Praktikumsbericht

[**IAV as an internship company**](#_j605te95792e) **0**

[**Topic of internship**](#_bhz0k6izde50) **1**

[**Organization**](#_iop0veuo6g5f) **3**

[**Software Architecture**](#_xyt717fdc6gg) **6**

[Filter Core](#_r8oiuar6xsos) 6

[Motion Model](#_mqbgwv3l9zb6) 7

[2D Models](#_7r2whbj3u6jc) 8

[CTRV (Constant turning rate and velocity)](#_qtenqaxu55fk) 8

[CTRA (Constant turning rate and acceleration)](#_3gpr90mb9hxb) 8

[Input Sensor Types](#_9z41o38buslq) 11

[IMU](#_94fkmgxgojq4) 12

[GPS](#_wgav6fgcaud5) 14

[Filter Algorithms](#_ehdor97dir1z) 16

[Fusion Strategies](#_lljf3v8e4ymt) 17

[Data triggered](#_5tstcgb8lz04) 17

[Time triggered](#_9frxqih0ivut) 18

# IAV as an internship company

IAV GmbH Ingenieurgesellschaft Auto und Verkehr is an engineering company in the automotive engineering that employs over 8000 members of staff. It was founded in 1983 as a spin-off at the TU with the goal of providing the infrastructure for development of automotive technologies. IAV's core competencies include solutions suitable for series production in all fields relating to electronics, powertrain and vehicle development. In addition to the development centers in Berlin, Gifhorn and Chemnitz IAV also owns a location in Dresden where my internship takes place. The range of services at the Dresden location includes cockpit electronics & telematics, HMI software, device & application software, hardware & software, security and ADAS validation. IAV Dresden also provides solutions for intelligent driving systems, where I also do my internship as part of the perception team. The perception team is familiar with local localization, mapping, SLAM, planning (i.e. the navigation stack) and deep learning methods such as object recognition etc.

# Topic of internship

The topic of the internship is Localization Fusion Library. The task consists of the design, planning, implementation and evaluation of the library. The library should be able to fuse measurements from wheel encoders, IMU sensors, GPS, camera odometry and lidars. For the input, the ROS data types are used as IDL. But the library itself should be ROS independent as IAV also supports other frameworks such as ADTF.

|  |  |
| --- | --- |
| **Sensors**   * Encoder * IMU * GPS * Camera * Lidar | **Odometries**   * Encoder Odometry * IMU Odometry * GPS Odometry * Visual Odometry * Lidar Odometry |

**Input / Output Data Types (as IDL)**

* [nav\_msgs/Odometry](http://docs.ros.org/api/nav_msgs/html/msg/Odometry.html)
* [sensor\_msgs/Imu](http://docs.ros.org/api/sensor_msgs/html/msg/Imu.html)
* [geometry\_msgs/PoseWithCovarianceStamped](http://docs.ros.org/api/geometry_msgs/html/msg/PoseWithCovarianceStamped.html)

The library should allow the usage of different Fusion Algorithms such as Extended Kalman Filter(EKF), Unscented Kalman Filter(UKF) or Partikel Filter.

The inspiration for the library comes mostly from [Robot\_Localization\_Package](https://github.com/cra-ros-pkg/robot_localization) which is a very famous package in the navigation stack of ROS. But many aspects have been changed, re-designed or developed from scratch along the way. They will be explained once again under the headline Software Architecture.

The main features of the library are modularity and genericity. It should be possible to use and extend different states, motion models, sensor inputs, fusion algorithms and fusing strategies. The library should also be cross-compileable and thus executable on Linux, Windows or other platforms.

# 

# 

# 

# 

# 

# HEAT Concludes First Test Phase | IAV

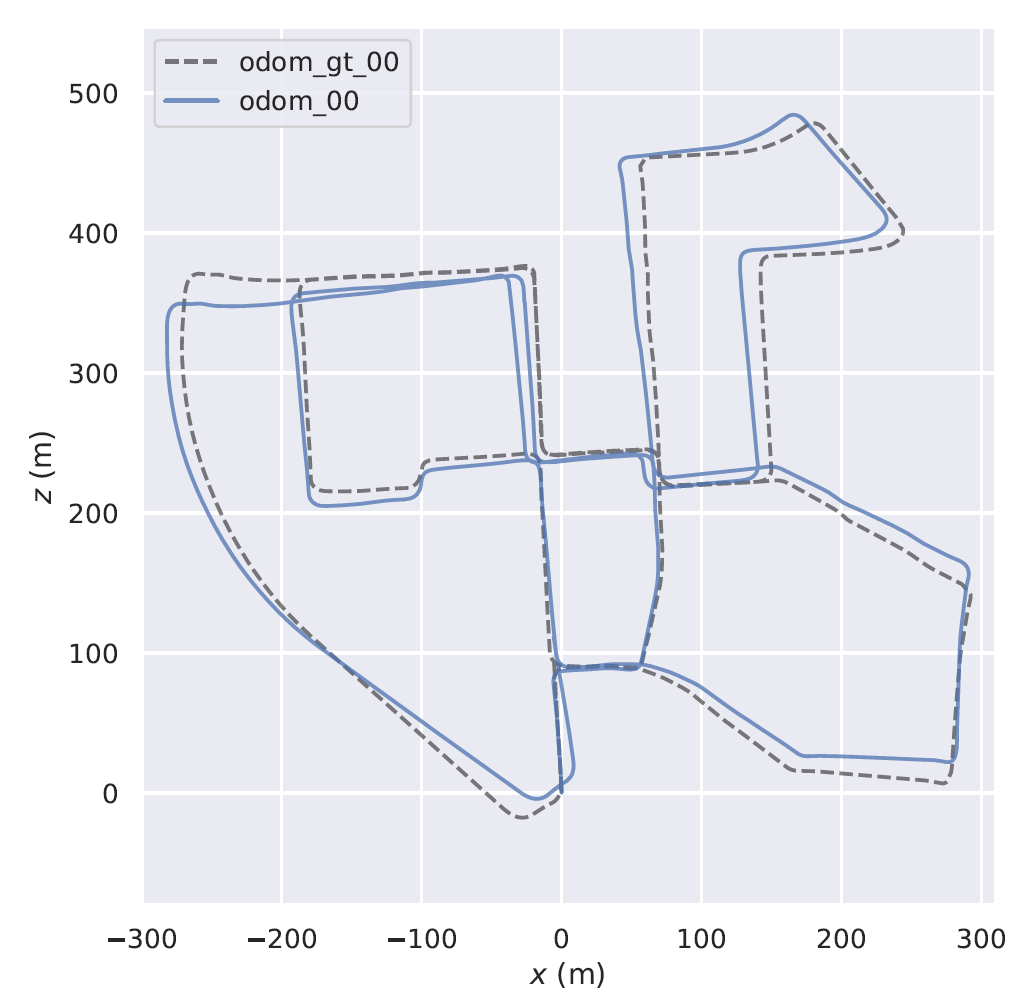
# 

# 

# 

# 

# [[1]](#footnote-0)



# 

# Organization

The following table gives an overview of the distribution of tasks during my internship.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Phases & Tasks** | | W1 | W2 | W3 | W4 | W5 | W6 | W7 | W8 | W9 | W  10 | W  11 | W  12 | W  13 | W  14 | W  15 | W  16 | W  17 | W  18 |
| 1. **Related Work Review** | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Checking available implementations |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Checking previous publications |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Report the outcomes with justifications |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1. **Initial Library Pipeline** | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Selecting one input and output datatypes |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Selecting one fusion algorithm |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Implementing initial library pipeline |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Testing the pipeline over a dataset |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Report the outcomes with discussions |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Generate a README / documentation |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1. **Update Library Pipeline** | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Allow various input / output datatypes |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Allow various fusion algorithms |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Implementing updated library pipeline |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Testing the pipeline over a dataset |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Report the outcomes with discussions |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Update a README / documentation |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

A detailed description of the tasks is again summarized below:

**1.a) Checking available implementations**

There are few packages available online, which are worth checking as first step before proceeding with implementation. As a sense of inspiration and possible citation they have to be reviewed.

**1.b) Checking previous publications**

Not all published work has open-source packages, accordingly, checking publication over the last 5-years in the topic would give a better understanding of the problem, the current advances and a guide on where to go.

**1.c) Report the outcomes with justifications**

Deliverable report on the outcomes of the previous steps, with justifications of the decisions to be made for the next steps.

**2.a) Selecting one input and output datatypes**

Recommendation to start with Odometry as the initial datatype for both inputs and output. Any selection should be decoded to an internal format, for the ease of modularity in the future.

**2.b) Selecting one fusion algorithm**

Recommendation to start with Extended Kalman Filter as the fusion algorithm, however, if there is a proper justification for another choice, it will be welcomed and appreciated. Any selection should be implemented using template classes, for the ease of modularity in the future.

**2.c) Implementing initial library pipeline**

Deliverable package with the implementation of the selected algorithm to fuse localizations with selected datatypes.

**2.d) Testing the pipeline over a dataset**

Testing must be done over a dataset with reference localization.

**2.e) Report the outcomes with discussions**

Based on selected evaluation metrics, the obtained results are compared to the reference one, in addition to discussion/comments on the behavior as validation to the initial hypothesis that fusion leads to better outcome, and in case of otherwise, justification is required.

**2.f) Generate a README/documentation**Writing a full README for the package with documentation on how to use

**3.a) Allow various input / output datatypes**

Through a configuration file, allow the possibility for more datatypes for the inputs and outputs, which should be decoded to the same internal format.

**3.b) Allow various fusion algorithms**

Through a configuration file, allow the possibility for more algorithms for the fusion, which should be using the same template classes.

**3.c) Implementing updated library pipeline**

Deliverable update of the package with the implementation of the extension of various algorithms to fuse localizations with various datatypes.

**3.d) Testing the pipeline over a dataset**

Testing must be done over the same dataset with reference localization.

**3.e) Report the outcomes with discussions**

Updated the report with the findings based on the same evaluation metrics

**3.f)Update a README / documentation**

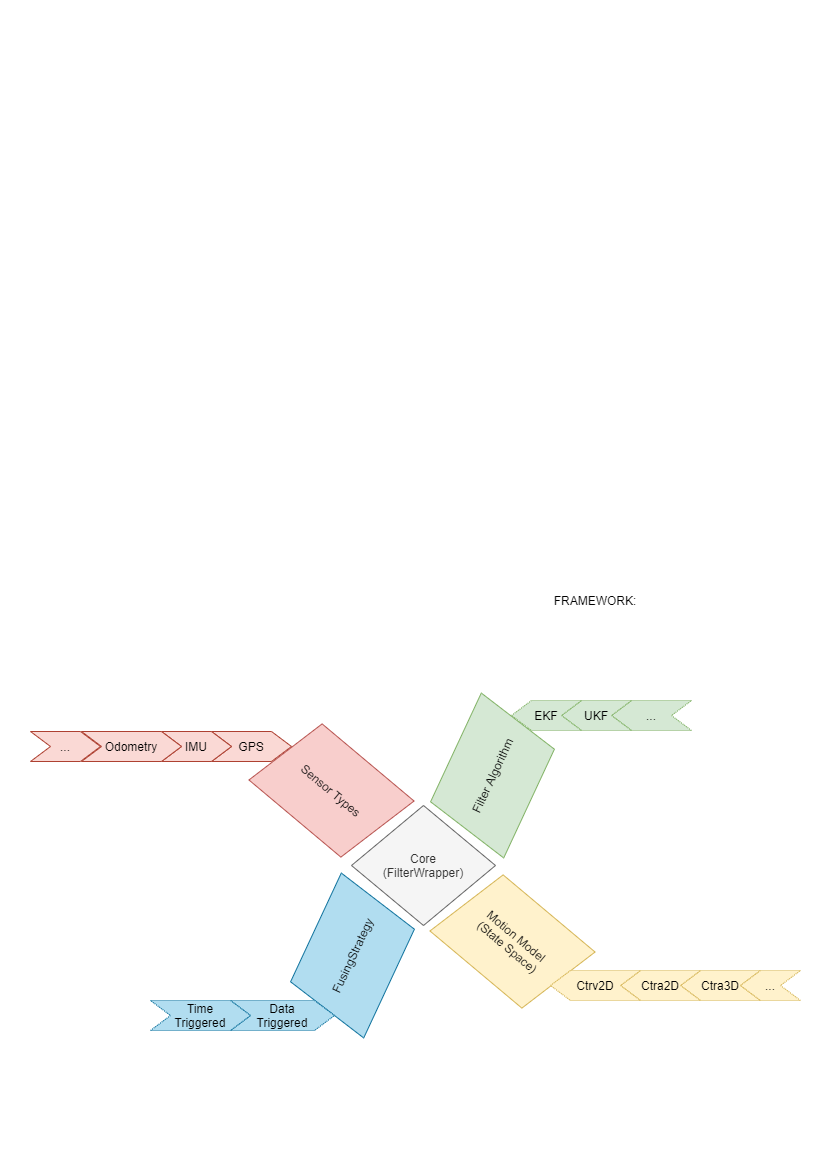
Update the README for the package with documentation on how to use.

# Software Architecture

As mentioned before the library offers modularity and genericity which makes it flexible for all kinds of sensor fusion and frameworks to be used with.

## Filter Core

The core of the library consists of four modules: the sensor types, filter algorithms, fusing strategies and motion models. These can all be combined in a desired way. The FilterWrapper is the center of the core and the first point where the measurements come in before being transformed to an internal format for further usage. Here all modules are put together and the necessary callbacks are exchanged.



## Motion Model

The motion models we implement predicts a state where the velocity/acceleration parts are all car-frame based whereas absolute values like pose and orientation are world-frame base.

We want to allow both 2D and 3D based motion models[[2]](#footnote-1)[[3]](#footnote-2). In difference to robot\_localization library we don't always use the same 3D motion model and project it in 2D if necessary but allow the usage of different 2D/3D motion models. The motivation is saving up computations for the 2D case and allowing the motion model choice to be generic in general that could open up tracking other important values such as different biases, e.g. acceleration bias of magnetometer.

**Design**

The motion model serves as an interface which:

* provides a mean to predict the state vector according to the model from the actual state and a delta\_time
* provides the computation of the process jacobian on a given state and delta\_time

Some considerations for the motion model:

* it doesn't own any member variable
* the outputs are given per reference to avoid unnecessary copying of matrices
* to avoid computation errors for the Jacobian-predict step and to save some computation we offer a method that does both compute\_jacobian\_and\_predict().

**Input / Outputs / API**

The following inputs are provided:

* State (predict(), compute\_jacobian(), compute\_jacobian\_and\_predict())
* Delta time (predict(), compute\_jacobian(), compute\_jacobian\_and\_predict())

The following outputs are provided:

* State (predict(), compute\_jacobian\_and\_predict())
* Jacobian (compute\_jacobian(), compute\_jacobian\_and\_predict())

### 2D Models

#### **CTRV (Constant turning rate and velocity)**

This is a nonlinear motion model. It assumes constant velocity, i.e. accelerations are not considered at all and velocity changes can come only from the sensors directly.  
The CTRV motion model has the following state:

x // x position

y // y position

yaw // orientation

vx // x velocity

vy // y velocity

vyaw // turning rate

With the following update equations:  
  
x = x + dt\*vx\*cos(yaw) - dt\*vy\*sin(yaw)  
y = y + dt\*vx\*sin(yaw) + dt\*vy\*cos(yaw)  
yaw = yaw + dt\*vyaw  
vx = vx  
vy = vy  
vyaw = vyaw

Resulting in the following jacobian:  
  
1 0 -dt\*vx\*sin(yaw)-dt\*vy\*cos(yaw) dt\*cos(yaw) -dt\*sin(yaw) 0  
0 1 dt\*vx\*cos(yaw)-dt\*vy\*sin(yaw) dt\*sin(yaw) dt\*cos(yaw) 0  
0 0 1 0 0 dt  
0 0 0 1 0 0  
0 0 0 0 1 0  
0 0 0 0 0 1

#### 

#### **CTRA (Constant turning rate and acceleration)**

This nonlinear motion model assumes constant turn rate and acceleration. The turn rate is assumed to be independent of all other parameters, such as velocity or acceleration.

The CTRA motion model has the following state:

x // x position

y // y position

yaw // orientation

vx // x velocity

vy // y velocity

vyaw // turning rate

ax // x acceleration

ay // y acceleration

With the following update equations:  
  
x = x + dt\*vx\*cos(yaw) - dt\*vy\*sin(yaw) + 0.5\*dt^2\*ax\*cos(yaw) - 0.5\*dt^2\*ay\*sin(yaw)  
y = y + dt\*vx\*sin(yaw) + dt\*vy\*cos(yaw) + 0.5\*dt^2\*ax\*sin(yaw) + 0.5\*dt^2\*ay\*cos(yaw)  
yaw = yaw + dt\*vyaw  
vx = vx + dt\*ax  
vy = vy + dt\*ay  
vyaw = vyaw  
ax = ax  
ay = ay

Resulting in the following jacobian:  
  
1 0 -dt\*vx\*sin(yaw)-dt\*vy\*cos(yaw)-0.5\*dt^2\*ax\*sin(yaw)-0.5\*dt^2\*ay\*cos(yaw) dt\*cos(yaw) -dt\*sin(yaw) 0 0.5\*dt^2\*cos(yaw) -0.5\*dt^2\*sin(yaw)  
0 1 dt\*vx\*cos(yaw)-dt\*vy\*sin(yaw)+0.5\*dt^2\*ax\*cos(yaw)-0.5\*dt^2\*ay\*sin(yaw) dt\*sin(yaw) dt\*cos(yaw) 0 0.5\*dt^2\*sin(yaw) 0.5\*dt^2\*cos(yaw)  
0 0 1 0 0 dt 0 0  
0 0 0 1 0 0 dt 0  
0 0 0 0 1 0 0 dt  
0 0 0 0 0 1 0 0  
0 0 0 0 0 0 1 0  
0 0 0 0 0 0 0 1

**3D Models**This model follows the same logic as the 2D motion model to extend the model for the 3d dimension(z-direction).

**CTRA (Constant turning rate and acceleration)**The derivations get more complicated as we now have 2 more directions to consider.  
The CTRA3d motion model has the following state:

x // x position

y // y position

z // z position

roll // x orientation

pitch // y orientation

yaw // z orientation

vx // x velocity

vy // y velocity

vz // z velocity

vroll // x turning rate

vpitch // y turning rate

vyaw // z turning rate

ax // x acceleration

ay // y acceleration

az // z acceleration

With the following update equations:

cr := cos(cr); cp := cos(cp); cy := cos(cy)

sr := sin(sr); sp := sin(sp); sy := sin(sy)

tp := tan(pitch)

x = x + dt\*vx\*cy\*cp + dt\*vy\*(cy\*sp\*sr-sy\*cr) + dt\*vz\*(cy\*sp\*cr+sy\*sr) + 0.5\*dt^2\*ax\*cy\*cp + 0.5\*dt^2\*ay\*(cy\*sp\*sr-sy\*cr) + 0.5\*dt^2\*az\*(cy\*sp\*cr+sy\*sr)

y = y + dt\*vx\*sy\*cp + dt\*vy\*(sy\*sp\*sr+cy\*cr) + dt\*vz\*(sy\*sp\*cr-cy\*sr) + 0.5\*dt^2\*ax\*sy\*cp + 0.5\*dt^2\*ay\*(sy\*sp\*sr+cy\*cr) + 0.5\*dt^2\*az\*(sy\*sp\*cr-cy\*sr)

z = z + -dt\*vx\*sp + dt\*vy\*cp\*sr + dt\*vz\*cp\*cr + 0.5\*dt^2\*ax\*sp + 0.5\*dt^2\*ay\*cp\*sr + 0.5\*dt^2\*az\*cp\*cr

roll = roll + dt\*vroll + dt\*sr\*tp\*vpitch + dt\*cr\*tp\*vyaw

pitch = pitch + dt\*cr\*vpitch - dt\*sr\*vyaw

yaw = yaw + dt\*sr/cp\*vpitch + dt\*cr/cp\*vyaw

vx = vx + dt\*ax

vy = vy + dt\*ay  
vz = vz + dt\*azvroll = vroll; vpitch = vpitch; vyaw = vyaw

Resulting in the following jacobian:  
  
dx/droll = dt\*vx\*0 + dt\*vy\*(cy\*sp\*cr+sy\*sr) + dt\*vz\*(-cy\*sp\*sr+sy\*cr) + 0.5\*dt^2\*ax\*0 + 0.5\*dt^2\*ay\*(cy\*sp\*cr+sy\*sr) + 0.5\*dt^2\*az\*(-cy\*sp\*sr+sy\*cr)

dx/dpitch = -dt\*vx\*cy\*sp + dt\*vy\*cy\*cp\*sr + dt\*vz\*cy\*cp\*cr - 0.5\*dt^2\*ax\*cy\*sp + 0.5\*dt^2\*ay\*cy\*cp\*sr + 0.5\*dt^2\*az\*cy\*cp\*cr

dx/dyaw = -dt\*vx\*sy\*cp + dt\*vy\*(-sy\*sp\*sr-cy\*cr) + dt\*vz\*(-sy\*sp\*cr+cy\*sr) - 0.5\*dt^2\*ax\*sy\*cp + 0.5\*dt^2\*ay\*(-sy\*sp\*sr-cy\*cr) + 0.5\*dt^2\*az\*(-sy\*sp\*cr+cy\*sr)

dy/droll = dt\*vx\*0 + dt\*vy\*(sy\*sp\*cr-cy\*sr) + dt\*vz\*(-sy\*sp\*sr-cy\*cr) + 0.5\*dt^2\*ax\*0 + 0.5\*dt^2\*ay\*(sy\*sp\*cr-cy\*sr) + 0.5\*dt^2\*az\*(-sy\*sp\*sr-cy\*cr)

dy/dpitch = -dt\*vx\*sy\*sp + dt\*vy\*sy\*cp\*sr + dt\*vz\*sy\*cp\*cr - 0.5\*dt^2\*ax\*sy\*sp + 0.5\*dt^2\*ay\*sy\*cp\*sr + 0.5\*dt^2\*az\*sy\*cp\*cr

dy/dyaw = dt\*vx\*cy\*cp + dt\*vy\*(cy\*sp\*sr-sy\*cr) + dt\*vz\*(cy\*sp\*cr+sy\*sr) + 0.5\*dt^2\*ax\*cy\*cp + 0.5\*dt^2\*ay\*(cy\*sp\*sr-sy\*cr) + 0.5\*dt^2\*az\*(cy\*sp\*cr+sy\*sr)

dz/droll = -dt\*vx\*0 + dt\*vy\*cp\*cr - dt\*vz\*cp\*sr + 0.5\*dt^2\*ax\*0 + 0.5\*dt^2\*ay\*cp\*cr - 0.5\*dt^2\*az\*cp\*sr

dz/dpitch = -dt\*vx\*cp - dt\*vy\*sp\*sr - dt\*vz\*sp\*cr + 0.5\*dt^2\*ax\*cp - 0.5\*dt^2\*ay\*sp\*sr - 0.5\*dt^2\*az\*sp\*cr  
  
dz/dyaw = -dt\*vx\*0 + dt\*vy\*0 + dt\*vz\*0 + 0.5\*dt^2\*ax\*0 + 0.5\*dt^2\*ay\*0 + 0.5\*dt^2\*az\*0  
droll/droll = dt\*cr\*tp\*vpitch - dt\*sr\*tp\*vyaw;  
droll/dpitch = dt\*sr/cp^2\*vpitch + dt\*cr/cp^2\*vyaw;  
droll/dyaw = dt\*0\*vpitch + dt\*0\*vyaw;  
dpitch/droll = -dt\*sr\*vpitch - dt\*cr\*vyaw  
dpitch/dpitch = dt\*0\*vpitch - dt\*0\*vyaw  
dpitch/dyaw = dt\*0\*vpitch - dt\*0\*vyaw  
dyaw/droll = dt\*cr/cp\*vpitch - dt\*sr/cp\*vyaw  
dyaw/dpitch = dt\*sr\*tp/cp\*vpitch + dt\*cr\*tp/cp\*vyaw  
dyaw/dyaw = dt\*0\*vpitch + dt\*0\*vyaw

## Input Sensor Types

**Design**

The sensor module serve as transformation units that :

* transform the measurement parts together with their covariances in the fusing frame
* write them into an internal format of measurements that is the same for all kinds of measurements and provide the necessary information for the fusion such as state\_to\_measurement mapping matrix, covariance, time\_stamp etc.

Some considerations for the sensors:

* they handle errors in the incoming measurements
* make sure the covariance isn’t ill conditioned

**Input / Outputs / API**

The following inputs are provided:

* Odometry
* ImuMsg
* GpsMsg

The following outputs are provided:

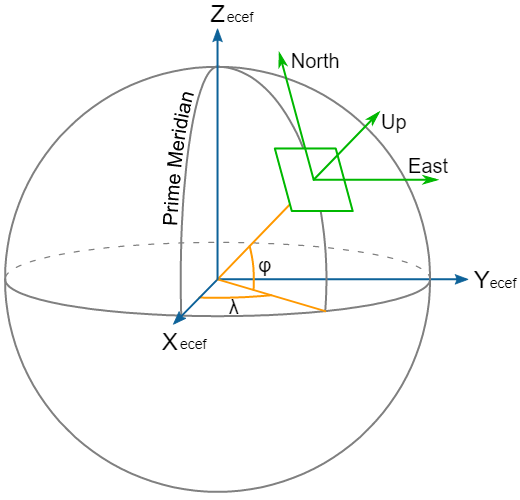
* Measurement

As mentioned before the supported sensor measurement types are Odometry, IMU and GPS. Nevertheless, it is not quite straight forward on how to come to these types of messages from cameras or lidars. We decided to keep the library simple and focus only on fusion. For such scenarios, other external libraries are to be used or the sensor models are to build on your own.

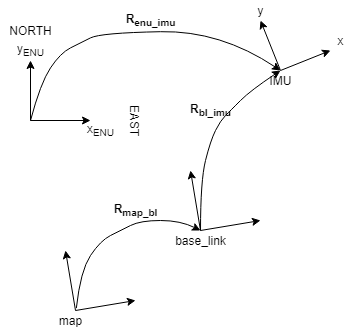
Before we continue it is important to clarify how the measurements from each sensor are fused. The state consists normally of a pose part which includes a position and an orientation, a velocity part(angular & linear) and acceleration. The parts of messages that directly give information about the pose part are firstly transformed in the map(odom) frame before being fused. All velocities and accelerations are fused in the base\_link frame which identifies the mobile robot. While Odometry messages are quite straightforward to be fused, IMUs and GPUs present a challenge as they offer absolute measurements in an odometry system. For this reason I will briefly explain their fusion before continuing.

### IMU

In the following I will refer to IMUs equipped with magnetometer and accelerometer, which I will call *fully equipped*. If one of the both sensors would miss, the orientations presented by IMU would have as reference some power-on initial coordinate system which we may or may not make use of, as the orientation would then be estimated only from the angular velocities which we (could) fuse directly. As I understand, *fully equipped* IMUs give out orientations referenced to a world fixed frame. In most scenarios these world fixed frames are ENU(x-east, y-north and z-up) or NED( x-north, y-east, z-down) based. Both coordinate frames are formed from a plane tangent to the Earth's surface. As shown in the photo below taken from [wikipedia](https://en.wikipedia.org/wiki/Local_tangent_plane_coordinates).

We are interested only in the orientation of the ENU(NED) coordinate system relative to the orientation of our world coordinate system and not for its origin as we deal only with rotations.

The situation is the following:

1. We get orientations in the world fixed frame (roll\_ENU, pitch\_ENU, yaw\_ENU) of the IMU sensor **(R\_enu\_imu\_meas)**
2. We know how the IMU sensor is oriented in our vehicle and we have an estimate of it from our map frame **(R\_map\_imu = R\_map\_bl \* R\_bl\_imu)**
3. We want to know how to transform these ENU referenced orientation measurements to our map frame **(R\_map\_imu\_meas)**
4. Afterwards we just need to make up for the orientation of the IMU on the vehicle **(R\_map\_bl\_new = R\_map\_imu\_meas \* R\_bl\_imu^-1)**

The only difficulty in all of this is how to get **R\_map\_enu** transformation which would allow us to express the measurements in map frame as: **R\_map\_imu\_meas = R\_map\_enu \* R\_enu\_imu\_meas**. As everything is built upon this logic it is important to make sure that the quality of R\_map\_enu is of good quality.

There are 2 possible solutions for that:

1. We provide the Power-On Orientation of the Vehicle in ENU coordinates so **(R\_map\_imu = R\_enu\_imu)**
2. We make use of the first(or average of N first) IMU measurement to calculate **R\_map\_enu = R\_map\_bl \* R\_bl\_imu \* R\_enu\_imu^-1**. And then use that to transform the future measurements. This method has the other advantage that it can be used for arbitrary world fixed reference too. Maybe even for the case where the IMU is not *fully equiped*. But it comes with a bias depending on how bad our orientation estimation **R\_map\_bl** is at that time.
3. Finally we fuse **R\_map\_bl\_meas = R\_map\_enu \* R\_enu\_imu\_meas \* R\_bl\_imu^-1**

The above logic seems to work in practise but there are still 2 problems that I couldn’t find a solution to.

**1.Transformation of Covariances:**

While transforming the orientations is straight forward: **R\_map\_bl\_meas = R\_map\_enu \* *R\_enu\_imu\_meas* \* R\_bl\_imu^-1**, transforming the covariances is not so obvious because our orientation **R\_enu\_imu\_meas** is once rotated and once used to rotate **R\_bl\_imu^-1**. The rotation with R\_map\_enu corresponds to a covariance **C\_new = R\_map\_enu \* C \* R\_map\_enu^T**. *How to consider the second rotation*?

**2.Euler angles are not continuous:**

The usage of *Euler Angles* comes with a Gimbal Lock which causes loss of DOF if pitch is pi/2. I know too that choosing the order of euler angles could avoid this discontinuity for specific scenarios. Since the Eigen Library only provides transformations based on the ‘sxyz’ ordering, I was forced to write my own functions for the transformations: **euler\_angles <-> quaternion <-> rotation\_matrix.**

The origin of the problem comes from the fact that euler angles are used in the state instead of quaternion which is gimbal free. So the question would be: ***Is there any way that we could have the quaternion as part of the state instead of RPY or does it introduce complications that does not justify its advantages?***

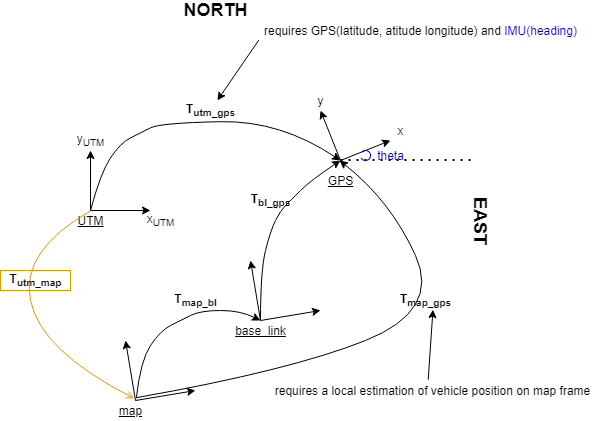
### GPS

GPS msgs come in the form of longitude, latitude and altitude. In order to be able to fuse such measurements we should be able to transform them in a local world fixed cartesian coordinate system.   
  
We're interested in a transform from lat/long coordinates to our robot's map frame. As we don't want to deal directly with lat/longs, the first thing is to convert the lat/long coordinates into [UTM](https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system) coordinates. This gives us our GPS location in a Cartesian coordinate frame. As I understand this coordinate system corresponds to small regions(~around 360km) and have to be changed if we pass to a new region. For example in [Germany](https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system#/media/File:LA2-Europe-UTM-zones.png) we have 32U and 33U. Nevertheless I assume for now that this coordinate system is fixed and trust the geography\_lib to take care of it. In addition to that I assume as in robot\_localization that this coordinate system has an ENU orientation(x-east, y-north, z-up).  
  
What we want to know is the position and orientation of our robot's world frame in the UTM grid. That corresponds on knowing **T\_map\_utm**, so that every position given in UTM frame can transformed into map frame by multiplying:   
**x\_map = T\_map\_utm \* x\_utm**.

How to find this transformation **T\_map\_utm**?

We can find this transformation as long as we know one of the transformations from **T\_utm\_x** where **x** can be any of the frames connected to map and gps frame, i.e. we have already estimated **T\_map\_x**. So we can just write: **T\_map\_utm = T\_map\_x \* T\_utm\_x^-1**. This **x** in our case would be the the gps frame, so **T\_map\_utm = T\_map\_gps \* T\_utm\_gps^-1**.

1. **T\_map\_gps**: can be acquired from the pose estimation of the vehicle in map frame (**T\_map\_bl**) and the pose of the sensor in vehicle frame(**T\_bl\_gps**) as **T\_map\_gps= T\_map\_bl \* T\_bl\_gps**.
2. **T\_utm\_gps**: is a little harder to get. First of all from the information that we get from the GPS sensor can be converted only into a position without orientation (**P\_utm\_gps**). But in order to create the full transformation we require third orientation. This orientation should be either provided through a configuration file or provided by an IMU sensor. For the second case this translates to knowing **R\_enu\_imu**. Since the UTM and ENU are assumed to have the same orientation we have **R\_utm\_imu = R\_enu\_imu**. So **R\_utm\_gps = R\_utm\_imu \* R\_imu\_gps** where **R\_imu\_gps** is given since we know the orientations of both IMU and GPS relative to the vehicle frame(base\_link).

This transformation **T\_map\_utm** can then be used to interpret all positions provided by the GPS in the map frame.

So no mater of the cases [(a)&(b)](https://github.com/cra-ros-pkg/robot_localization/issues/550#issuecomment-606466118) where the vehicle starts with an offset angle from the ENU or the vehicle is in a GPS-denied area for some time, and get its first GPS reading when it is at some later pose in the map we should be able to still treat the problem the same.

**Note**: as both orientation from IMU and position from GPS are absolute we could even average multiple estimations of **T\_map\_utm** in order to avoid any unwanted bias.

## Filter Algorithms

**Extended Kalman Filter[[4]](#footnote-3)** is the most common filter used for state estimation in sensor fusion. It allows the estimation of a state through a mean and a covariance. It assumes that all noises are white and Gaussian. It is called extended as it deals with the nonlinearities of the prediction step by linearization through the first term of Taylor series.

The method consists of a predict step and an update step.

In each step both the state estimation and its covariance are changed.

**Predict**

Predict state estimate x = predict(x,dt)

Predict covariance estimate P = JPJT + Q

where J is the Jacobian of predict() and Q the process noise.

**Update**

Innovation dz = z - Hx

Calculate Kalman Gain K = PHT (HPHT +R)-1

Update covariance estimate P = (I -KH)P(I\_KH)T

Update the state estimate x = x + Kz

where H is the state to measurement mapping matrix and R the observation noise. The covariance is updated through the Joseph form which deviates from the standard way for stability reasons when the covariance is not well conditioned.

## Fusion Strategies

As measurements can arrive in an unpredictable order it is important to choose a strategy for the fusion pipeline. The straight forward way is the data triggered approach.

### Data triggered

Measurement arrives:

└> if older than last used measurement: ignore the measurement  
└> if newer: predict to measurement time, correct the state with the measurement and publish new estimate

But this simple approach comes with a problem. In general we have to deal with delayed measurements. These are measurements that are captured at a time point but have to be processed before being fused, which delays their arrival. A good example for this is visual odometry. The data triggered approach would deal with this by ignoring them.

**Problem**: Delayed measurements + IMU works with: 100 Hz  
 + Visual odometry: 10Hz

Time ***t=0***: Incorporate an IMU measurement

Time ***t=0.1***: A Visual Odometry measurement is produced but is ready to fuse only at . time ***t=0.15*** after being preprocessed

Time ***t=0.11***: Another IMU measurement arrives and is incorporated

└> the last IMU measurement classifies the Visual Odometry as old

This would mean it would be not possible to use visual odometry at all.

### Time triggered

The idea here ist to keep a history on the fused measurements and estimated states. In addition to that all new measurements are inserted and sorted in a buffer according to their time stamp. The whole pipeline is triggered by a periodic time triggered function that runs in a separate thread. This function is always called with the timestamp up to each the measurements in the buffer have to be integrated. The frequency on which we call this function decides the frequency at which we publish the state estimate too.

The function deals with the buffered measurements as following:

* it integrates all measurements with timestamp older than t
* if any of the measurements is older than the current time of the filter
  + wo go back to the first state with older timestamp than this measurement and just ignore the newer states
  + insert all history measurements up to the state’s timestamp in the measurement buffer which we then can fuse again afterwards
* after any measurement integration add measurement and state in their respective history\_dequeues
* if no measurement in the buffer still keep updating up to the current time t

In such a way we can deal with delayed unsynchronized measurements successfully. In addition to that such approach allows for a threaded input pipeline where each sensor can be run in its own separate thread.

Stempel und Unterschrift des Praktikumbetreuers

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. https://www.iav.com/news/erster-autonomer-shuttlebus/ [↑](#footnote-ref-0)
2. <http://fusion.isif.org/proceedings/fusion08CD/papers/1569107835.pdf> [↑](#footnote-ref-1)
3. <https://perso.uclouvain.be/georges.bastin/paper37.pdf> [↑](#footnote-ref-2)
4. https://en.wikipedia.org/wiki/Extended\_Kalman\_filter [↑](#footnote-ref-3)