** – Right or left error. Function should be implemented to handle both.

Returns Phi

inline void **setQ**(Eigen::Vector3d q)

Set Q from vector.

Parameters **q** – Vector that becomes diagonal of Q.

inline void **setQ**(Eigen::Matrix3d q)

Set Q from matrix.

Parameters **q** – Matrix that is set as Q.

inline void **setQ**(double q)

Set Q from scalar.

Parameters **q** – Scalar that becomes diagonal of Q

1.8.2 Dynamic Odometry Process Model

class InEKF::**OdometryProcessDynamic** : public InEKF::*ProcessModel*<*SE2*<Eigen::Dynamic>, *SE2*<>>

Odometry process model with variable number of columns, for use in SLAM on *SE2*.

Public Functions

inline **OdometryProcessDynamic**()

Construct a new Odometry Process Dynamic object.

inline **OdometryProcessDynamic**(float theta_cov, float x_cov, float y_cov)

Construct a new Odometry Process Dynamic object and set corresponding covariances.

Parameters

- **theta_cov** – Standard deviation of rotation between timesteps.
- **x_cov** – Standard deviation of x between timesteps.

- **y_cov** – Standard deviation of y between timesteps.

inline **OdometryProcessDynamic**(Eigen::Vector3d q)

Construct a new Odometry Process Dynamic object. Set Q from vector.

Parameters **q** – Vector that becomes diagonal of Q.

inline **OdometryProcessDynamic**(Eigen::Matrix3d q)

Construct a new Odometry Process Dynamic object. Set Q from matrix.

Parameters **q** – Matrix that is set as Q.

inline **~OdometryProcessDynamic**()

Destroy the Odometry Process Dynamic object.

virtual *SE2*<Eigen::Dynamic> **f**(*SE2*<> u, double dt, *SE2*<Eigen::Dynamic> state) override

Overriden from base class. Propagates the model $X_{t+1} = XU$. Landmarks are left as is.

Parameters

- **u** – Control
- **dt** – Delta time
- **state** – Current state

Returns Updated state estimate

virtual MatrixCov **makePhi**(const *SE2*<> &u, double dt, const *SE2*<Eigen::Dynamic> &state, *ERROR* error) override

Overriden from base class. If right, this is the identity. If left, it's the adjoint of U. Landmark elements are the identity in both versions of Phi.

Parameters

- **u** – Control
- **dt** – Delta time
- **state** – Current state estimate (shouldn't be needed unless doing an "Imperfect InEKF")
- **error** – Right or left error. Function should be implemented to handle both.

Returns Phi

inline void **setQ**(Eigen::Vector3d q)

Set Q from vector.

Parameters **q** – Vector that becomes diagonal of Q.

inline void **setQ**(Eigen::Matrix3d q)

Set Q from matrix.

Parameters **q** – Matrix that is set as Q.

inline void **setQ**(double q)

Set Q from scalar.

Parameters **q** – Scalar that becomes diagonal of Q

1.8.3 GPS

class `InEKF::GPSSensor` : public InEKF::*MeasureModel*<*SE2*<Eigen::Dynamic>>

GPS Sensor for use in *SE2* SLAM model.

Public Functions

`GPSSensor`(double std = 1)

Construct a new *GPSSensor* object.

Parameters `std` – The standard deviation of the measurement.

inline `~GPSSensor`()

Destroy the *GPSSensor* object.

virtual VectorB `processZ`(const Eigen::VectorXd &z, const *SE2*<Eigen::Dynamic> &state) override

Overriden from the base class. Needed to fill out H/z with correct number of columns based on number of landmarks in state.

Parameters

- `z` – Measurement
- `state` – Current state estimate.

Returns Processed measurement.

1.8.4 Landmark Sensor

class `InEKF::LandmarkSensor` : public InEKF::*MeasureModel*<*SE2*<Eigen::Dynamic>>

Landmark sensor used in SLAM on *SE2*.

Public Functions

`LandmarkSensor`(double std_r, double std_b)

Construct a new Landmark Sensor object.

Parameters

- `std_r` – Range measurement standard deviation
- `std_b` – Bearing measurement standard deviation

inline `~LandmarkSensor`()

Destroy the Landmark Sensor object.

void `sawLandmark`(int idx, const *SE2*<Eigen::Dynamic> &state)

Sets H based on what landmark was recently seen.

Parameters

- `idx` – Index of landmark recently seen.
- `state` – Current state estimate. Used for # of landmarks.

double **calcMahDist**(const Eigen::VectorXd &z, const *SE2*<Eigen::Dynamic> &state)

Calculates Mahalanobis distance of having seen a certain landmark. Used for data association.

Parameters

- **z** – Range and bearing measurement
- **state** – Current state estimate

Returns Mahalanobis distance

virtual VectorB **processZ**(const Eigen::VectorXd &z, const *SE2*<Eigen::Dynamic> &state) override

Overriden from base class. Converts r,b -> x,y coordinates and shifts measurement covariance. Then fills out z accordingly.

Parameters

- **z** – Measurement
- **state** – Current state estimate.

Returns Processed measurement.

virtual MatrixS **calcSInverse**(const *SE2*<Eigen::Dynamic> &state) override

Overriden from base class. If using RInEKF, takes advantage of sparsity of H to shrink matrix multiplication. Otherwise, operates identically to base class.

Parameters **state** – Current state estimate.

Returns Inverse of measurement noise.

inline virtual MatrixH **makeHError**(const *SE2*<Eigen::Dynamic> &state, *ERROR* iekfERROR) override

Overriden from base class. Saves filter error for later use, then calls base class.

Parameters

- **state** – Current state estimate.
- **iekfERROR** – Type of filter error.

Returns H_error_

1.9 Core Classes

1.9.1 Error

class inekf.**ERROR**(*value*)

Type of invariant error. Has options for left or right.

Attributes:

LEFT	Left error
RIGHT	Right error

LEFT = 0

Left error

RIGHT = 1

Right error

1.9.2 Invariant Extended Kalman Filter

class inekf.**InEKF**(*pModel*, *x0*, *error*)

The Invariant Extended Kalman Filter

Parameters

- **pModel** (*ProcessModel*) – Process model
- **state** (*inekf.LieGroup*) – Initial state, must be of same group that process model uses and must be uncertain
- **error** (*ERROR*) – Right or left invariant error

Methods:

addMeasureModel(name, m)	Add measurement model to the filter.
predict(u[, dt])	Prediction Step.
update(name, m)	Update Step.

Attributes:

state	Current state estimate.

addMeasureModel(*name*, *m*)

Add measurement model to the filter.

Parameters

- **name** (str) – Name of measurement model.
- **m** (*MeasureModel*) – A measure model pointer, templated by the used group.

predict(*u*, *dt=1*)

Prediction Step.

Parameters

- **u** (*control*) – Must be same as what process model uses.
- **dt** (float) – Delta t. Used sometimes depending on process model. Defaults to 1.

Returns State estimate

Return type *inekf.LieGroup*

property state

Current state estimate. May be read or wrriten, but can't be edited in place.

Returns state

Return type *inekf.LieGroup*

update(*name*, *m*)

Update Step.

Parameters

- **name** (str) – Name of measurement model.
- **z** (np.ndarray) – Measurement. May vary in size depending on how measurement model processes it.

Returns State estimate.

Return type *inekf.LieGroup*

1.9.3 Measure Model

class inekf.**MeasureModel**(*b*, *M*, *error*)

Base class measure model. Written to be inherited from, but in most cases this class will be sufficient. More information on inheriting can be seen in *Custom Models*.

We have overloaded the [] operator to function as a python template. Example of this can be seen in *Getting Started*.

Templates:

- Group State's group that is being tracked, of type *inekf.LieGroup*.

Attributes:

H	Linearized matrix H.
H_error	This is the converted H used in InEKF if it's a right filter with left measurement or vice versa.
M	Measurement covariance.
b	b vector used in measure model.
error	Type of error of the filter (right/left)

Methods:

calcSInverse(state)	Calculate inverse of measurement noise S, using H_error.
calcV(z, state)	Computes innovation based on measurement model.
makeHError(state, iekfERROR)	Sets and returns H_error for settings where filter error type != measurement error type.
processZ(z, state)	Process measurement before putting into InEKF.
setHandb(b)	Sets measurement vector b and recreates H accordingly.

property H

Linearized matrix H. Will be automatically created from b in constructor unless overriden. May be read or written, but not modified in place.

Returns np.ndarray

property H_error

This is the converted H used in InEKF if it's a right filter with left measurement or vice versa. Used in calcSInverse if overriden. Probably won't need to be overwritten. May be read or written, but not modified in place.

Returns np.ndarray

property M

Measurement covariance.

Returns np.ndarray

property b

b vector used in measure model.

Returns `np.ndarray`

calcSInverse(*state*)

Calculate inverse of measurement noise S, using H_error. Called fourth in the update step.

Parameters **state** (*inekf.LieGroup*) – Current state estimate.

Returns Inverse of measurement noise.

Return type `np.ndarray`

calcV(*z*, *state*)

Computes innovation based on measurement model. Called third in the update step.

Parameters

- **z** (`np.ndarray`) – Measurement.
- **state** (*inekf.LieGroup*) – Current state estimate.

Returns Truncated innovation.

Return type `np.ndarray`

property error

Type of error of the filter (right/left)

Returns *ERROR*

makeHError(*state*, *iekfERROR*)

Sets and returns H_error for settings where filter error type != measurement error type. Done by multiplying H by adjoint of current state estimate. Called second in update step.

Parameters

- **state** (*inekf.LieGroup*) – Current state estimate.
- **iekfERROR** (*ERROR*) – Type of filter error.

Returns H_error

Return type `np.ndarray`

processZ(*z*, *state*)

Process measurement before putting into InEKF. Can be used to change frames, convert r/b->x/y, or append 0s. By default is used to append zeros/ones onto it according to b vector set. Called first in update step.

Parameters

- **z** (`np.ndarray`) – Measurement
- **state** (*inekf.LieGroup*) – Current state estimate.

Returns Processed measurement.

Return type `np.ndarray`

setHandb(*b*)

Sets measurement vector b and recreates H accordingly. Useful if vector b isn't constant.

Parameters **b** (`np.ndarray`) – Measurement model b

1.9.4 Process Model

class inekf.**ProcessModel**

Base class process model. Must be inheriting from, base class isn't implemented. More information on inheriting can be seen in *Custom Models*.

We have overloaded the [] operator to function as a python template. Example of this can be seen in *Getting Started*.

Templates:

- Group State's group that is being tracked, of type *inekf.LieGroup*.
- U Form of control. Can be either a group of *inekf.LieGroup*, or a vector. Vectors can be used for example by "Vec3" or 3 for a vector of size 3. -1, "D", or "VecD" for dynamic control size.

Attributes:

Q	Process model covariance.

Methods:

f(u, dt, state)	Propagates state forward one timestep.
makePhi(u, dt, state)	Make a discrete time linearized process model matrix, with $Phi = exp(ADelta t)$.

property Q

Process model covariance. May be read or written, but not modified in place.

Returns np.ndarray

f(*u*, *dt*, *state*)

Propagates state forward one timestep. Must be overriden, has no default implementation.

Parameters

- **u** (*control*) – Control
- **dt** (float) – Delta time
- **state** (*inekf.LieGroup*) – Current state

Returns Updated state estimate

Return type *inekf.LieGroup*

makePhi(*u*, *dt*, *state*)

Make a discrete time linearized process model matrix, with $Phi = exp(ADelta t)$. Must be overriden, has no default implementation.

Parameters

- **u** (*control*) – Control
- **dt** (float) – Delta time
- **state** (*inekf.LieGroup*) – Current state estimate (shouldn't be needed unless doing an "Imperfect InEKF")
- **error** (*ERROR*) – Right or left error. Function should be implemented to handle both.

Returns Phi

Return type np.ndarray

1.10 Lie Groups

1.10.1 SO(2)

class inekf.**SO2**(**args*, ***kwargs*)

Bases: *inekf.lie_groups.LieGroup*

2D rotational states, also known as the 2x2 special orthogonal group, SO(2).

See the C++ counterpart (*InEKF::SO2*) for documentation on constructing an object. Further, we have overloaded the [] operator to function as a python template. Example of this can be seen in *Getting Started*. Templates include:

Templates:

- A Number of augmented Euclidean states. Can be -1 or "D" for dynamic. Defaults to 0.

1.10.2 SE(2)

class inekf.**SE2**(**args*, ***kwargs*)

Bases: *inekf.lie_groups.LieGroup*

2D rigid body transformation, also known as the 3x3 special Euclidean group, SE(2).

See the C++ counterpart (*InEKF::SE2*) for documentation on constructing an object. Further, we have overloaded the [] operator to function as a python template. Example of this can be seen in *Getting Started*. Templates include:

Templates:

- C Number of Euclideans columns to include. Can be -1 or "D" for dynamic. Defaults to 1.
- A Number of augmented Euclidean states. Can be -1 or "D" for dynamic. Defaults to 0.

Methods:

__getitem__(idx)	Gets the ith positional column of the group.
addCol(x, sigma)	Adds a column to the matrix state.

__getitem__(*idx*)

Gets the ith positional column of the group.

Parameters **idx** (float) – Index of column to get, from 0 to C-1.

Returns np.ndarray

addCol(*x*, *sigma*)

Adds a column to the matrix state. Only usable if C = Eigen::Dynamic.

Parameters

- **x** (np.ndarray) – Column to add in.
- **sigma** (np.ndarray) – Covariance of element. Only used if state is uncertain.

1.10.3 SO(3)

class `inekf.SO3`(**args*, ***kwargs*)

Bases: *inekf.lie_groups.LieGroup*

3D rotational states, also known as the 3x3 special orthogonal group, SO(3).

See the C++ counterpart (*InEKF::SO3*) for documentation on constructing an object. Further, we have overloaded the `[]` operator to function as a python template. Example of this can be seen in *Getting Started*. Templates include:

Templates:

- `A` Number of augmented Euclidean states. Can be -1 or "D" for dynamic. Defaults to 0.

1.10.4 SE(3)

class `inekf.SE3`(**args*, ***kwargs*)

Bases: *inekf.lie_groups.LieGroup*

3D rigid body transformation, also known as the 4x4 special Euclidean group, SE(3).

See the C++ counterpart (*InEKF::SE3*) for documentation on constructing an object. Further, we have overloaded the `[]` operator to function as a python template. Example of this can be seen in *Getting Started*. Templates include:

Templates:

- `C` Number of Euclideans columns to include. Can be -1 or "D" for dynamic. Defaults to 1.
- `A` Number of augmented Euclidean states. Can be -1 or "D" for dynamic. Defaults to 0.

Methods:

__getitem__(idx)	Gets the ith positional column of the group.
addCol(x, sigma)	Adds a column to the matrix state.

`__getitem__`(*idx*)

Gets the ith positional column of the group.

Parameters **idx** (`float`) – Index of column to get, from 0 to C-1.

Returns `np.ndarray`

`addCol`(*x*, *sigma*)

Adds a column to the matrix state. Only usable if C = Eigen::Dynamic.

Parameters

- **x** (`np.ndarray`) – Column to add in.
- **sigma** (`np.ndarray`) – Covariance of element. Only used if state is uncertain.

1.10.5 Lie Group Base

class inekf.**LieGroup**

Attributes:

Ad	Get adjoint of group element.
R	Gets rotational component of the state.
aug	Get additional Euclidean state of object.
cov	Get covariance of group element.
inverse	Invert group element.
log	Move this element from group -> algebra -> R^n
mat	Get actual group element.
uncertain	Returns whether object is uncertain, ie if it has a covariance.

Methods:

*Ad_*(g)	Compute the linear map Adjoint
__invert__()	Invert group element.
__matmul__(rhs)	Combine transformations.
addAug(a, sigma)	Adds an element to the augmented Euclidean state.
exp(xi)	Move an element from R^n -> algebra -> group
*log_*(g)	Move an element from group -> algebra -> R^n
wedge(xi)	Move element in R^n to the Lie algebra.

property **Ad**

Get adjoint of group element.

Returns np.ndarray

static **Ad_**(*g*)

Compute the linear map Adjoint

Parameters **g** (*inekf.LieGroup*) – Element of group

Returns np.ndarray

property **R**

Gets rotational component of the state.

Returns *inekf.SO2*

__invert__()

Invert group element. Drops augmented state and covariance.

Returns *inekf.LieGroup*

__matmul__(*rhs*)

Combine transformations. Augmented states are summed.

Parameters **rhs** (*inekf.LieGroup*) – Right hand element of multiplication.

Returns *inekf.LieGroup*

addAug(*a*, *sigma*)

Adds an element to the augmented Euclidean state. Only usable if A = Eigen::Dynamic.

Parameters

- **x** (`float`) – Variable to add.
- **sigma** (`float`) – Covariance of element. Only used if state is uncertain.

`property aug`

Get additional Euclidean state of object.

Returns `np.ndarray`

`property cov`

Get covariance of group element.

Returns `np.ndarray`

`static exp(xi)`

Move an element from R^n -> algebra -> group

Parameters **xi** (`np.ndarray`) – Tangent vector

Returns *inekf.LieGroup*

`property inverse`

Invert group element. Augmented portion and covariance is dropped.

Returns *inekf.LieGroup*

`property log`

Move this element from group -> algebra -> R^n

Returns `np.ndarray`

`static log_(g)`

Move an element from group -> algebra -> R^n

Parameters **g** (*inekf.LieGroup*) – Group element

Returns `np.ndarray`

`property mat`

Get actual group element.

Returns `np.ndarray`

`property uncertain`

Returns whether object is uncertain, ie if it has a covariance.

Returns `bool`

`static wedge(xi)`

Move element in R^n to the Lie algebra.

Parameters **xi** (`np.ndarray`) – Tangent vector

Returns `np.ndarray`

1.11 Inertial Models

See the Underwater Inertial example to see these classes in usage.

1.11.1 Inertial Process Model

class inekf.**InertialProcess**(**args: Any*, ***kwargs: Any*)

Bases: `inekf._inekf.ProcessModel_SE3_2_6_Vec6`

Inertial process model. Integrates IMU measurements and tracks biases. Requires "Imperfect InEKF" since biases don't fit into Lie group structure.

Methods:

f(u, dt, state)	Overriden from base class.
makePhi(u, dt, state)	Overriden from base class.
setAccelBiasNoise(std)	Set the accelerometer bias noise.
setAccelNoise(std)	Set the accelerometer noise.
setGyroBiasNoise(std)	Set the gryo bias noise.
setGyroNoise(std)	Set the gyro noise.

f(*u*, *dt*, *state*)

Overriden from base class. Integrates IMU measurements.

Parameters

- **u** (`np.ndarray`) – 6-Vector. First 3 are angular velocity, last 3 are linear acceleration.
- **dt** (`float`) – Delta time
- **state** (`inekf.SE3[2,6]`) – Current state

Returns Updated state estimate

Return type `inekf.SE3[2,6]`

makePhi(*u*, *dt*, *state*)

Overriden from base class. Since this is used in an "Imperfect InEKF", both left and right versions are slightly state dependent.

Parameters

- **u** (`np.ndarray`) – 6-Vector. First 3 are angular velocity, last 3 are linear acceleration.
- **dt** (`float`) – Delta time
- **state** (`inekf.SE3[2,6]`) – Current state estimate (shouldn't be needed unless doing an "Imperfect InEKF")
- **error** (*ERROR*) – Right or left error. Function should be implemented to handle both.

Returns Phi

Return type `np.ndarray`

setAccelBiasNoise(*std*)

Set the accelerometer bias noise. Defaults to 0 if not set.

Parameters **std** (`float`) – Accelerometer bias standard deviation

setAccelNoise(*std*)

Set the accelerometer noise. Defaults to 0 if not set.

Parameters **std** (float) – Accelerometer standard deviation

setGyroBiasNoise(*std*)

Set the gryo bias noise. Defaults to 0 if not set.

Parameters **std** (float) – Gyroscope bias standard deviation

setGyroNoise(*std*)

Set the gyro noise. Defaults to 0 if not set.

Parameters **std** (float) – Gyroscope standard deviation

1.11.2 Depth Sensor

class inekf.**DepthSensor**(**args: Any*, ***kwargs: Any*)

Bases: `inekf._inekf.MeasureModel_SE3_2_6`

Pressure/Depth sensor measurement model for use with inertial process model. Uses pseudo-measurements to fit into a left invariant measurement model.

Parameters **std** (float) – The standard deviation of a measurement.

Methods:

calcSInverse(state)	Overriden from base class.
processZ(z, state)	Overriden from the base class.
setNoise(std)	Set the measurement noise

calcSInverse(*state*)

Overriden from base class. Calculate inverse of measurement noise S, using the Woodbury Matrix Identity

Parameters **state** (inekf.SE3[2,6]) – Current state estimate.

Returns Inverse of measurement noise.

Return type np.ndarray

processZ(*z*, *state*)

Overriden from the base class. Inserts psuedo measurements for the x and y value to fit the invariant measurement.

Parameters

- **z** (np.ndarray) – Measurement
- **state** (inekf.SE3[2,6]) – Current state estimate.

Returns Processed measurement.

Return type np.ndarray

setNoise(*std*)

Set the measurement noise

Parameters **std** (float) – The standard deviation of the measurement.

1.11.3 Doppler Velocity Log

class inekf.**DVLSensor**(**args: Any*, ***kwargs: Any*)

Bases: `inekf._inekf.MeasureModel_SE3_2_6`

DVL sensor measurement model for use with inertial process model.

There's a number of available constructors, see `InEKF::DVLSensor` for a list of all of them.

Methods:

`processZ`(z, state)	Overriden from base class.
`setNoise`(std_dvl, std_imu)	Set the noise covariances.

processZ(*z*, *state*)

Overriden from base class. Takes in a 6 vector with DVL measurement as first 3 elements and IMU as last three and converts DVL to IMU, then makes it the right size and passes it on.

Parameters

- **z** (`np.ndarray`) – Measurement
- **state** (`inekf.SE3[2,6]`) – Current state estimate.

Returns Processed measurement.

Return type `np.ndarray`

setNoise(*std_dvl*, *std_imu*)

Set the noise covariances.

Parameters

- **std_dvl** (`float`) – Standard deviation of DVL measurement.
- **std_imu** (`float`) – Standard deviation of gyropscope measurement (needed b/c we transform frames).

1.12 SE2 Models

See the Victoria Park example to see these classes in usage.

1.12.1 Odometry Process Model

class inekf.**OdometryProcess**(**args: Any*, ***kwargs: Any*)

Bases: `inekf._inekf.ProcessModel_SE2_1_0_SE2_1_0`

Odometry process model with single column.

There's a number of available constructors, see `InEKF::OdometryProcess` for a list of all of them.

Methods:

`f`(u, dt, state)	Overriden from base class.
`makePhi`(u, dt, state)	Overriden from base class.
`setQ`(q)	Set Q from a variety of sources

f(*u*, *dt*, *state*)

Overriden from base class. Propagates the model $X_{t+1} = XU$

Parameters

- **u** (*inekf.SE2*) – Rigid body transformation of vehicle since last timestep.
- **dt** (float) – Delta time
- **state** (*inekf.SE2*) – Current state

Returns Updated state estimate

Return type *inekf.SE2*

makePhi(*u*, *dt*, *state*)

Overriden from base class. If right, this is the identity. If left, it's the adjoint of U.

Parameters

- **u** (*inekf.SE2*) – Rigid body transformation of vehicle since last timestep.
- **dt** (float) – Delta time
- **state** (*inekf.SE2*) – Current state estimate (shouldn't be needed unless doing an "Imperfect InEKF")
- **error** (*ERROR*) – Right or left error. Function should be implemented to handle both.

Returns Phi

Return type np.ndarray

setQ(*q*)

Set Q from a variety of sources

Parameters **q** (np.ndarray or float) – Can be a float, 3-vector, or 3x3-matrix. Sets the covariance Q accordingly.

1.12.2 Dynamic Odometry Process Model

class inekf.**OdometryProcessDynamic**(**args: Any*, ***kwargs: Any*)

Bases: inekf._inekf.ProcessModel_SE2_D_0_SE2_1_0

Odometry process model with variable number of columns, for use in SLAM on SE2.

There's a number of available constructors, see *InEKF::OdometryProcessDynamic* for a list of all of them.

Methods:

f(u, dt, state)	Overriden from base class.
makePhi(u, dt, state)	Overriden from base class.
setQ(q)	Set Q from a variety of sources

f(*u*, *dt*, *state*)

Overriden from base class. Propagates the model $X_{t+1} = XU$. Landmarks are left as is.

Parameters

- **u** (*inekf.SE2*) – Rigid body transformation of vehicle since last timestep.
- **dt** (float) – Delta time

- **state** (inekf.SE2[-1,0]) – Current state

Returns Updated state estimate

Return type inekf.SE2[-1,0]

makePhi(*u*, *dt*, *state*)

Overriden from base class. If right, this is the identity. If left, it's the adjoint of U. Landmark elements are the identity in both versions of Phi.

Parameters

- **u** (*inekf.SE2*) – Rigid body transformation of vehicle since last timestep.
- **dt** (float) – Delta time
- **state** (inekf.SE2[-1,0]) – Current state estimate (shouldn't be needed unless doing an "Imperfect InEKF")
- **error** (*ERROR*) – Right or left error. Function should be implemented to handle both.

Returns Phi

Return type np.ndarray

setQ(*q*)

Set Q from a variety of sources

Parameters q (np.ndarray or float) – Can be a float, 3-vector, or 3x3-matrix. Sets the covariance Q accordingly.

1.12.3 GPS

class inekf.**GPSSensor**(**args: Any*, ***kwargs: Any*)

Bases: inekf._inekf.MeasureModel_SE2_D_0

GPS Sensor for use in SE2 SLAM model.

Parameters std (float) – The standard deviation of a measurement.

Methods:

processZ(z, state)	Overriden from the base class.

processZ(*z*, *state*)

Overriden from the base class. Needed to fill out H/z with correct number of columns based on number of landmarks in state.

Parameters

- **z** (np.ndarray) – Measurement
- **state** (inekf.SE2[-1,0]) – Current state estimate.

Returns Processed measurement.

Return type np.ndarray

1.12.4 Landmark Sensor

class `inekf.LandmarkSensor`(**args: Any*, ***kwargs: Any*)

Bases: `inekf._inekf.MeasureModel_SE2_D_0`

Landmark sensor used in SLAM on SE2

Parameters

- **std_r** (`float`) – Range measurement standard deviation
- **std_b** (`float`) – Bearing measurement standard deviation

Methods:

calcMahDist(state)	Calculates Mahalanobis distance of having seen a certain landmark.
calcSInverse(state)	Overriden from base class.
makeHError(state, iekfERROR)	Overriden from base class.
processZ(z, state)	Overriden from base class.
sawLandmark(state)	Sets H based on what landmark was recently seen.

calcMahDist(*state*)

Calculates Mahalanobis distance of having seen a certain landmark. Used for data association.

Parameters

- **z** (`np.ndarray`) – Range and bearing measurement
- **state** (`inekf.SE2[-1,0]`) – Current state estimate

Returns Mahalanobis distance

Return type `float`

calcSInverse(*state*)

Overriden from base class. If using RInEKF, takes advantage of sparsity of H to shrink matrix multiplication. Otherwise, operates identically to base class.

Parameters **state** (`inekf.SE2[-1,0]`) – Current state estimate.

Returns Inverse of measurement noise.

Return type `np.ndarray`

makeHError(*state*, *iekfERROR*)

Overriden from base class. Saves filter error for later use, then calls base class.

Parameters

- **state** (`inekf.SE2[-1,0]`) – Current state estimate.
- **iekfERROR** (*ERROR*) – Type of filter error.

Returns H_error

Return type `np.ndarray`

processZ(*z*, *state*)

Overriden from base class. Converts r,b -> x,y coordinates and shifts measurement covariance. Then fills out z accordingly.

Parameters

- **z** (np.ndarray) – Measurement
- **state** (inekf.SE2[-1,0]) – Current state estimate.

Returns Processed measurement.

Return type np.ndarray

sawLandmark(*state*)

Sets H based on what landmark was recently seen.

Parameters

- **seen.** (*idx Index of landmark recently*) –
- **landmarks.** (*state Current state estimate. Used for # of*) –

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

O

P

Q

R

S

U

W