Final Report Concept Phase

Overview (not important)

The best overview of the library can be obtained by following the life of a sensor message through a filter. The message's life goal is to make the filter a better place by improving its state estimate.
It is born by a sensor and shaped to take a standard form by its drivers. No matter the form, all messages contain measurements relative to their sensors and in best case covariances(expected squared-errors) they come with. Beside that, the messages have a certificate(header) which contains information about their time of birth(time_stamp) and birthplace(frame_id).The time_stamp allows us to tell if a message is older than another and so treat them differently. The frame_id enables a global interpretation of the measurements independently of the sensor it came from.
Just after creation these messages trigger some callback functions which immediately transform them into an internal measurement format for a *unified* usage in the future. Our standard formats consist of: *nav_msgs/Odometry*, *geometry_msgs/PoseWithCovarianceStamped*, *gemoetry_msgs/TwistWithCovarianceStamped* and *sensor_msgs/Imu*.
Whether or not we use the time-triggered options the measurements either migrate into a time-sorted queue and wait to be processed with a constant frequency or are immediately processed. The processing consists of a temporal_update which makes a prediction of the state to the next time step and so increases the uncertainty, followed by a correction step which makes use of the measurement values and their covariances to correct the predicted state. This signalizes the end of life for messages in a data-triggered framework. We would keep them a little longer if the process would run in a time-triggered manner. This would assure us to consider measurements older than the measurements already processed. Finally the estimated state and its covariance matrix are shaped into odmoetry_msg and published, which allows other filters to make use of it.

Motion Model

MotionModel consists of a static functional class with no member variables. It consist of 3 functions that can update the state according to a dt, compute the Jacobian at a certain state or do both in the right order. The motivation for using a static class is to avoid having a seperate MotionModel object for each filter(static methods are the same through all the objects of that class in our case).
One could have only functions and use a namespace to form an interface, but that cannot be used as template parameter for our filter.
As a start we implement 2 models,a 3D MotionModel and the Ackermann 2D MotionModel(corresponding to the CATR MotionModel).

Measurement

The motivation of having a Measurement class is to represent sensor-msgs in a unified format. It holds a time_stamp and overloaded operators for comparing in terms of time_stamp. In addition to that it stores the measurement vector, covariance matrix update_vector. The last member is a true/false vector of size num_states, that shows with false the part of measurement to be ignored. In difference to other libraries, we store only the values of the sub-measurement($\underline{z}$**)** and the corresponding sub-covariance($\underline{R}$**)** that are marked with true in update_vector. The sub-jacobi($\underline{H}$) is a state to sub_measurement mapping and can be computed on request.

$$\underline{z} = \underline{H}x$$

The size of members *measurement* and *covariances* could be decided in compile-time if we give the size of measurement as template parameter(needs further discussion, not very sure)

Measurement Conversions

Input: msg, update_vector
Output: measurement

These functions enable the creation of Measurement-objects for each sensor-msg.
They:

- extract the measurement values and covariances
- transform/rotate them together with update_vector.
- Reduce the measurement values and covariances only to the part described by the update vector.
- return the measurement with reduced values and covariances and rotated update_vector and the threshold

For further info look into robot_localization's:

ekf.cpp
└──Ekf::correct (Row: 69-164)

MeasurementTimeKeeper

Class that keeps the time relative to the timestamp of the last measurement. By doing that it enables both unsynchronized msgs(e.g. Played from bag) and synchronized ones that match to the global time. It does that by keeping the *global receiving time* and *time stamp* of the last measurement and the *last temporal update time* in the frame of the time stamp. The difference of global receiving time and time stamp defines a shift to converse times from global to time_stamp frame vice-versa.

FilterHierarchy

The filter hierarchy is organized in such a way that it is generic(allows different template parameters: *Filter(EFK,UFK,SqrtEKF..), Framework(ROS, ADFT), MotionModel/State size, Sensors)* and simple(clearly defined which parts are responsible for each function) to allow changes in a structured way.

***TLDR*:**
FilterBase: Responsible for fusing algorithm and keeps the estimated state
FilterWrapper: Offers the necessary interface to FitlerNode and takes care of time-keeping and together with FilterNode implements the time-triggered and data-triggered options. To be precise it keeps the measurement queue, does its processing and makes sure that everything is properly initialized before usage.
FilterNode: Wrapps FitlerWrapper and implements the part of the library that use framework's(ROS/ADFT) components. In addition to that it does the configuration parsing and publishes the estimated state/covariance for further usage.
FilterAPI: Hides the backend from the user.

FilterBase

FilterBase is an abstract class which is responsible for the fusing algorithm and uses the motion model. All different filter algorithms should inherit from it and define all its methods. It is the only class that stores the state/covariance to be estimated and for that, offers corresponding getters to ask for it.

FilterWrapper (wrapps FilterBase)

FilterWrapper's is a class that provides the necessary interface to the FilterNode through FilterAPI. It keeps the time for the filter and holds the configurations of all sensors and of the filter.
Its main method, *process_measurement*(firstly used only with the data-triggered option) transforms the measurements in the internal format and then calls *temporial_update* & *observation_update* steps.

We keep the methods *temporial_update* and *observation_update* seperated since they are required to be called separately when time-triggered option is used. Because of that the case difference takes place in *process_measurement* which is called from the FilterNode callbacks and either goes further to update the state estimation or just puts the measurement in the buffer.

FilterAPI

FilterAPI is hides the backend of the library and only offers to the users the necessary functions such as the callback functions, configure and get_state().

FilterNode

This class serves as the interface to ROS. In case one wants to use ADFT the functions have to be changed accordingly. It parses ROS related configuration parameters and owns callback functions for each possible sensor-msg. Some important member variables it stores are different frames such as odom, base_link, map_frame and sensor_frames.
All callbacks form the transformation matrix and together with the measurement values/covariance call the corresponding functions of the FilterAPI. If data_triggered is provided the state estimation is published afterwards.

Data- vs Time-triggered measurement processing

The goal is to make it possible through a bool variable to choose between data- or time-triggering. The time-triggered framework would require an additional queue for the measurements, a separate thread to process them continuously and a publisher for ROS/ADFT. With that said it would require changes in FilterWrapper and in FilterNode(extra publisher). For the development, we start with the data-triggered way and expand it in a further iteration.