ZAFTIG VTOL ESTIMATOR

Kenneth James Jensen

February 20, 2009

The control algorithms for Zaftig require an accurate estimation of the state of the kite including position, velocity, orientation, and angular velocity. Though many of these states are measured directly (*i.e.* angular rates), other states are "hidden" such as the angle of rotation about an axis parallel to the tether or are not measured with sufficient accuracy. Thus, it is necessary to introduce a state estimator, which combines the various measurements with a model of the kite. Here we describe an extended Kalman filter developed specifically for Zaftig in a VTOL flight mode.

Document ID number:

Revision History:

Version:	Comments:
_____	_____
_____	_____

Submitted by:

_____ date: _____

Received/Approved by:

_____ date: _____

Contents

1. Introduction

To accurately estimate the state of Zaftig, it is necessary to combine information from the various sensors including a quickly updating inertial measurement unit (IMU) and a host of more slowly updating position and orientation sensors. Similar sensor fusion and state estimation problems appear in all inertial navigation systems (INSs), and consequently there are mature techniques for approaching these problems. Many of the design decisions for this estimator are based on what has worked well in previous INSs.

Before delving into the details of the estimator, we give a broad overview here. The estimator uses an extended Kalman filter (EKF). There are of course numerous other potential means of estimating state (*e.g.* unscented Kalman filters, particle filters, *etc...*), some of which claim to be superior to the EKF; however, we chose the EKF because of its extensive use in INSs and the wealth of information available on its application.

The estimator uses a "tightly-coupled" topology, meaning that information from all the sensors and states are used together. This is opposed to a "loosely-coupled" topology where the IMU is separate from the rest of the Kalman filter, which is only used to correct offsets and update biases. Many INSs take the loosely-coupled approach. However, because of the highly dynamic nature of the kite system, we felt that the tightly-coupled approach could yield significant benefits. The disadvantage of the tightly-coupled approach is the increase in complexity and computation time.

The estimator does not, as of yet, incorporate a detailed model of the kite. Instead, it models the kite as a rigid body subject to white noise accelerations. Again, this simple model is often used in INSs. Of course, incorporating an accurate model of the kite would improve filter performance. However, this would significantly complicate filter development and require that the filter be revalidated for each new kite.

Though this estimator is currently only intended to be used for VTOL, it was designed to be easily extended to crosswind flight conditions. Thus, we did not make certain simplifications, such as using a linear Kalman filter, which would suffice for VTOL. To extend the current estimator to a crosswind estimator, it should only be necessary to add pitot tube measurements, add logic for switching between VTOL mode and crosswind mode, and test how the filter performs in the dynamic conditions of crosswind flight.

Lastly, we note that the EKF is implemented in a different coordinate system than the simulator. During VTOL, the kite should, hopefully, be pointing close to upwards. In the simulator coordinate system, this results in Euler angles close to the $\pi/2$ singularity, which could cause problems for potential controllers. Thus, we adopt a global coordinate system centered at the GLAS, but with axes oriented in the same manner as the kite's body axes if the kite were pointed straight upward. We call this coordinate system the *nominal* coordinate system. Of course, it is trivial to convert these coordinates to the standard or other coordinates if necessary.

2. Continuous-Discrete Extended Kalman Filter

The extended Kalman filter is described in detail in numerous references, so we only briefly review it here. The EKF is divided into two steps: propagation and update. During the propagation step, the estimated state $\hat{\mathbf{x}}(t)$ ("$\hat{}$" indicates an estimate) and the covariance matrix $\mathbf{P}(t)$ are integrated based solely on the model of the system (*i.e.* no new information from measurements). During the update step, the estimated state before the measurement $\hat{\mathbf{x}}_k(-)$ and the covariance before the measurement $\mathbf{P}_k(-)$ are updated to include the new information from the measurements to become $\hat{\mathbf{x}}_k(+)$ and $\mathbf{P}_k(+)$ ("$-$" indicates a pre-measurement estimate and the "$+$" indicates a post-measurement estimate).

The continuous propagation equations are:

$$\dot{\hat{\mathbf{x}}}(t) = \mathbf{f}(\hat{\mathbf{x}}(t), t) + \mathbf{w}(t) \tag{1}$$

$$\dot{\mathbf{P}}(t) = \mathbf{F}(\hat{\mathbf{x}}(t), t)\mathbf{P}(t) + \mathbf{P}(t)\mathbf{F}^T(\hat{\mathbf{x}}(t), t) + \mathbf{Q}(t) \tag{2}$$

Here $\mathbf{f}$ is a non-linear function describing the model, $\mathbf{w}$ is the "process" noise, and $\mathbf{Q}$ is the process noise covariance matrix. $\mathbf{F}(\hat{\mathbf{x}})$ is the linearized system model about the estimate $\hat{\mathbf{x}}$ defined by

$$\mathbf{F}(\hat{\mathbf{x}}(t), t) \equiv \left.\frac{\partial \mathbf{f}(\mathbf{x}(t), t)}{\partial \mathbf{x}(t)}\right|_{\mathbf{x}(t)=\hat{\mathbf{x}}(t)}. \tag{3}$$

The discrete update equations are:

$$\hat{\mathbf{x}}_k(+) = \hat{\mathbf{x}}_k(-) + \mathbf{K}_k\left[\mathbf{z}_k - \mathbf{h}_k(\hat{\mathbf{x}}_k(-))\right] \tag{4}$$

$$\mathbf{P}_k(+) = \left[\mathbf{I} - \mathbf{K}_k\mathbf{H}_k(\hat{\mathbf{x}}_k(-))\right]\mathbf{P}_k(-) \tag{5}$$

with the Kalman gain matrix given by

$$\mathbf{K}_k = \mathbf{P}_k(-)\mathbf{H}_k^T(\hat{\mathbf{x}}_k(-))\left[\mathbf{H}_k(\hat{\mathbf{x}}_k(-))\mathbf{P}_k(-)\mathbf{H}_k^T(\hat{\mathbf{x}}_k(-)) + \mathbf{R}_k\right]^{-1} \tag{6}$$

Here $\mathbf{z}$ is the actual measurement, $\mathbf{h}$ is the non-linear measurement equation that converts an estimated state $\hat{\mathbf{x}}_k(-)$ to an estimated measurement $\hat{\mathbf{z}}$ and $\mathbf{R}$ is the measurement noise covariance matrix. $\mathbf{H}$ is the linearized measurement matrix about an estimate defined by

$$\mathbf{H}_k(\hat{\mathbf{x}}_k(-)) \equiv \left.\frac{\partial \mathbf{h}_k(\mathbf{x}(t_k))}{\partial \mathbf{x}(t_k)}\right|_{\mathbf{x}(t_k)=\hat{\mathbf{x}}_k(-)}. \tag{7}$$

3. State vector and model selection

The primary design choices during development of a Kalman filter are the selection of a state vector and system model. We chose the state vector $\mathbf{x}$ defined by:

$$\mathbf{x} = \begin{bmatrix} \mathbf{r} & \dot{\mathbf{r}} & \ddot{\mathbf{r}} & \mathbf{q} & \omega & \dot{\omega} & \mathbf{b}^{\text{acc}} & \mathbf{b}^{\text{gyro}} \end{bmatrix}^T \tag{8}$$

where $\mathbf{r}$ is the kite vector in the nominal coordinate system, $\mathbf{q}$ is the quaternion describing a rotation from the nominal coordinate system to the kite coordinate system, ω is the angular rate, and $\mathbf{b}^{\text{acc}}$ and $\mathbf{b}^{\text{gyro}}$ are the accelerometer and gyro biases. This state vector contains the standard states: position, velocity, orientation, and angular rates. However, it also includes acceleration and angular acceleration. In the literature, this is termed a position, velocity, and acceleration (PVA) state vector and is commonly recommended for highly dynamic vehicles. Future versions of the filter may incorporate other states such as accelerometer/gyro scale factors and misalignment matrices, or may even delete states which prove unnecessary.

The model of the system is simply a rigid body subject to white noise linear and angular accelerations. This model can be tailored, to some extent, for a specific kite by adjusting the magnitudes of the process noise accelerations. The biases are also subject to small process noise terms.

3.1. Quaternion representation.

There are numerous ways of representing orientation: Euler angles, direction cosine matrices, Rodrigues parameters, and quaternions, to name just a few of the more popular representations. Each representation has its advantages and disadvantages. Though Euler angles are conceptually the simplest representation, they suffer from singularities at rotations of $\pm\pi/2$. The quaternion representation was chosen as the primary representation for the estimator due to its lack of singularities, its simple quadratic form, and its wide-spread use in INSs.

However, the quaternion representation has its drawbacks as well. It uses four numbers to represent a rotation (which should only require three numbers). Thus because there is a relation between the four numbers, the covariance matrix is singular, which causes numerical problems for the EKF. This problem and some solutions are described in more detail below.

Here we briefly review some of the properties of the quaternion rotation representation. For a rotation of angle θ about the unit vector $\hat{\mathbf{e}} = \begin{bmatrix} e_x & e_y & e_z \end{bmatrix}^T$ the four terms of the quaternion are

$$\mathbf{q} = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix}^T \tag{9}$$

$$\begin{aligned} q_0 &= \cos(\theta/2) \\ q_1 &= e_x \sin(\theta/2) \\ q_2 &= e_y \sin(\theta/2) \\ q_3 &= e_z \sin(\theta/2) \end{aligned} \tag{10}$$

This is the standard ordering in MATLAB; however, beware that other orderings are common!

The norm of the quaternion representation of a rotation is one, as is easily seen from Eq. 10.

$$q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1 \tag{11}$$

The direction cosine matrix, which transforms a vector in nominal coordinates to a vector in body coordinates, is

$$\mathbf{C}_n^b = \begin{pmatrix} 1-2q_2^2-2q_3^2 & 2(q_0q_3+q_1q_2) & 2(q_1q_3-q_0q_2) \\ 2(q_1q_2-q_0q_3) & 1-2q_1^2-2q_3^2 & 2(q_0q_1+q_2q_3) \\ 2(q_0q_2+q_1q_3) & 2(q_2q_3-q_0q_1) & 1-2q_1^2-2q_2^2 \end{pmatrix} \tag{12}$$

The time-derivative of a quaternion is given by the formula

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{\Omega}(\omega)\mathbf{q} \tag{13}$$

where

$$\mathbf{\Omega}(\omega) = \begin{pmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -p & 0 \end{pmatrix} \tag{14}$$

Another common matrix that appears in quaternion algebra is

$$\mathbf{\Xi}(\mathbf{q}) = \begin{pmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{pmatrix} \tag{15}$$

which is simply $2\partial\dot{\mathbf{q}}/\partial\omega$.

4. State propagation details

Given our state vector and model, the equation for state propagation, Eq. 1, is:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) = \begin{bmatrix} \dot{\mathbf{r}} \\ \ddot{\mathbf{r}} \\ 0 \\ \frac{1}{2}\mathbf{\Omega}(\omega)\mathbf{q} \\ \dot{\omega} \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \mathbf{w}^{\text{acc}} \\ 0 \\ 0 \\ \mathbf{w}^{\text{ang}} \\ \mathbf{w}^{\text{a.bias}} \\ \mathbf{w}^{\text{g.bias}} \end{bmatrix} \tag{16}$$

Integrating this equation over a time interval Δt, we determine the propagated state vector, accurate to second order in Δt, to be

$$\hat{\mathbf{x}}_{k+1}(-) \approx \begin{bmatrix} \hat{\mathbf{r}}_k + \dot{\hat{\mathbf{r}}}_k\Delta t + \frac{1}{2}\ddot{\hat{\mathbf{r}}}_k\Delta t^2 \\ \dot{\hat{\mathbf{r}}}_k + \ddot{\hat{\mathbf{r}}}_k\Delta t \\ \ddot{\hat{\mathbf{r}}}_k \\ \left[\left(1-\frac{|\mathbf{\Delta}\hat{\theta}_k|^2}{8}\right)\mathbf{I} + \frac{1}{2}\mathbf{\Omega}(\mathbf{\Delta}\hat{\theta}_k)\right]\hat{\mathbf{q}}_k \\ \hat{\omega}_k + \dot{\hat{\omega}}_k\Delta t \\ \dot{\hat{\omega}}_k \\ \hat{\mathbf{b}}_k^{\text{acc}} \\ \hat{\mathbf{b}}_k^{\text{gyro}} \end{bmatrix} \tag{17}$$

with

$$\mathbf{\Delta}\hat{\theta}_k = \hat{\omega}_k\Delta t + \frac{1}{2}\dot{\hat{\omega}}_k\Delta t^2. \tag{18}$$

State	Measured process noise	Recommended noise	Units
$\ddot{x}$	-	1×10^6	$m^2/s^4/Hz$
$\ddot{y}$	-	1×10^6	$m^2/s^4/Hz$
$\ddot{z}$	-	1×10^6	$m^2/s^4/Hz$
$\dot{p}$	-	1×10^3	$rad^2/s^4/Hz$
$\dot{q}$	-	1×10^5	$rad^2/s^4/Hz$
$\dot{r}$	-	1×10^4	$rad^2/s^4/Hz$
b_x^{acc}	-	1×10^{-6}	$m^2/s^4/Hz$
b_y^{acc}	-	1×10^{-6}	$m^2/s^4/Hz$
b_z^{acc}	-	1×10^{-6}	$m^2/s^4/Hz$
b_p^{gyro}	-	1×10^{-6}	$rad^2/s^2/Hz$
b_q^{gyro}	-	1×10^{-6}	$rad^2/s^2/Hz$
b_r^{gyro}	-	1×10^{-6}	$rad^2/s^2/Hz$

TABLE 1. Process noise

Here we have assumed that axis about which the kite is rotating remains constant over the interval Δt.

To propagate the covariance matrix, we need the matrix $\mathbf{F}$ defined in Eq. 3. Taking the derivatives of Eq. 16, we find

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} 0 & \mathbf{I}_{3\times3} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{I}_{3\times3} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}\mathbf{\Omega}(\omega) & \frac{1}{2}\mathbf{\Xi}(\mathbf{q}) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{I}_{3\times3} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \tag{19}$$

We assume that $\mathbf{F}$ remains constant during the integration of the covariance matrix. Thus, the propagation equation, accurate to Δt^2, is

$$\mathbf{P}_{k+1} = \mathbf{\Phi}_k \mathbf{P}_k \mathbf{\Phi}_k^T + \mathbf{Q}_k \tag{20}$$

where

$$\mathbf{\Phi}_k \approx \mathbf{I} + \mathbf{F}(\hat{\mathbf{x}}_k(+))\Delta t + \mathbf{F}^2(\hat{\mathbf{x}}_k(+))\frac{\Delta t^2}{2}. \tag{21}$$

Depending on the rate the filter is run, it may only be necessary to integrate to first order in Δt, which could save some processing power.

The primary method of tuning the model for different kites is through the process noise. A table of measured and recommended process noises is shown in Tbl. 1. The measured noise reflects accelerations that we've observed during VTOL. The recommended noise is what has worked well in the model. Obviously, as we become more familiar with the filter performance, the recommended values for process noise may change.

The noise here is given as a continuous spectral density. To convert these values to a discrete noise covariance matrix we integrate Q over the sample time interval. More specifically, $\mathbf{Q}_k = \int_0^{\Delta t} dt' \mathbf{\Phi}(t')\mathbf{Q}\mathbf{\Phi}^T(t')$.

Because we use quaternions to represent rotations, the covariance matrix is necessarily singular. This results in numerical problems during some of the matrix calculations, which cause the filter to diverge. There are elegant solutions to this problem, which involving either using a smaller

Sensor	Measurement	Actual measurement noise	Recommended noise	Units
IMU	a_x	3×10^{-5}	”	$\mathrm{m}^2/\mathrm{s}^4/\mathrm{Hz}$
	a_y	3×10^{-5}	”	$\mathrm{m}^2/\mathrm{s}^4/\mathrm{Hz}$
	a_z	1×10^{-3}	”	$\mathrm{m}^2/\mathrm{s}^4/\mathrm{Hz}$
	p	3×10^{-7}	”	$\mathrm{rad}^2/\mathrm{s}^2/\mathrm{Hz}$
	q	3×10^{-7}	”	$\mathrm{rad}^2/\mathrm{s}^2/\mathrm{Hz}$
	r	3×10^{-7}	”	$\mathrm{rad}^2/\mathrm{s}^2/\mathrm{Hz}$
KLAS	ϕ	1×10^{-7}	1×10^{-2}	$\mathrm{rad}^2/\mathrm{Hz}$
	θ	1×10^{-7}	1×10^{-1}	$\mathrm{rad}^2/\mathrm{Hz}$
GLAS	x	$l_T^2 \cdot 1 \times 10^{-7}$	1×10^{-1}	m^2/Hz
	y	$l_T^2 \cdot 1 \times 10^{-7}$	1×10^{-1}	m^2/Hz
	z	$(\frac{l_T^2}{2} \cdot 1 \times 10^{-7})^2$	1×10^{-2}	m^2/Hz
Ultrasonic	d_{ultra}	-	-	m^2/Hz
Tension	T	-	-	N^2/Hz

TABLE 2. Measurement noise

covariance matrix or a slightly modified representation of the rotations. However, another common method of solving this problem is simply to add more process noise to the quaternions. Future versions of the filter should implement the more rigorous solution.

5. STATE UPDATE DETAILS

The nonlinear equations describing the various measurements are:

$$\mathbf{h}(\mathbf{x}) = \begin{bmatrix} \mathbf{a}^{\mathrm{IMU}} \\ \omega^{\mathrm{IMU}} \\ \mathbf{r} \\ \phi^{\mathrm{KLAS}} \\ \theta^{\mathrm{KLAS}} \\ d^{\mathrm{ultra}} \end{bmatrix} = \begin{bmatrix} \mathbf{C}_n^b(\ddot{\mathbf{r}} - \mathbf{g}) + \omega \times (\omega \times \mathbf{b}^{\mathrm{off}}) + \dot{\omega} \times \mathbf{b}^{\mathrm{off}} + \mathbf{b}^{\mathrm{acc}} \\ \omega + \mathbf{b}^{\mathrm{gyro}} \\ \mathbf{r} \\ \arctan\left(\frac{[\mathbf{C}_n^b \mathbf{r}]_y}{\sqrt{[\mathbf{C}_n^b \mathbf{r}]_x^2 + [\mathbf{C}_n^b \mathbf{r}]_z^2}}\right) \\ \arctan\left(\frac{[\mathbf{C}_n^b \mathbf{r}]_x}{[\mathbf{C}_n^b \mathbf{r}]_z}\right) \\ \frac{x}{1-2q_2^2-2q_3^2} \end{bmatrix}. \tag{22}$$

Here we describe these measurement equations briefly. The acceleration as measured by the IMU consists of four terms: accelerations due to kite movement and gravity, which are transferred from the nominal coordinate system to the body system, centripetal acceleration due to the kite's angular velocity, tangential acceleration due to the kite's angular acceleration, and finally the accelerometer biases. The centripetal and tangential acceleration terms both depend on the offset of the IMU from the center of mass $\mathbf{b}^{\mathrm{off}}$. The angular velocity as measured by the IMU simply consists of the true angular velocity and the gyro biases. Both the ϕ_{KLAS} and θ_{KLAS} measurement equations make use of the fact that the tether vector is simply $-\mathbf{C}_n^b \mathbf{r}$. The arctan formulas then follow directly from the definition of the Euler angles. Finally, the ultrasonic distance sensor measures the distance from the center of mass to the ground, which is given by $x/[\mathbf{C}_n^b]_{xx}$.

From these equations, it is possible to derive the sensitivity matrix $\mathbf{H}$. However, showing all the elements of the sensitivity matrix here would be a bit tedious, so to view these elements, the interested reader is directed to the code listing.

The noise associate with each measurement is shown in Tbl. 2. The noise listed here is the continuous noise spectral density, which may be converted to a discrete noise covariance by $R_k = R/\Delta t$.

There are clearly huge discrepancies between the actual measurement noise and the recommended measurement noise. For the KLAS and GLAS, this discrepancy is due to the nature of the sensor. Though these sensors measure the line angle very accurately, the line angle does not necessarily reflect the true position or orientation of the kite. Because it is difficult to include the tether dynamics in the filter, we simply increase the noise level for this sensor, which effectively weights its input less. Note in particular that we add more measurement noise to θ_{KLAS} than ϕ_{KLAS}. When the line tension is low, the KLAS points downward, which causes particularly bad estimates of orientation. If the initial state of the kite is not well know it may also be necessary to increase the noise for the IMU sensors to prevent the filter from diverging.

6. SIMULINK CODE DESCRIPTION

The top-level diagram of the estimator is shown in Fig. 1. At the top-level, the estimator takes in sensor data from the GS, FPGA, and IMU, and converts this data to what we consider the measurements for the filter. For example, we consider the measurements of the GLAS to be x, y and z, rather than θ_{GLAS} and ψ_{GLAS}, for two reasons. First, x, y, and z correspond to states in the filter, which makes implementation easier. Second, future ground-based sensors (*e.g.* stereo vision or LIDAR) may be able to measure kite distance.

After being converted, the measurements are passed to the filter. At the top-level, there are only two visible aspects of the filter. First, the filter is not started until `startCycle` cycles have passed. This is due to peculiarities during the startup of the xPC computer. It also helps during simulation, which sometimes passes NaNs during the first cycle. Once these other issues are resolved, this may be removed. Second, a pulse generator sends a pulse every `slowUpdateTime` seconds. Though the EKF runs at the same rate as the IMU, it only performs updates from the GS and FPGA measurements every `slowUpdateTime` s. This allows the filter to run faster. Also, many of the FPGA and GS sensors do not provide information as fast as the IMU in the sense that they are limited by tether dynamics, etc. In the current implementation, `slowUpdateTime` must be much greater than `sample_time` or the filter will not work properly. This is because the propagation step does not occur during the slow update.

Finally, at the top-level, the estimator converts its internal state (Eq. 8) to the state used by the controller.

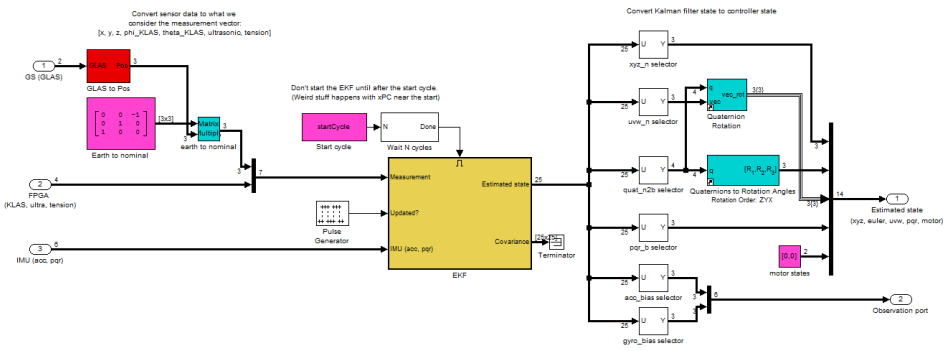

FIGURE 1. Top-level of estimator

The EKF block is shown in Fig. 2. There are three primary subblocks within the EKF block, the prefilter, propagate, and update blocks.

The prefilter block can serve multiple purposes. It can perform "reasonableness" checks on the sensor data. Kalman filters are only optimal filters for Gaussian noise. Other forms of noise, such as large glitches in readings, can seriously degrade filter performance and even cause divergence. If the sensor reading is beyond what is expected (based on, for example, covariance information), the sensor may be switched off for that cycle. The prefilter can also switch off sensors for other reasons. For example, the KLAS and possibly the GLAS should be turned off during low tension situations. Finally, the prefilter can average signals over the slow update cycle.

Currently, the prefilter performs "reasonableness" checks in the form of rate limits on all sensors and it also switches off the GLAS and KLAS sensors during low tension. It is expected that as we become more familiar with Zaftig, further checks, switches, and filtering will occur in this block.

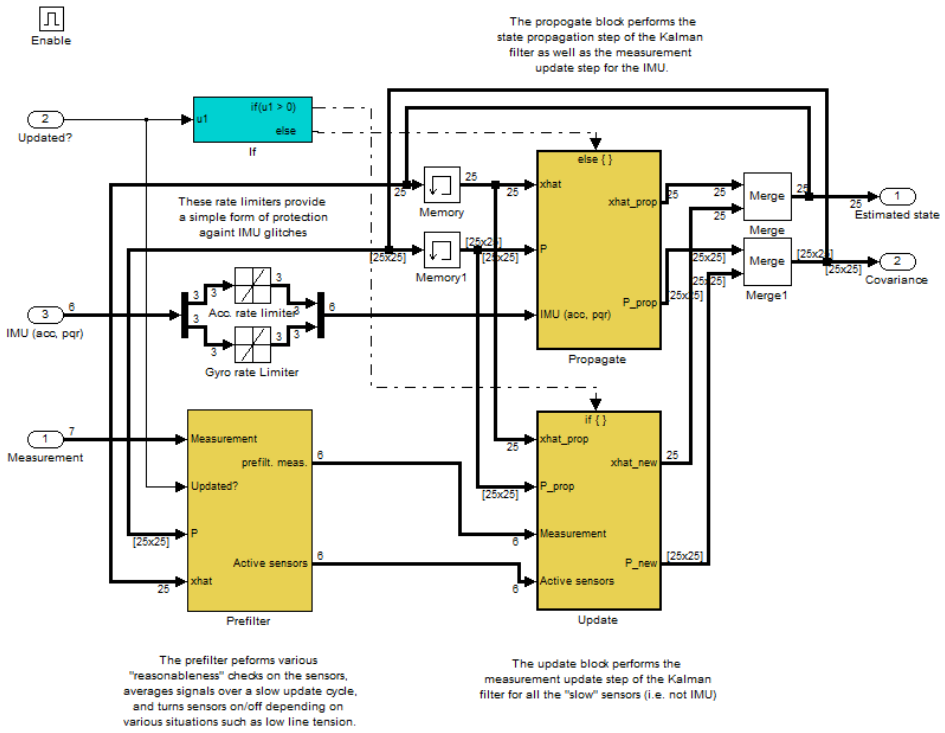

FIGURE 2. Inside the EKF block

The propagate block performs state propagation. However, its name is a bit of a misnomer because it also performs the update step for the IMU data. The update block performs the state update step based on GS and FPGA sensors. Both blocks are entirely implemented as embedded MATLAB code. The propagate block depends on the process noise, `Qvals`, and the IMU measurement noise, `Rvals(1:6)`, and `sample_time`, while the update block is configurable via the FPGA and GS sensors measurement noise `Rvals(7:end)` and `slowUpdateTime`.

We chose to separate the propagate and update blocks in this manner for a few reasons. Because fewer measurements are dealt with in each cycle, the filter does not have to invert a large matrix and thus can run faster. Also, most of the GS/FPGA sensors do not give useful information at the rate the IMU does. Finally, we assume that the IMU will be used in all flight modes, but other flight modes may require different sets of sensors. For example, the pitot tube should not be used during

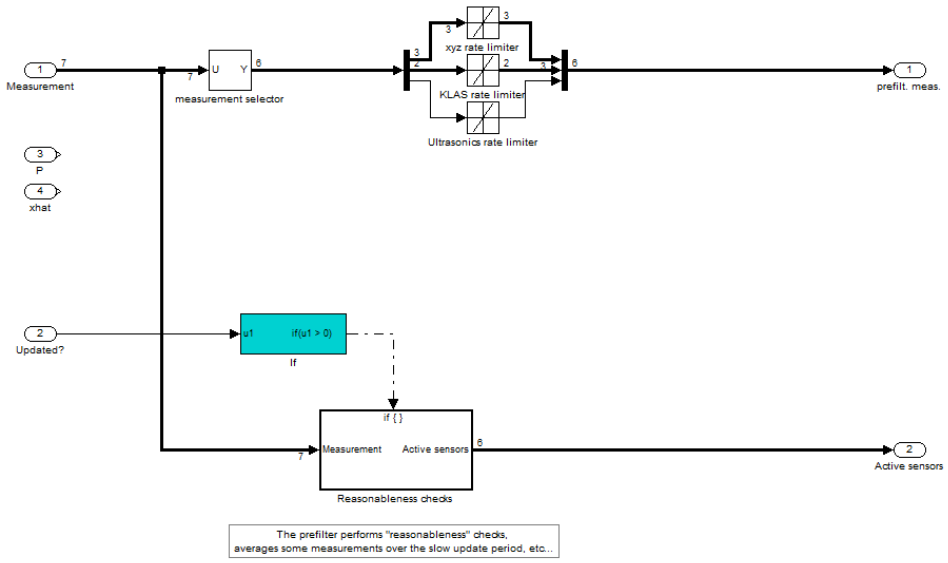

FIGURE 3. Inside the prefilter block

VTOL, while the ultrasonic sensor should not be used during crosswind flight. Separating the update block from the propagate block provides a simple way of switching-out sensors depending on flight mode. It's important to note that the update block should be run at a much slower rate than the propagate block because it uses a computer cycle but does not integrate the state of the system.

As is shown, the outputs of the propagate and update blocks are saved in the memory blocks for the next cycle. The memory blocks, and thus the filter as a whole, are initialized via the `xhat0` and `P0` variables.

7. Potential improvements

There are numerous ways for this estimator to be improved, which may be implemented depending on time and necessity. Here is a short list of potential improvements:

- Switch to multiplicative quaternion updates. This solves the quaternion covariance matrix singularity problem in an elegant way.
- Add states for correlated errors in the GLAS and KLAS measurements to deal with tether dynamics.
- Add dynamics from model.
- Add propagation step to the slow update?
- Perform integration on the IMU DSP.

8. Code

8.1. **State propagate code.**

```
function [xhat_prop, P_prop] = propagate(xhat, P, IMU, Qvals, Rvals, dt, imuOffset)
persistent F Q R H

% N = number of states
```

```
N = 25;

% Initialize F, Q, R, H
if isempty(F)

    % Initialize state transition matrix jacobian, F
    F = zeros(N,N);
    F(1,2) = 1;                 % x
    F(2,3) = 1;                 % ddx
    F(4,5) = 1;                 % y
    F(5,6) = 1;                 % ddy
    F(7,8) = 1;                 % z
    F(8,9) = 1;                 % ddz
    F(14,15) = 1;               % p
    F(16,17) = 1;               % q
    F(18,19) = 1;               % r


    % Initialize process noise covariance matrix (multiplied by dt), Q
    Q = zeros(N,N);

    % acceleration noise
    Sxx = Qvals(1);
    Syy = Qvals(2);
    Szz = Qvals(3);

    % angular acceleration noise
    Sdpdp = Qvals(4);
    Sdqdq = Qvals(5);
    Sdrdr = Qvals(6);

    % accelerometer bias noise
    Sbddxbddx = Qvals(7);
    Sbddybddy = Qvals(8);
    Sbddzbddz = Qvals(9);

    % rate gyro bias noise
    Sbpbp = Qvals(10);
    Sbqbq = Qvals(11);
    Sbrbr = Qvals(12);

    % accelerometer scale factor noise
    %Sasfx = Qvals(13);
    %Sasfy = Qvals(14);
    %Sasfz = Qvals(15);

    % rate gyro scale factor noise
    %Sgsfx = Qvals(16);
    %Sgsfy = Qvals(17);
```

```
    %Sgsfz = Qvals(18);

    % Process noise covariance for acceleration
    % (see Applied Optimal Estimation by A. Gelb, pg. 149)
    Q3x3 = [[dt^5/20,    dt^4/8,     dt^3/6];
            [dt^4/8,     dt^3/3,     dt^2/2];
            [dt^3/6,     dt^2/2,     dt]];

    Q(20,20) = Sbddxbddx*dt;
    Q(21,21) = Sbddybddy*dt;
    Q(22,22) = Sbddzbddz*dt;
    Q(23,23) = Sbpbp*dt;
    Q(24,24) = Sbqbq*dt;
    Q(25,25) = Sbrbr*dt;
    %Q(26,26) = Sasfx*dt;
    %Q(27,27) = Sasfy*dt;
    %Q(28,28) = Sasfz*dt;
    %Q(29,29) = Sgsfx*dt;
    %Q(30,30) = Sgsfy*dt;
    %Q(31,31) = Sgsfz*dt;

    Q(1:3,1:3) = Sxx*Q3x3;
    Q(4:6,4:6) = Syy*Q3x3;
    Q(7:9,7:9) = Szz*Q3x3;

    % Process noise covariance for angular velocity
    % (see Applied Optimal Estimation by A. Gelb, pg. 149)
    Q2x2 = [[dt^3/3, dt^2/2];
            [dt^2/2, dt]];

    Q(14:15,14:15) = Sdpdp*Q2x2;
    Q(16:17,16:17) = Sdqdq*Q2x2;
    Q(18:19,18:19) = Sdrdr*Q2x2;

    % Initialize measurement noise covariance matrix, R
    R = zeros(6,6);
    R(1,1) = Rvals(1)/dt;        % a_IMU_x
    R(2,2) = Rvals(2)/dt;        % a_IMU_y
    R(3,3) = Rvals(3)/dt;        % a_IMU_z
    R(4,4) = Rvals(4)/dt;        % p
    R(5,5) = Rvals(5)/dt;        % q
    R(6,6) = Rvals(6)/dt;        % r

    % Initialize sensitivity matrix, H
    H = zeros(6,N);
end;

% quaternion (nominal to body)
q0 = xhat(10);
```

```
q1 = xhat(11);
q2 = xhat(12);
q3 = xhat(13);

% angular velocity and acceleration
p = xhat(14);
dp = xhat(15);
q = xhat(16);
dq = xhat(17);
r = xhat(18);
dr = xhat(19);

F(10,11) = -p/2;
F(10,12) = -q/2;
F(10,13) = -r/2;
F(11,10) = p/2;
F(11,12) = r/2;
F(11,13) = -q/2;
F(12,10) = q/2;
F(12,11) = -r/2;
F(12,13) = p/2;
F(13,10) = r/2;
F(13,11) = q/2;
F(13,12) = -p/2;

F(10,14) = -q1/2;
F(10,16) = -q2/2;
F(10,18) = -q3/2;
F(11,14) = q0/2;
F(11,16) = -q3/2;
F(11,18) = q2/2;
F(12,14) = q3/2;
F(12,16) = q0/2;
F(12,18) = -q1/2;
F(13,14) = -q2/2;
F(13,16) = q1/2;
F(13,18) = q0/2;



Sdpdp = Qvals(4);
Sdqdq = Qvals(5);
Sdrdr = Qvals(6);
Ap = [-q1; q0; q3; -q2];
Aq = [-q2; -q3; q0; q1];
Ar = [-q3; q2; -q1; q0];
Qquat = dt^5/80*(Sdpdp*Ap*Ap' + Sdqdq*Aq*Aq' + Sdrdr*Ar*Ar');
Qquatpqr = [Sdpdp*Ap*[dt^4/16 dt^2/12], Sdqdq*Aq*[dt^4/16 dt^2/12], Sdrdr*Ar*[dt^4/16 dt^2/12]];
```

```
Q(10:13, 10:13) = Qquat;
Q(10:13, 14:19) = Qquatpqr;
Q(14:19, 10:13) = Qquatpqr';


% to keep P well-conditioned (should use MEKF)
Sq0q0 = 1e-3;%1e-2;
Sq1q1 = 1e-1;%1;
Sq2q2 = 1e-1;%1;
Sq3q3 = 1e-1;%1;

Q(10,10) = Sq0q0*dt;
Q(11,11) = Sq1q1*dt;
Q(12,12) = Sq2q2*dt;
Q(13,13) = Sq3q3*dt;


%%%
% Propagate estimate
%
xhat_p = zeros(25,1);

xhat_p(1) = xhat(1) + xhat(2)*dt + xhat(3)*dt*dt/2;
xhat_p(2) = xhat(2) + xhat(3)*dt;
xhat_p(3) = xhat(3);

xhat_p(4) = xhat(4) + xhat(5)*dt + xhat(3)*dt*dt/2;
xhat_p(5) = xhat(5) + xhat(6)*dt;
xhat_p(6) = xhat(6);

xhat_p(7) = xhat(7) + xhat(8)*dt + xhat(9)*dt*dt/2;
xhat_p(8) = xhat(8) + xhat(9)*dt;
xhat_p(9) = xhat(9);

dTheta = [p*dt+dp*dt*dt/2, q*dt+dq*dt*dt/2, r*dt+dr*dt*dt/2];
dThetaMag = 1 - (dTheta(1)*dTheta(1)+dTheta(2)*dTheta(2)+dTheta(3)*dTheta(3))/8;
xhat_p(10) = dThetaMag*xhat(10) + (-dTheta(1)*q1-dTheta(2)*q2-dTheta(3)*q3)/2;
xhat_p(11) = dThetaMag*xhat(11) + (dTheta(1)*q0-dTheta(2)*q3+dTheta(3)*q2)/2;
xhat_p(12) = dThetaMag*xhat(12) + (dTheta(1)*q3+dTheta(2)*q0-dTheta(3)*q1)/2;
xhat_p(13) = dThetaMag*xhat(13) + (-dTheta(1)*q2+dTheta(2)*q1+dTheta(3)*q0)/2;

xhat_p(14) = xhat(14) + xhat(15)*dt;
xhat_p(15) = xhat(15);
xhat_p(16) = xhat(16) + xhat(17)*dt;
xhat_p(17) = xhat(17);
xhat_p(18) = xhat(18) + xhat(19)*dt;
xhat_p(19) = xhat(19);

xhat_p(20) = xhat(20);
```

```
xhat_p(21) = xhat(21);
xhat_p(22) = xhat(22);

xhat_p(23) = xhat(23);
xhat_p(24) = xhat(24);
xhat_p(25) = xhat(25);


%%%
% Propagate covariance
%
Phi = eye(25)+F*dt+F*F*dt*dt/2;
P_p = Phi*P*Phi' + Q;
%FP = F*P;
%P_p = P + FP + FP' + Q;%P + F*P + P*F' + Q;


%%%
% Compute observation estimates
%
ddx = xhat_p(3)+9.81;
ddy = xhat_p(6);
ddz = xhat_p(9);
q0 = xhat_p(10);
q1 = xhat_p(11);
q2 = xhat_p(12);
q3 = xhat_p(13);
p = xhat_p(14);
q = xhat_p(16);
r = xhat_p(18);
acc_n = xhat_p([3,6,9]);
omega = xhat_p([14 16 18]);
domega = xhat_p([15 17 19]);
acc_bias = xhat_p([20 21 22]);
gyro_bias = xhat_p([23 24 25]);
sfx = 1;%xhat_p(26);
sfy = 1;%xhat_p(27);
sfz = 1;%xhat_p(28);
gsfx = 1;%xhat_p(29);
gsfy = 1;%xhat_p(30);
gsfz = 1;%xhat_p(31);

C_n2b = [1-2*q2*q2-2*q3*q3, 2*(q0*q3+q1*q2), 2*(q1*q3-q0*q2);
         2*(q1*q2-q0*q3), 1-2*q1*q1-2*q3*q3, 2*(q0*q1+q2*q3);
         2*(q0*q2+q1*q3), 2*(q2*q3-q0*q1), 1-2*q1*q1-2*q2*q2];

yhat = zeros(6,1);
yhat0 = C_n2b*(acc_n+[9.81;0;0]) ...
    + cross(omega,cross(omega,imuOffset'))' ...
```

```
    + cross(domega,imuOffset')';
yhat([1,2,3]) = [sfx;sfy;sfz].*yhat0 ...
    + acc_bias;
yhat([4,5,6]) = omega + gyro_bias; %[gsfx;gsfy;gsfz].*omega + gyro_bias;


%%%
% calcH
%

% IMU offset from CG
b1 = imuOffset(1);
b2 = imuOffset(2);
b3 = imuOffset(3);

% IMU acceleration measurement
H(1,3) = sfx*C_n2b(1,1);                       % d(a_IMU_x)/d(a_e_x)
H(1,6) = sfx*C_n2b(1,2);                       % d(a_IMU_x)/d(a_e_y)
H(1,9) = sfx*C_n2b(1,3);                       % d(a_IMU_x)/d(a_e_z)
H(1,10) = sfx*(2*q3*ddy-2*q2*ddz);             % d(a_IMU_x)/d(q0)
H(1,11) = sfx*(2*q2*ddy+2*q3*ddz);             % d(a_IMU_x)/d(q1)
H(1,12) = sfx*(-4*q2*ddx+2*q1*ddy-2*q0*ddz);   % d(a_IMU_x)/d(q2)
H(1,13) = sfx*(-4*q3*ddx+2*q0*ddy+2*q1*ddz);   % d(a_IMU_x)/d(q3)
H(1,14) = sfx*(q*b2+r*b3);                             % d(a_IMU_x)/d(p)
H(1,15) = sfx*0;                                     % d(a_IMU_x)/d(dp)
H(1,16) = sfx*(p*b2-2*q*b1);                           % d(a_IMU_x)/d(q)
H(1,17) = sfx*b3;                                    % d(a_IMU_x)/d(dq)
H(1,18) = sfx*(-2*r*b1+p*b3);                          % d(a_IMU_x)/d(r)
H(1,19) = sfx*-b2;                                   % d(a_IMU_x)/d(dr)
H(1,20) = 1;                              % d(a_IMU_x)/d(bddx)
%H(1,26) = yhat0(1);

H(2,3) = sfy*C_n2b(2,1);                      % d(a_IMU_y)/d(a_e_x)
H(2,6) = sfy*C_n2b(2,2);                      % d(a_IMU_y)/d(a_e_y)
H(2,9) = sfy*C_n2b(2,3);                      % d(a_IMU_y)/d(a_e_z)
H(2,10) = sfy*(-2*q3*ddx+2*q1*ddz);           % d(a_IMU_y)/d(q0)
H(2,11) = sfy*(2*q2*ddx-4*q1*ddy+2*q0*ddz);   % d(a_IMU_y)/d(q1)
H(2,12) = sfy*(2*q1*ddx+2*q3*ddz);            % d(a_IMU_y)/d(q2)
H(2,13) = sfy*(-2*q0*ddx-4*q3*ddy+2*q2*ddz);  % d(a_IMU_y)/d(q3)
H(2,14) = sfy*(-2*p*b2+q*b1);                          % d(a_IMU_y)/d(p)
H(2,15) = sfy*-b3;                                   % d(a_IMU_y)/d(dp)
H(2,16) = sfy*(r*b3+p*b1);                             % d(a_IMU_y)/d(q)
H(2,17) = sfy*0;                                     % d(a_IMU_y)/d(dq)
H(2,18) = sfy*(q*b3-2*r*b2);                           % d(a_IMU_y)/d(r)
H(2,19) = sfy*b1;                                    % d(a_IMU_y)/d(dr)
H(2,21) = 1;                              % d(a_IMU_y)/d(bddy)
%H(2,27) = yhat0(2);

H(3,3) = sfz*C_n2b(3,1);                      % d(a_IMU_z)/d(a_e_x)
```

```
H(3,6) = sfz*C_n2b(3,2);                          % d(a_IMU_z)/d(a_e_y)
H(3,9) = sfz*C_n2b(3,3);                          % d(a_IMU_z)/d(a_e_z)
H(3,10) = sfz*(2*q2*ddx-2*q1*ddy);                      % d(a_IMU_z)/d(q0)
H(3,11) = sfz*(2*q3*ddx-2*q0*ddy-4*q1*ddz);       % d(a_IMU_z)/d(q1)
H(3,12) = sfz*(2*q0*ddx+2*q3*ddy-4*q2*ddz);       % d(a_IMU_z)/d(q2)
H(3,13) = sfz*(2*q1*ddx+2*q2*ddy);                      % d(a_IMU_z)/d(q3)
H(3,14) = sfz*(r*b1-2*p*b3);                            % d(a_IMU_z)/d(p)
H(3,15) = sfz*b2;                                     % d(a_IMU_z)/d(dp)
H(3,16) = sfz*(-2*q*b3+r*b2);                           % d(a_IMU_z)/d(q)
H(3,17) = sfz*-b1;                                    % d(a_IMU_z)/d(dq)
H(3,18) = sfz*(p*b1+q*b2);                              % d(a_IMU_z)/d(r)
H(3,19) = sfz*0;                                      % d(a_IMU_z)/d(dr)
H(3,22) = 1;                                % d(a_IMU_z)/d(bddz)
%H(3,28) = yhat0(3);

% pqr measurement
H(4,14) = 1;%gsfx*1;  % dp/dp
H(4,23) = 1;
%H(4,29) = p;
H(5,16) = 1;  % dq/dq
H(5,24) = 1;
%H(5,30) = q;
H(6,18) = 1;  % dr/dr
H(6,25) = 1;
%H(6,31) = r;


%%%
% Compute residual
%
residual = IMU - yhat;

%%%
% Compute Kalman gain
%
P_pH = P_p*H';
K = P_pH/(H*P_pH+R);

%%%
% Update estimate
%
xhat_prop = K*residual + xhat_p;
xhat_prop(10:13) = xhat_prop(10:13)/norm(xhat_prop(10:13));

%%%
% Update covariance
%
P_prop = (eye(N)-K*H)*P_p;
```

8.2. **State update code.**

```
function [xhat_new, P_new] = update(xhat_prop, P_prop, meas, activeSensors, dt, Rvals)
persistent H

% N = number of states
N = 25;

% Initialize sensitivity matrix, H
if isempty(H)
    H = zeros(6,N);
end;

% Set measurement noise of inactive sensors to large value
% (avoid using really large values for matrix computation reasons)
for k=1:length(activeSensors)
    if ~activeSensors(k)
        Rvals(k) = 1e2;
    end;
end;

% Initialize measurement noise covariance matrix, R
R = diag(Rvals)/dt;



%%%
% Compute observation estimates
%

% position in nominal frame
x = xhat_prop(1);
y = xhat_prop(4);
z = xhat_prop(7);
xyz_n = xhat_prop([1,4,7]);

% quaternion (nominal to body)
q0 = xhat_prop(10);
q1 = xhat_prop(11);
q2 = xhat_prop(12);
q3 = xhat_prop(13);

% DCM (nominal to body)
C_n2b = [1-2*q2^2-2*q3^2, 2*(q0*q3+q1*q2), 2*(q1*q3-q0*q2);
         2*(q1*q2-q0*q3), 1-2*q1^2-2*q3^2, 2*(q0*q1+q2*q3);
         2*(q0*q2+q1*q3), 2*(q2*q3-q0*q1), 1-2*q1^2-2*q2^2];
xyz_b = C_n2b*xyz_n;


% estimated observation
```

```
yhat = zeros(6,1);

% position (from GLAS)
yhat([1,2,3]) = xyz_n;

% KLAS (phi)
yhat(4) = atan2(xyz_b(2), sqrt(xyz_b(1)*xyz_b(1)+xyz_b(3)*xyz_b(3)));

% KLAS (theta)
yhat(5) = atan2(-xyz_b(1), -xyz_b(3));

% Ultrasonic distance
yhat(6) = xyz_n(1)/C_n2b(1,1);


%%%
% Compute H
%

% coordinates of GS in body frame (i.e. this is the tether vector)
xt = -xyz_b(1);
yt = -xyz_b(2);
zt = -xyz_b(3);

% position measurement
H(1,1) = 1;   % dx/dx
H(2,4) = 1;   % dy/dy
H(3,7) = 1;   % dz/dz

drtdx = -C_n2b;
drtdq = [-2*q3*y+2*q2*z, -2*q2*y-2*q3*z, 4*q2*x-2*q1*y+2*q0*z, 4*q3*x-2*q0*y-2*q1*z;
    2*q3*x-2*q1*z, -2*q2*x+4*q1*y-2*q0*z, -2*q1*x-2*q3*z, 2*q0*x+4*q3*y-2*q2*z;
    -2*q2*x+2*q1*y, -2*q3*x+2*q0*y+4*q1*z, -2*q0*x-2*q3*y+4*q2*z, -2*q1*x-2*q2*y];

% phi_K measurement
f = -yt;
g = sqrt(xt^2+zt^2);
datan = 1/(1+(f/g)^2);

h = datan/g^2*[xt*yt/g, -g, yt*zt/g];

H(4,1) = h*drtdx(:,1);    % d(phi)/d(x)
H(4,4) = h*drtdx(:,2);    % d(phi)/d(y)
H(4,7) = h*drtdx(:,3);    % d(phi)/d(z)
H(4,10) = h*drtdq(:,1);   % d(phi)/d(q0)
H(4,11) = h*drtdq(:,2);   % d(phi)/d(q1)
H(4,12) = h*drtdq(:,3);   % d(phi)/d(q2)
H(4,13) = h*drtdq(:,4);   % d(phi)/d(q3)
```

```
% theta_K measurement
f2 = xt;
g2 = zt;
datan2 = 1/(1+(f2/g2)^2);

h2 = datan2/g2^2*[g2, 0, -f2];

H(5,1) = h2*drtdx(:,1);    % d(theta)/d(x)
H(5,4) = h2*drtdx(:,2);    % d(theta)/d(y)
H(5,7) = h2*drtdx(:,3);    % d(theta)/d(z)
H(5,10) = h2*drtdq(:,1);   % d(theta)/d(q0)
H(5,11) = h2*drtdq(:,2);   % d(theta)/d(q1)
H(5,12) = h2*drtdq(:,3);   % d(theta)/d(q2)
H(5,13) = h2*drtdq(:,4);   % d(theta)/d(q3)

% ultrasound distance
H(6,1) = 1/C_n2b(1,1);                  % d(d_ultra)/d(x)
H(6,12) = 4*x*q2/C_n2b(1,1)^2;          % d(d_ultra)/d(q2)
H(6,13) = 4*x*q3/C_n2b(1,1)^2;          % d(d_ultra)/d(q3)

% yaw measurement on test stand
%f3 = -2*(q1*q2-q0*q3);
%g3 = q0*q0-q1*q1+q2*q2-q3*q3;
%datan3 = 1/(1+(f3/g3)^2);
%H(12,10) = datan3*(g3*2*q3-f3*2*q0)/g3^2;
%H(12,11) = datan3*(-g3*2*q2+f3*2*q1)/g3^2;
%H(12,12) = datan3*(-g3*2*q1-f3*2*q2)/g3^2;
%H(12,13) = datan3*(g3*2*q0+f3*2*q3)/g3^2;


% set measurement for inactive sensors to the estimated observation
for k=1:length(activeSensors)
    if ~activeSensors(k)
        meas(k) = yhat(k);
    end;
end;

%%%
% Compute residual
%
residual = meas - yhat;

%%%
% Compute Kalman gain
%
K = P_prop*H'/(H*P_prop*H'+R);

%%%
```

```
% Update estimate
%
xhat_new = K*residual + xhat_prop;

% normalize quaternion (necessary?)
xhat_new(10:13) = xhat_new(10:13)/norm(xhat_new(10:13));

%%%
% Update covariance
%
P_new = (eye(N)-K*H)*P_prop;

% symmetrize P (neccessary?)
P_new = 0.5*(P_new+P_new');
```