Autonomous Vehicle Simulation (AVS) Laboratory, University of Colorado

Basilisk Technical Memorandum

Document ID: Basilisk-test_unitThrusterDynamics

SUNLINE EKF MODULE AND TEST

Prepared by	T. Teil

Status: Initial document
Scope/Contents
This module implements and tests a Extended Kalman Filter in order to estimate the sunline direction.

Rev:	**Change Description**	**By**
Draft	Initial Revision	T. Teil

Contents

1 Introduction **1**

2 Filter Set-up, initialization, and I/O **1**
2.1 Dynamics 1
2.2 User initialization 2
2.3 Inputs and Outputs 3

3 Filter Algorithm **3**

4 Test Design **5**
4.1 `sunline_individual_test` 5
4.2 `StatePropStatic` 5
4.3 `StatePropVariable` 7
4.4 `Full Filter test` 7

---

1 Introduction

The Extended Kalman filter (EKF) in the AVS Basilisk simulation is a sequential filter implemented to give the best estimate of the desired states. In this method we estimate the sun heading as well as it's rate of change. The EKF reads in the message written by the coarse sun sensor, and writes a message containing the sun estimate.

This document summarizes the content of the module, how to use it, and the test that was implemented for it.

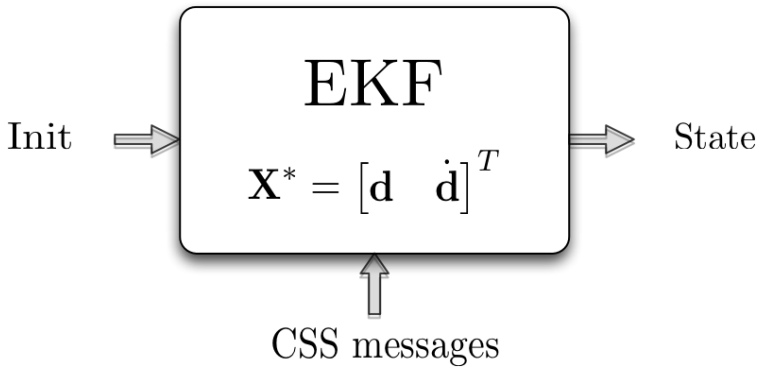

Fig. 1: Filter module

2 Filter Set-up, initialization, and I/O

2.1 Dynamics

The states that are estimated in this filter are the sunline vector, and it's rate of change $\boldsymbol{X}^* = \begin{bmatrix} \boldsymbol{d} \end{bmatrix}^T$. The star superscript represents that this is the reference state.

The dynamics are given in equation 1. Given the nature of the filter, there is an unobservable state component: the rotation about the $\boldsymbol{d}$ axis. In order to remedy this, we project the states along this axis and subtract them, in order to measure only observable state components.

$$\boldsymbol{F}(\boldsymbol{X}) = \dot{\boldsymbol{d}} = -\boldsymbol{\omega} \times \boldsymbol{d} \tag{1}$$

This leads us to the computation of the dynamics matrix $A = \left[\frac{\partial \boldsymbol{F}(\boldsymbol{X},t_i)}{\partial \boldsymbol{X}}\right]^*$. The partials are given in equation 2, and were verified in Mathematica.

$$A = \left[\frac{\partial \boldsymbol{F}(\boldsymbol{d},t_i)}{\partial \boldsymbol{d}}\right] \tag{2}$$

$$= -\left[\tilde{\boldsymbol{\omega}}\right] \tag{3}$$

If rate gyro measurements are available, we can use the $\boldsymbol{\omega}$ vector that they provide. In the case where they are not available, we can approximate it by logging an extra time step of the sun heading vector estimate $\boldsymbol{d}$:

$$\boldsymbol{\omega}_k = \frac{1}{\Delta t}\frac{\boldsymbol{d}_k \times \boldsymbol{d}_{k-1}}{\|\boldsymbol{d}_k \times \boldsymbol{d}_{k-1}\|}\arccos\left(\frac{\boldsymbol{d}_k \cdot \boldsymbol{d}_{k-1}}{\|\boldsymbol{d}_k\|\|\boldsymbol{d}_{k-1}\|}\right) \tag{4}$$

The measurement model is given in equation 5, and the H matrix defined as $H = \left[\frac{\partial \boldsymbol{G}(\boldsymbol{X},t_i)}{\partial \boldsymbol{X}}\right]^*$ is given in equation 6.

In this filter, the only measurements used are from the coarse sun sensor. For the i^{th} sensor, the measurement is simply given by the dot product of the sunline heading and the normal to the sensor. This yields easy partial derivatives for the H matrix, which is a matrix formed of the rows of transposed normal vectors (only for those which received a measurement). Hence the H matrix has a changing size depending on the amount of measurements.

$$\boldsymbol{G}_i(\boldsymbol{X}) = \boldsymbol{n}_i \cdot \boldsymbol{d} \tag{5}$$

$$\boldsymbol{H}(\boldsymbol{X}) = \begin{bmatrix} \boldsymbol{n}_1^T \\ \vdots \\ \boldsymbol{n}_i^T \end{bmatrix} \tag{6}$$

2.2 User initialization

In ordre for the filter to run, the user must set a few parameters:

- The angle threshold under which the coarse sun sensors do not read the measurement: `FilterContainer.sensorU = 0.`
- The process noise value, for instance `FilterContainer.qProcVal = 0.001`
- The measurement noise value, for instance `FilterContainer.qObsVal = 0.001`
- The threshold in the covariance norm leading to the switch from the EKF update to the linear Kalman Filter update (discussed more closely in the Measurement update part): `FilterContainer.ekfSwitch = 5`

- The initial covariance: `Filter.covar = [0.4, 0., 0., 0., 0., 0., 0., 0.4, 0., 0., 0., 0., 0., 0., 0.4, 0., 0., 0., 0., 0., 0., 0.004, 0., 0., 0., 0., 0., 0., 0.004, 0., 0., 0., 0., 0., 0., 0.004]`
- The initial state : `Filter.state = [1., 1., 1., 0., 0., 0.]`
- The initial state error can not be initialized and therefore set to zero. If the true trajectory is known though, it can be set as such: `Filter.state = [1., 0., 1., 0., 0.1, 0.]`

The messages must also be set as such:

- `filterObject.navStateOutMsgName = "sunline_state_estimate"`
- `filterObject.filtDataOutMsgName = "sunline_filter_data"`
- `filterObject.cssDataInMsgName = "css_sensors_data"`
- `filterObject.cssConfInMsgName = "css_config_data"`

2.3 Inputs and Outputs

The EKF reads in the measurements from the coarse sun sensors. These are under the form of a list of cosine values. Knowing the normals to each of the sensors, we can therefore use them to estimate sun heading.

3 Filter Algorithm

Once the filter has been properly setup in the python code, it can go through it's algorithm:

Initialization

First the filter is initialized. This can be done at any time during a simulation in order to reset the filter.

- Time is set to t_0
- The state $\boldsymbol{X}^*$ is set to the initial state $\boldsymbol{X}_0^*$
- The state error $\boldsymbol{x}$ is set to it's initial value $\boldsymbol{x}_0$
- The covariance P is set to the initial state P_0

Time Update

At some time t_i, if the update filter method is called, a time update will first be executed.

- The state is propagated using the dynamics $\boldsymbol{F}$ with initial conditions $\boldsymbol{X}^*(t_{i-1})$
- Compute the dynamics matrix $A(t) = \left[\frac{\partial \boldsymbol{F}(\boldsymbol{X},t)}{\partial \boldsymbol{X}}\right]^*$ which is evaluated on the reference trajectory
- Integrate the STM, $\dot{\Phi}(t, t_{i-1}) = A(t)\Phi(t, t_{i-1})$ with initial conditions $\Phi(t_{i-1}, t_{i-1}) = I$

This gives us $\boldsymbol{X}^*(t_i)$ and $\Phi(t_i, t_{i-1})$.

Observation read in

If no measurement is read in at time t_i:

- $\boldsymbol{X}^*(t_i)$ previously computed becomes the most recent reference state
- $\boldsymbol{x}_i = \bar{x}_i = \Phi(t_i, t_{i-1})x_{i-1}$ is the new state error
- $P_i = \bar{P}_i = \Phi(t_i, t_{i-1})P_{i-1}\Phi^T(t_i, t_{i-1})$ becomes the updated covariance

If a measurement is read in, the algorithm computes the observation, the observation state matrix, and the Kalman Gain.

- The observation ($\boldsymbol{Y}_i$) is compared to the observation model, giving the innovation: $\boldsymbol{y}_i = \boldsymbol{Y}_i - G(\boldsymbol{X}_i^*, t_i)$
- Compute the observation matrix along the reference trajectory: $\tilde{H}_i = \left[\frac{\partial \boldsymbol{G}(\boldsymbol{X}, t_i)}{\partial \boldsymbol{X}}\right]^*$
- Compute the Kalman Gain $K_i = \bar{P}_i \tilde{H}_i{}^T \left(\tilde{H}_i \bar{P}_i \tilde{H}_i{}^T + R_i\right)^{-1}$

Measurement Update

Depending on the covariance, the filter can either update as a classic, linear Kalman Filter, or as the Extended Kalman filter. This is done in order to assure robust and fast filter convergence. Indeed in a scenario with a very large initial covariance, the EKF's change in reference trajectory could delay or inhibit the convergence. In order to remedy this, a few linear updates are performed if the maximum value in the covariance is greater than a user-set threshold.

Linear update:

- The state error is updated using the time updated value: $\boldsymbol{x}_i = \bar{x}_i + K_i \left[\boldsymbol{y}_i - \tilde{H}_i \bar{x}_i\right]$
- The covariance is updated using the Joseph form of the covariance update equation: $P_i = \left(I - K_i \tilde{H}_i\right) \bar{P}_i \left(I - K_i \tilde{H}_i\right)^T + K_i R_i K_i^T$
- The reference state stays the same, and it's propagated value $\boldsymbol{X}^*(t_i)$ becomes $\boldsymbol{X}^*(t_{i-1})$

EKF update:

- The state error is updated using the innovation and the Kalman Gain: $\boldsymbol{x}_i = K_i \boldsymbol{y}_i$
- The reference state is changed by the state error: $\boldsymbol{X}^*(t_i) = \boldsymbol{X}^*(t_i) + \boldsymbol{x}_i$
- The covariance is updated using the Joseph form of the covariance update equation: $P_i = \left(I - K_i \tilde{H}_i\right) \bar{P}_i \left(I - K_i \tilde{H}_i\right)^T + K_i R_i K_i^T$
- The new reference state is now used $\boldsymbol{X}^*(t_i)$ becomes $\boldsymbol{X}^*(t_{i-1})$

4 Test Design

The unit test for the sunlineEKF module is located in:

`FswAlgorithms/attDetermination/sunlineEKF/_UnitTest/test_SunlineEKF.py`

As well as another python file containing plotting functions:
`FswAlgorithms/attDetermination/sunlineEKF/_UnitTest/SunlineEKF_test_utilities.py`

The test is split up into 4 subtests, the last one is parametrized in order to test different scenarios. The first test creaks up all of the individual filter methods and tests them individually. The second test verifies that in the case where the state is zeroed out from the start of the simulation, it remains at zero. The third test verifies the behavior of the time update in a general case. The final test is a full filter test.

4.1 `sunline_individual_test`

In each of these individual tests, random inputs are fed to the methods and their values are computed in parallel in python. These two values are then compared to assure that the correct computations are taking place.

- Dynamics Matrix: This method computes the dynamics matrix A. Tolerance to error $\epsilon = 10^{-10}$.
  Passed
- State and STM propagation: This method propagates the state using the $\boldsymbol{F}$ function as well as the STM using $\dot{\Phi} = A\Phi$. Tolerance to error $\epsilon = 10^{-10}$.
  Passed
- H and y propagation: This method computes the H matrix, and compares the measurements to the expected measurements given the state. Tolerance to error $\epsilon = 10^{-10}$.
  Passed
- Kalman gain: This method computes the K matrix. Tolerance to error $\epsilon = 10^{-10}$.
  Passed
- EKF update: This method performs the measurement update in the case of an EKF. Tolerance to error $\epsilon = 10^{-10}$.
  Passed
- Linear Update: This method performs the measurement update in the linear case. Tolerance to error $\epsilon = 10^{-10}$.
  Passed

4.2 `StatePropStatic`

This test runs the filter with no measurements. It initializes with a zeroed state, and assures that at the end of the simulation all values are still at zero. Plotted results are seen in Figure 2.

Tolerance to error: $\epsilon = 10^{-10}$
Passed

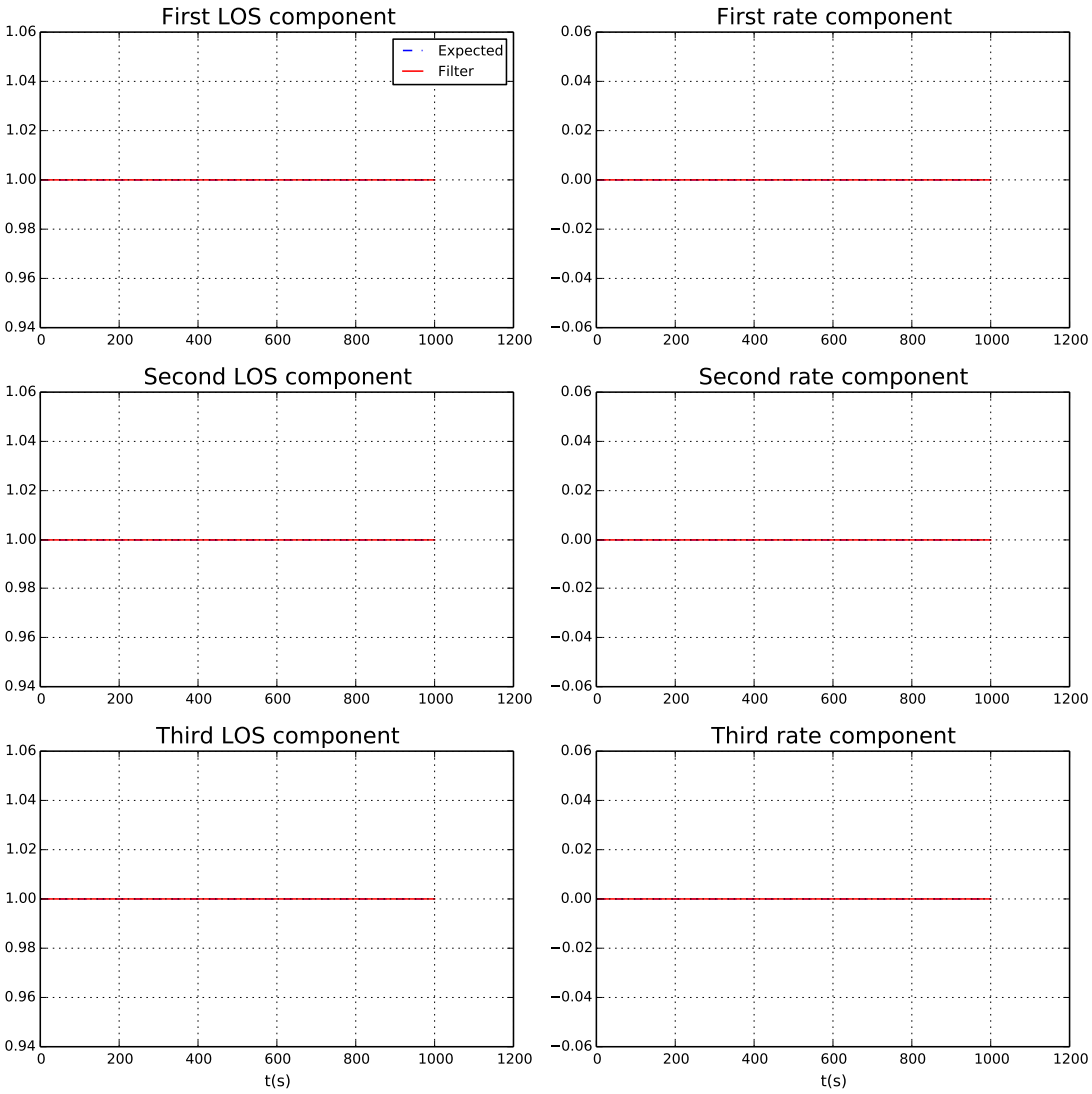

Fig. 2: States vs true states in static case

4.3 `StatePropVariable`

This test also takes no measurements in, but gives a random state with rate of change. It then tests that the states and covariance are as expected throughout the time of simulation. Plotted results are seen in Figure 3. We indeed see that the state and covariance for the test and the code overlap perfectly.

Tolerance to error: $\epsilon = 10^{-10}$

Passed

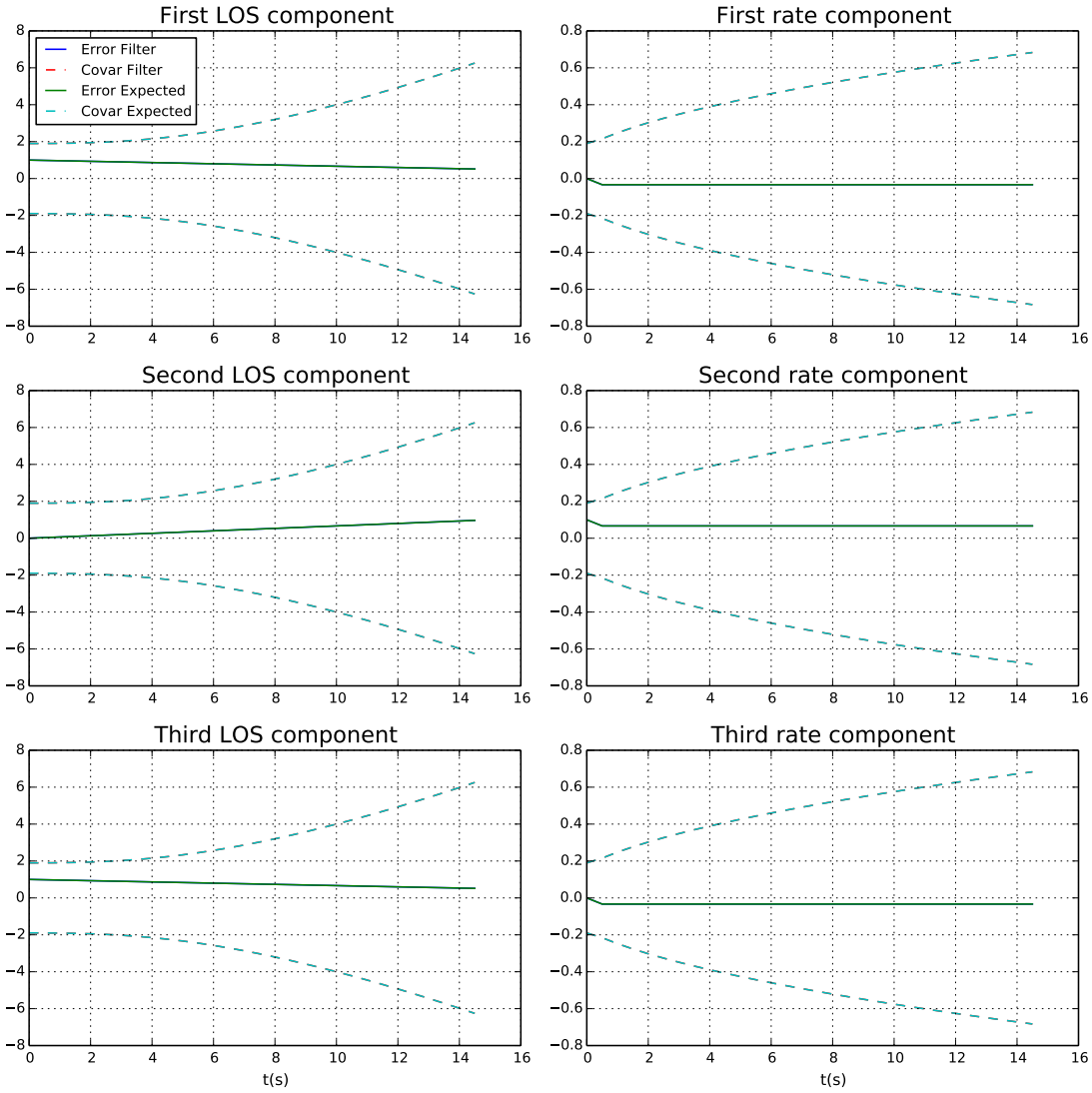

Fig. 3: State error and covariance vs expected Values

4.4 `Full Filter test`

This test the filter working from start to finish. No measurements are taken in for the first 20 time steps. Then a heading is given through the CSS message. Halfway through the simulation, measurements stop, and 20 time steps later a different heading is read. The filter must be robust and detect this change. This test is parametrized for different test lengths, different initial conditions, different measured headings, and with or without measurement noise. All these are successful.

Tolerance to error without measurement noise: $\epsilon = 10^{-10}$
Tolerance to error with measurement noise: $\epsilon = 10^{-2}$
Passed

Plotted results are seen in Figures 4, 5, and 6. Figure 4 shows the state error and covariance over the run. We see the covariance initially grow, then come down quickly as measurements are used. It grows once again as the measurements stop before bringing the state error back to zero with a change in sun heading.

Figure 5 shows the evolution of the state vector compared to the true values. The parts were there is a slight delay is due to the fact that no observations are read in.

Figure 6 shows the post fit residuals for the filter, with the 3σ measurement noise values. We see that the observations are read in well an that the residuals are brought back down to noise. We do observe a slight bias in the noise. This could be due to the equations of motion, and is not concerning.

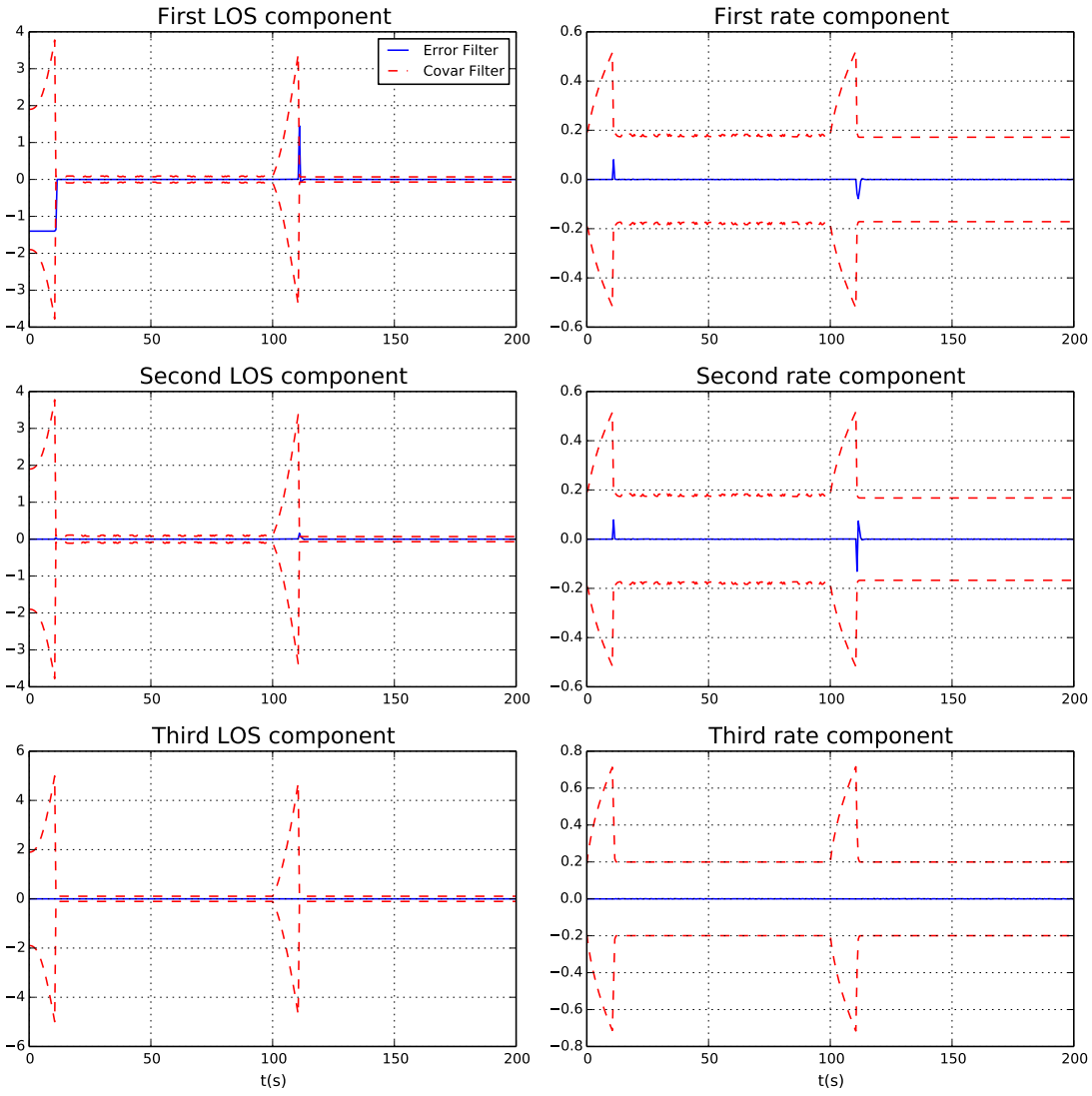

Fig. 4: State error and covariance

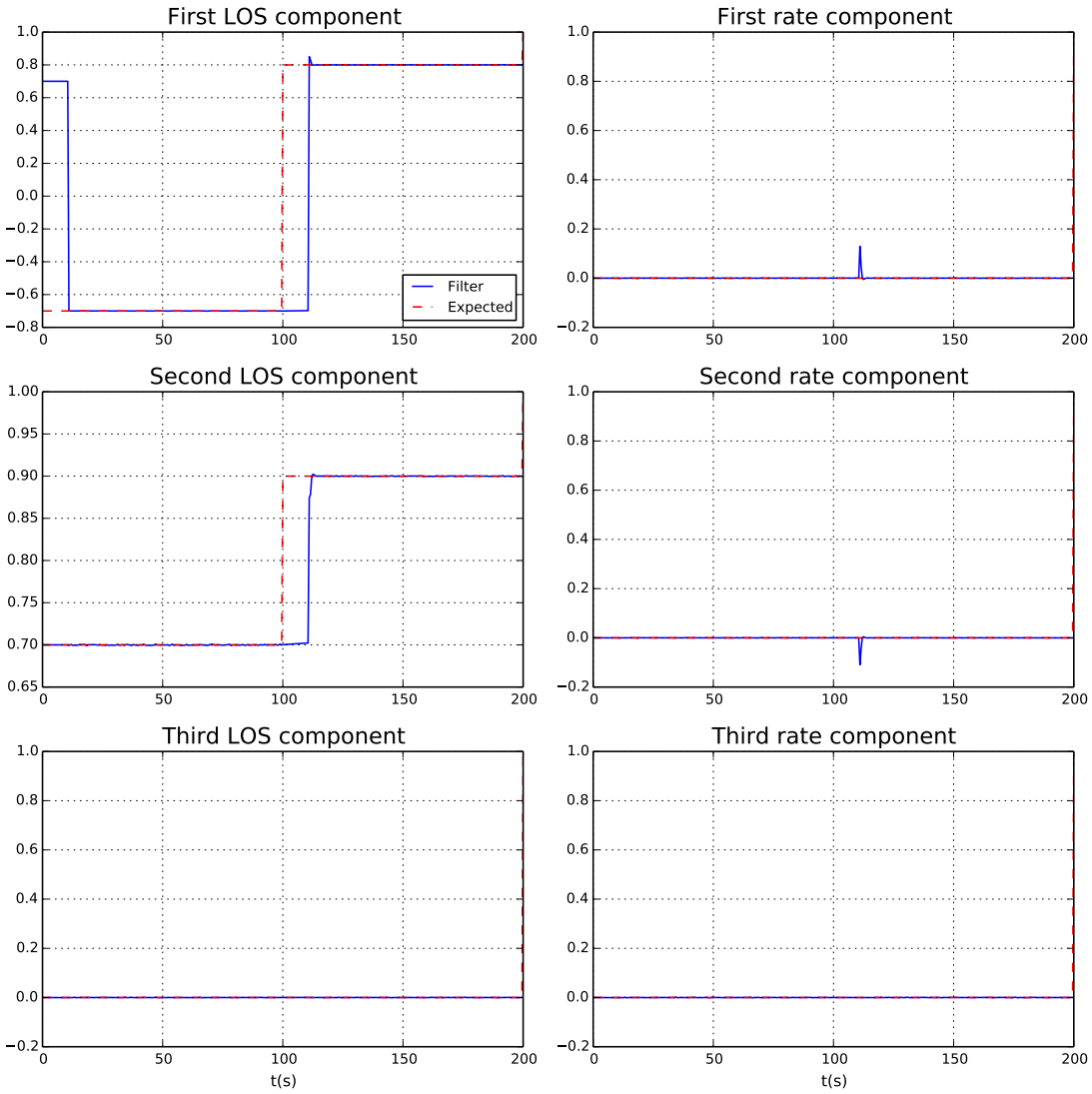

Fig. 5: States tracking target values

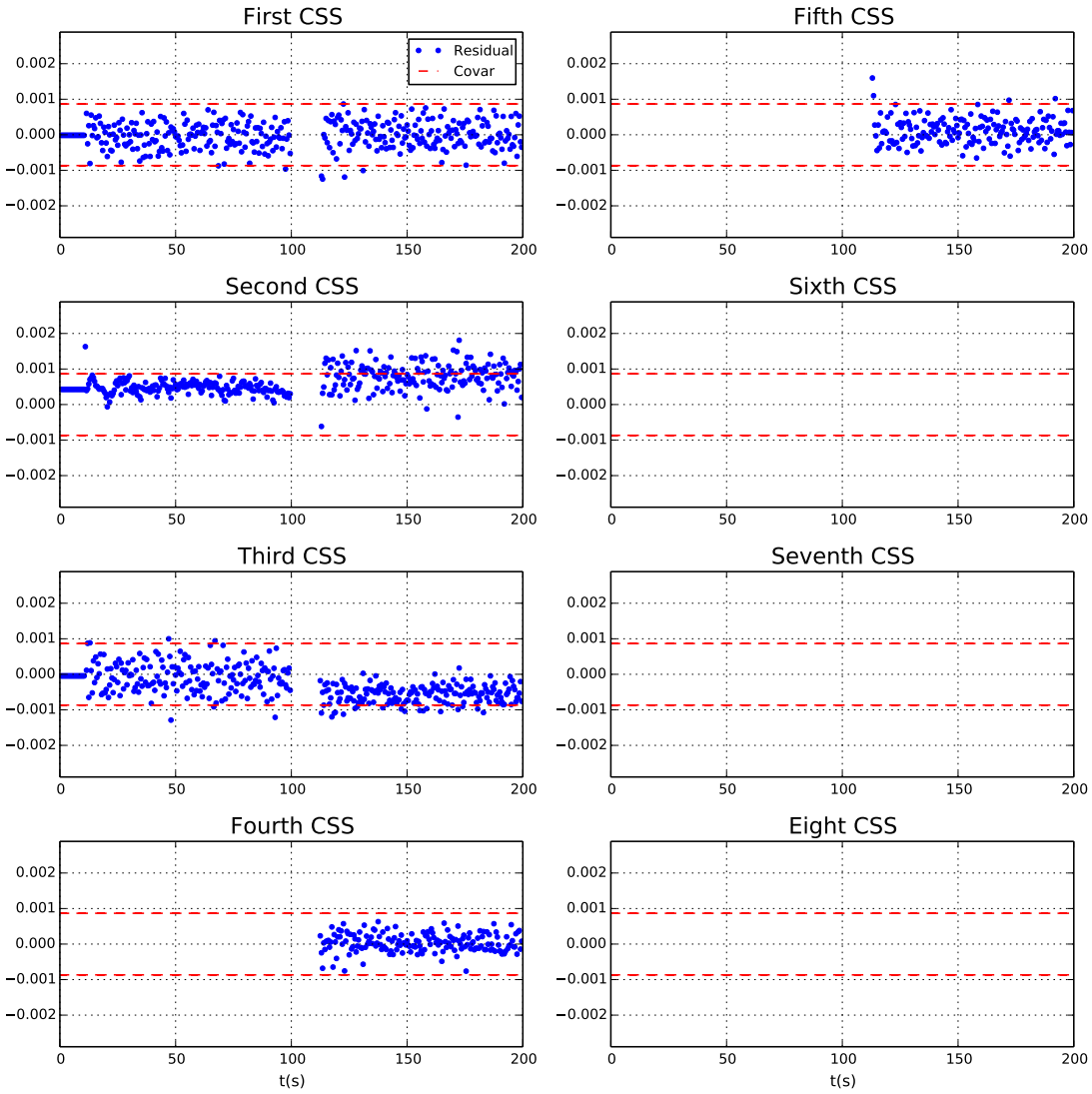

Fig. 6: Post Fit Residuals