Autonomous Vehicle Simulation (AVS) Laboratory,
University of Colorado

# Basilisk Technical Memorandum

Document ID: Basilisk-simpleNav

OVERVIEW AND TEST OF SIMPLE NAVIGATION MODEL

| | |
|---|---|
| Prepared by | S. Piggott |

| |
|---|
| **Status:** Initial Document |
| **Scope/Contents** |
| This is a report documenting the results of the Simple Navigation Model unit test created for the AVS Basilisk Simulation as part of the EMM project. It also includes a description of the module. |

| **Rev**: | **Change Description** | **By** |
|---|---|---|
| 1.0 | Initial Document | T. Teil |

## Contents

## 1 Model Description

The Simple Navigation model in the AVS Basilisk simulation is used to generate a stand-in for the real navigation system. Its use-case is to provide realistic navigation signals that are right at the spec of what ADCS claims for navigation system performance. For a typical spacecraft navigation system, the spec will almost always be at least a factor of two lower in performance than the expected nominal capabilities of the on-board navigation system. Therefore, we require the existence of a model that can provide spec-level navigation errors so that the functionality of the guidance and control subsystems can be verified as acceptable in the presence of navigation errors that are right at the spec.

The noise present in the simple navigation is designed to mimic the error signals that will be observed in the real navigation system. The true "noise" present in a vehicle's navigation system is always a combination of bias, white noise, and brown noise (or random walk). In order to provide this, a second-order Gauss-Markov process model was added to the simulation utilities that allows the user to configure a random walk process. The output of this model when applied to the vehicle position can be observed in Figure 1, and the attitude can be observed in Figure 2.

In this figure, the nominal position that the error was being distributed about was [10000.0, 0.0, 0.0] in meters and the error was bounded such that the absolute value of the error was less than 1000 m. The random walk had a white noise standard deviation of 10 meters.

In the model, the vehicle position, velocity, attitude, attitude rate, accumulated delta-velocity, and sun-pointing vector have errors applied separately. Arguably the sun-pointing vector should just use the vehicle attitude and the Sun position to get is pointing accuracy, but the Sun can be sensed independently of this and that is why there is a separate error source used for it.

The top-level model relies on a lower-level utility module called GaussMarkov that supplies the random walk process that is added to the truth states. Since the simple_nav model exercises this utility

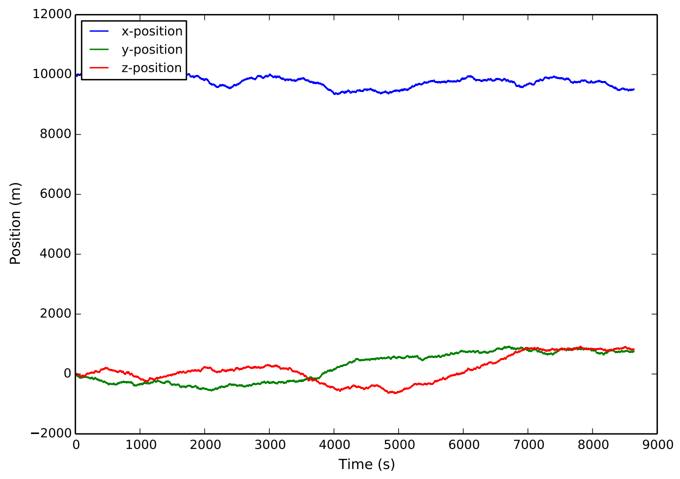

**Fig. 1:** Simple Navigation Position Signal

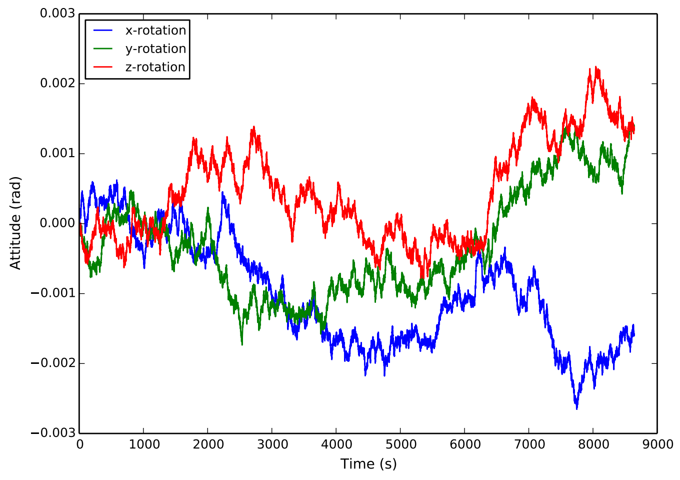

**Fig. 2:** Simple Navigation Att Signal

completely, the unit test documented here was also used to test the GaussMarkov model both in terms of functionality and in terms of code coverage. The coverage results are included in a table in the results.

## 2 Model Functions

This module allows the creation of a map function that maps the input message to the output message in which the content will be copied. The use of this can be seen in the user guide.

## 3 Model Assumptions and Limitations

The direct assumption behind the simple nav module, is that the random walk produced by a Gauss-Markov process can accurately model the uncertainties of a real sensor. Real sensor noise is a combination of bias, white noise, and brown noise. In order to provide this, a second-order Gauss-Markov process model was added to the simulation utilities that allows the user to configure a random walk process.

## 4 Test Description and Success Criteria

### 4.1 Test location

The unit test for the simple_nav module is located in:

```
SimCode/navigation/simple_nav/UnitTest/SimpleNavUnitTest.py
```

### 4.2 Subtests

This unit test is designed to functionally test the simulation model outputs as well as get complete code path coverage. The test design is broken up into three main parts:

1. Error Bound Enforcement: The simulation is run for 2.4 hours and the error bounds for all of the signals
2. Error Bound Usage: The error signals are checked for all of the model parameters over the course of the simulation to ensure that the error gets to at least 75% of its maximum error bound at some point to make sure that it isn't stuck around zero.
3. Corner Case Check: The simulation is intentionally given bad inputs to ensure that it alerts the user and does not crash.

### 4.3 Test success criteria

These tests are considered to pass if during the whole simulation time of 144 minutes, all the variables need to say within an allowable statistical error. This means that they must stay within their 1-sigma bounds $30\%$ of the time.

These sigma bounds are defined in Table 2. These are chosen in regard to the simulation's parameters and their orders of magnitude.

**Table 2:** Sigma Values

| Variable | Position | Velocity | Attitude | Rates | $\Delta$ V | Sun Position |
|---|---|---|---|---|---|---|
| Associated $\sigma$ | 5 (m) | 0.05 (m/s) | $\frac{5}{3600}$ (deg) | 0.05 (deg/s) | 0.1 (deg) | 5 (deg) |

## 5 Test Parameters

The test used for the simple navigation module doesn't have any parameters that are user set.

## 6 Test Results

### 6.1 Pass/Fail

The test results are explained below and summarized in Table 3.

1. Error Bound Enforcement: We do want to violate the error bound a statistically small number of times as most bounds are specified 3-sigma and we'll need to be at the spec to make sure it works. All signals remained inside their bounds greater than 1-sigma ( 30%) of the time.

2. Error Bound Usage: As stated above, we want to ensure that the random walk process is effectively utilizing the error bound that it has been given and not remaining mired near zero. All error signals cross up above 75% of their error bound at least once.

3. Corner Case Usage: All errors/warnings were stimulated and the simulation still ran without incident.

**Table 3:** Test Results

| SubTest | Result |
|---|---|
| Bound Enforcement | Passed |
| Bound Usage | Passed |
| Corner Case | Passed |

### 6.2 Test Coverage

The method coverage for all of the methods included in the simple_nav module are tabulated in Tables 4 and 5.

**Table 4:** Simple Navigation Test Analysis Results

| Method Name | Unit Test Coverage (%) | Runtime Self (%) | Runtime Children (%) |
|---|---|---|---|
| UpdateState | 100.0 | 0.08 | 15.0 |
| SelfInit | 100.0 | 0.0 | 0.0 |
| CrossInit | 100.0 | 0.0 | 0.0 |
| computeOutput | 100.0 | 0.0 | 0.0 |

**Table 5:** GaussMarkov Test Analysis Results

| Method Name | Unit Test Coverage (%) | Runtime Self (%) | Runtime Children (%) |
|---|---|---|---|
| computeNextState | 100.0 | 0.71 | 12.4 |
| setRNGSeed | 100.0 | 0.0 | 0.0 |
| setPropMatrix | 100.0 | 0.0 | 0.0 |
| getCurrentState | 100.0 | 0.0 | 0.0 |
| setUpperBounds | 100.0 | 0.0 | 0.0 |
| setNoiseMatrix | 100.0 | 0.0 | 0.0 |
| setPropMatrix | 100.0 | 0.0 | 0.0 |

For all of the code this test was designed for, the coverage percentage is 100%. The CPU usage of the model is higher than would be ideal although this might just be a symptom of the level of simplicity present in the overall simulation. The majority of the computations are coming from two pieces of the GaussMarkov code.

The first is the random number generator. The model is using one of the simplest random number generators in the standard template library. That is still a relatively expensive operation as random numbers are costly and we generate a new random number for each state. The second factor is in the state and noise propagation. Those are being performed with a matrix multiplication that is an $n^2$ operation. We could save some computations here in the future if we took away the cross-correlation capability from some of the states which would definitely be easy and accurate. It would just take some more code.

## 7 User Guide

The nominal set-up for the ephemeris converter is done as follows:

- `sNavObject = simple_nav.SimpleNav()`: Construct simple nav
- `sNavObject.walkBounds = sim_model.DoubleVector(errorBounds)`: add desired error bounds for the position, attitude....
- `sNavObject.PMatrix = sim_model.DoubleVector(pMatrix)`: Add the matrix of standard deviation values
- `sNavObject.crossTrans = True`
- `sNavObject.crossAtt = False`