Autonomous Vehicle Simulation (AVS) Laboratory, University of Colorado

Basilisk Technical Memorandum

Document ID: Basilisk-Corruptions

BASILISK CORRUPTION MODELING

Prepared by	S. Carnahan

Status: Draft
Scope/Contents
This documents the way that corruptions are currently applied in Basilisk and makes suggestions for changes.

Rev	Change Description	By	Date
1.0	First draft	S. Carnahan	20171018

Contents

To facilitate adequate and efficient error modeling in Basilisk, this document will discuss which "corruptions" need to be implemented. Then, for each module which needs corruptions it will discuss the current methods as well as the recommend changes and updates.

1 Desired Corruptions

Having not received FMEA documentation from LASP at this point, we have agreed that the following corruptions will be useful:

1. bias
2. random walk
3. stuck sensor (on, off, at present value, at set value, max, min)
4. noise
5. square wave
6. max/min saturation
7. triangle/ramp
8. discretization

My current understanding of these corruptions tells me they could be grouped like this:

1. Error
   (a) bias
   (b) white noise
   (c) brown noise
2. Stuck sensor

   (a) on/off
   (b) at present value
   (c) at set value
   (d) at max/min

3. Saturation
   (a) Max
   (b) Min
4. Discretization
   (a) Max/min only = 2 bit
   (b) ramp/triangle = 3 bit

2 Sensor Discussion

The IMU has most of the desired corruptions and I am familiar with it, so I will start with the IMU.

2.1 Error

The IMU has all three types of error from above. The IMU runs the code below when UpdateState() is called:

```
/* Compute true data */
computePlatformDR();
computePlatformDV(CurrentSimNanos);
/* Compute sensed data */
computeSensorErrors();
applySensorErrors(CurrentSimNanos);
applySensorDiscretization(CurrentSimNanos);
applySensorSaturation(CurrentSimNanos);
/* Output sensed data */
writeOutputMessages(CurrentSimNanos);
```

2.1.1 Bias

Bias is input to the IMU by the python code:

```
ImuSensor.senRotBias = [xRotBias, yRotBias, zRotBias]
ImuSensor.senTransBias = [xBias, yBias, zBias]
```

where the values in the bias lists are doubles and the names don't matter. The c++ code initializes these as

```
double senRotBias[3];
double senTransBias[3];
```

The bias values are unaffected by `computeSensorErrors()`. When `applySensorErrors()` is called,

the bias values are added per-axis to the errors computed in `computeSensorErrors()`. Then those values are added to the `trueValues` to get `sensedValues` . This is done no matter what, so bias is just turned off by having 0 values for bias.

2.1.2 Noise

Noise inputs are given to the IMU by the python code:

```
ImuSensor.PMatrixAccel = sim_model.DoubleVector(PMatrixAccel)
ImuSensor.walkBoundsAccel = sim_model.DoubleVector(errorBoundsAccel)
ImuSensor.PMatrixGyro = sim_model.DoubleVector(PMatrixGyro)
ImuSensor.walkBoundsGyro = sim_model.DoubleVector(errorBoundsGyro)
```

where the input matrices and bounds are calculated in python as:

```
PMatrixGyro = [0.0] * 3 * 3
PMatrixGyro[0*3+0] = PMatrixGyro[1*3+1] = PMatrixGyro[2*3+2] = senRotNoiseStd
PMatrixAccel = [0.0] * 3 * 3
PMatrixAccel[0*3+0] = PMatrixAccel[1*3+1] = PMatrixAccel[2*3+2] = senTransNoiseStd
self.PMatrixAccel = sim_model.DoubleVector(PMatrixAccel)
self.walkBoundsAccel = sim_model.DoubleVector(errorBoundsAccel)
self.PMatrixGyro = sim_model.DoubleVector(PMatrixGyro)
self.walkBoundsGyro = sim_model.DoubleVector(errorBoundsGyro)
```

where `senRotNoiseStd` and `senTransNoiseStd` are input by the user as $\mathbf{1.5}\times$ the desired standard deviation as scalar floats. The error bounds are input by the user as 3x1 lists of floats.

The values sent to the IMU are initialized in c++ as:

```
std::vector<double> PMatrixAccel;   //!< [-] Covariance matrix used to perturb state
std::vector<double> AMatrixAccel;   //!< [-] AMatrix that we use for error propagation
std::vector<double> walkBoundsAccel;//!< [-] "3-sigma" errors to permit for states
std::vector<double> navErrorsAccel; //!< [-] Current navigation errors applied to truth
std::vector<double> PMatrixGyro;    //!< [-] Covariance matrix used to perturb state
std::vector<double> AMatrixGyro;    //!< [-] AMatrix that we use for error propagation
std::vector<double> walkBoundsGyro; //!< [-] "3-sigma" errors to permit for states
std::vector<double> navErrorsGyro;  //!< [-] Current navigation errors applied to truth
```

`computeSensorErrors()` contains this code:

```
this->errorModelAccel.setPropMatrix(this->AMatrixAccel);
this->errorModelAccel.computeNextState();
this->navErrorsAccel = this->errorModelAccel.getCurrentState();
this->errorModelGyro.setPropMatrix(this->AMatrixGyro);
this->errorModelGyro.computeNextState();
this->navErrorsGyro = this->errorModelGyro.getCurrentState();
```

`setPropMatrix()` literally just sets:

```
propMatrix = prop
```

where both values are a double vector, prop is AMatrixAccel, and AMatrixAccel is just identity.

The Gauss-Markov model takes over when `computeNextState()` is called. This method iterates through the state vector given and calculates a random noise for each state. It also checks if a value greater than 0 was given as walk-bounds. If not, it doesn't do anything with the error given. If there is a bound, it applies an exponential correction to values that are too close to the edge. It states here :

```
/*! - Ideally we should find the statistical likelihood of violating a bound and use that.
However, that would require an error function, for now (PDR), just use an exponential
to pull states down that are getting "close" to the bound.*/
```

After the random numbers are generated and modified, the noise matrix (deviations) is multiplied by the random numbers vector to get noise along each axis. The weighted noise is then added to the current state of the GM model (adding the noise to the previous noise).

After noise is calculated, bias is added to the noise *within* the IMU model.

Then, these values are added to the truth values (element-wise) and stored as sensedValues. The noise added to acceleration and omega is multiplied by dt to get DV and PRV noise which is then added to those values as well.

Things to note about the Gauss Markov model:

1. I see how random walk is bounded, but not how it is created in the first place. is it the additive noise part? how is plain white noise made, then?
2. std deviations can be given per state
3. walk bounds can be given per state, but are assumed to be the same $\pm$
4. the random number generator is reseeded between `computeNextState()` calls, but not between states.
5. the walk bounds are limited exponentially, rather than using an error model?
6. GM does not use Eigen. (neither does the IMU)
7. GM requires the user to do significant work in python to generate the inputs. Can simpler inputs be given and GM or the sensor model does more work to generate noise matrices, etc?
8. the std deviation value input is actually 1.5 the output std deviation.
9. errors are computed with `computeNextState()` but must be retrieved with `getCurrentState()`

2.1.3 Discretization

After error application, sensedValues linear acceleration and angular rate are discretized. This is based on the least significant bit (LSB) input.

```
ImuSensor.gyroLSB = gyroLSBIn
ImuSensor.accelLSB = accelLSBIn
```

Essentially, the sensed value is rounded down in magnitude to the nearest multiple of a least significant bit. The difference from before and after discretization is integrated over the time step and added/subtracted with the DeltaV or step PRV value to simulate the integration error due to discretization.

Things to note about discretization:

1. This would be difficult, but not impossible, to generalize because the integration of the discretization error is not straightforward for the general case.
2. The DeltaV value is not necessarily a multiple of the least significant bit.
3. There is not bit-limit on the data. should we have a bit limit and should does a negative sign require a bit? By bit-limit I mean the output data is 2-bit or 4-bit and scaled to fit somehow. A lot of GPS receivers are 1 bit or 2 bit. Doing this turns the data into square waves, like Mar mentioned. That makes square waves just a special case of discretization.
4. Discretization is all internal to the IMU. There is no model that can be reused for other sensors right now.
5. if discretization is used for square/triangle waves, how are the two of those distinguished logically?

2.1.4 Saturation

Finally, the sensedValues are saturated. This is your basic max/min situation for each state (along each axis for the IMU). Again, the error due to saturation of acceleration and omega is integrated to get error due to saturation for DV and PRV.
Inputs are given by:

```
ImuSensor.senTransMax = senTransMaxIn
ImuSensor.senRotMax = senRotMaxIn
```

Notes on saturation:

1. saturation cannot be set separately for each state (axis). it is set separately for rotational and linear states. Saturation values are considered the same $\pm$.
2. this would not be easily generalizable because the integration of the error is not straightforward. it would be straightforward to generalize the direct discretization parts.
3. saturated values are not forced to be discretized values. This is left to the user. since the discretization is a floor, does it make sense to just reverse the order? Should discretization be a floor or a round?
4. if saturated values were given with a max and a min rather than one $\pm$ value, those values could be set equal to some desired value, giving a "stuck" behavior. alternatively, saturation could be a stuck behavior that is activated outside of certain bounds. Does this work with the way that the errors are integrated for DV and PRV?

2.2 Recommendations

2.2.1 Noise

1. Make GM model use Eigen
2. make a unit test for GM
3. Update the random walk bounds per the comment to use an error model.
4. Make an overall noise model that combines bias, white noise, and brown noise and gives a single value back to to whoever asks for it in one line. Make options for the types of noise that this outputs, with Gaussian being one type.
5. Bring noise matrix generation into noise method only asking the user for standard deviations and type of noise
6. make it so that the given standard deviation doesn't have to be scaled by 1.5.
7. this could all be set from python by user using syntax like:
   `IMUsensor.noise.type = "normal"`
   `IMUsensor.noise.std = [1.5,1.1, 1.3]`
   `IMUsensor.noise.bias = [1., 2., 3.]`

   for things like the imu where there are different states (accel, DV, PRV, etc.) should the code be re-written to make these all part of one state vector? Should it work like IMUsensor.accel.noise.bias? this means that a different instance of the noise module would be used for each separate state, while states could have multiple "sub-states" (like coordinate axes).
8. there should be a toggle for noise on/off:
   `IMUsensor.noise.on = 0`
9. this noise behavior could then be standardized across sensors like `CSS.noise.on = 1` by use of a standard noise utility.

2.2.2 Discretization

1. give an option to input number of bits
2. given an option for floor, ceiling, and round
3. how to deal with discretization of integrated values?
4. run discretization on saturation values in selfInit() to ensure saturated output is a proper discrete value
5. example:
   `IMUsensor.linear.discrete.LSB = 0.1` Currenty, LSB inputs are the float value that can be represented by a bit. Is this the best/good enough way to enter the value?

   `IMUsensor.angular.discrete.numBits = 4` Here, angular and linear are different instances of discrete that the programmer makes for the linear and angular states.

6. if number of bits are given and saturation is given, the width of a bit can be determined to discretize values across the full range of possible outputs. This can be taken care of in selfInit()?
7. integrating discretization errors remains in the realm of each sensor model.
8. make a switch: `IMUsensor.linear.discrete.on = 1`

2.2.3 Saturation

1. make saturation specific to each state (i.e. each axis for the imu)
2. make both a max and a min value for each state:
   `IMUsensor.linear.sat.max = [1,1,1]`
   `IMUsensor.linear.sat.min = [-1,-1,-0.9]`
3. make a helper function to go with saturation to simulates stuck values when asked for it. i.e., the helper function adjusts the saturation limits as necessary.
4. provide an on/off switch.

2.2.4 General for IMU

1. convert to Eigen
2. saturation checks the instantaneous value. It could be updated to check the average value over the past time step as well.
3. make corruption groups (.linear and .angular)
4. attach .noise, .discrete, and .sat to the corruption groups.

I am not sure of the "best practice" for building these types of things. Please provide whatever critiques you see fit. With the