

AN OBJECT-ORIENTED SENSOR AND SENSOR SYSTEM DESIGN

Jason R. Neuhaus
Senior Member
Aerospace/Software Engineer

Unisys Corporation
NASA Langley Research Center
Mail Stop 169
Hampton, VA 23681

Abstract

This paper presents an Object-Oriented Design for modeling sensors, and their associated errors and failures. By applying Object-Oriented techniques to the modeling of sensors, a generic sensor model and a sensor system to manage the sensors were created. Using this design, the process of adding new sensors at any location on the aircraft, taking into account the changes in dynamics at a point other than the center of gravity, has been greatly simplified. This design also includes a comprehensive set of methods for implementing errors and failures that can be applied to any sensor.

Introduction

The Object-Oriented¹ Sensor and SensorSystem classes were designed to provide a generic method for aircraft simulations to model sensors and failures, as well as provide a simplified way to obtain changes in sensor inputs based on the location on the aircraft. The design presented in this paper is currently used at NASA Langley Research Center in the Langley Standard Real-time Simulation in C++ (LaSRS++) framework. This design allows any aircraft or vehicle in the LaSRS++ framework to easily create a sensor system to manage sensors at specified locations and apply failures. The interface provided by the sensor system, settings available for each sensor, and large number of sensor failure modes available allow aircraft to easily simulate sensors with a large degree of flexibility.

The main classes used to implement this design include a SensorSystem, Sensor, DynamicsAtPoint, and

Copyright © 2001 by the American Institute of Aeronautics and Astronautics, Inc. No copyright is asserted in the United States under Title 17, U.S. Code. The U.S. Government has a royalty-free license to exercise all rights under the copyright claimed herein for Governmental purposes. All other rights are reserved by the copyright owner.

FailureMode class. These classes are discussed in more detail in the LaSRS++ Sensor Classes section below.

The SensorSystem class is the interface between the aircraft and the sensors. It maintains a list of all sensors for the aircraft, calculates the dynamics at the sensor locations, and allows sensor failures to be activated. Sensors are modeled using the Sensor class. The location of the sensor may be specified so that effects due to the location of the sensor relative to the center of gravity may be simulated. The effects due to the location of the sensors are calculated using the DynamicsAtPoint class.

The Sensor class computes an output, the sensor signal, based on an input signal, errors, and failures using several internal methods. Available errors include a scale factor, constant bias, random bias, and random noise. The sensor can also implement a sampling rate and can use a first and/or second order filter to model the sensor behavior. The FailureMode class is used by the sensor class to model sensor failures.

The FailureMode class allows the user to select from a list of failure types that influence the sensor signal. The failed output value is based on the selected failure type.

The DynamicsAtPoint class allows for the calculation of position, orientation, velocity, angular velocity, acceleration, and angular acceleration at a sample point given the same values at a reference point and the distance from the reference point to the sample point.

Through the use of the SensorSystem, Sensor, DynamicsAtPoint, and FailureMode classes, this design allows any aircraft or vehicle in the LaSRS++ framework to easily create a sensor system to manage sensor models and sensor failures.

LaSRS++ Sensor Classes

The following classes were used in the LaSRS++ Sensor System implementation. The main classes include DynamicsAtPoint, Sensor, FailureMode, and SensorSystem. Some of the other classes mentioned in the following sections, or in the Unified Modeling Language (UML) diagrams², are discussed in the Miscellaneous Classes section. Figure 1 shows the LaSRS++ Sensor System class overview.

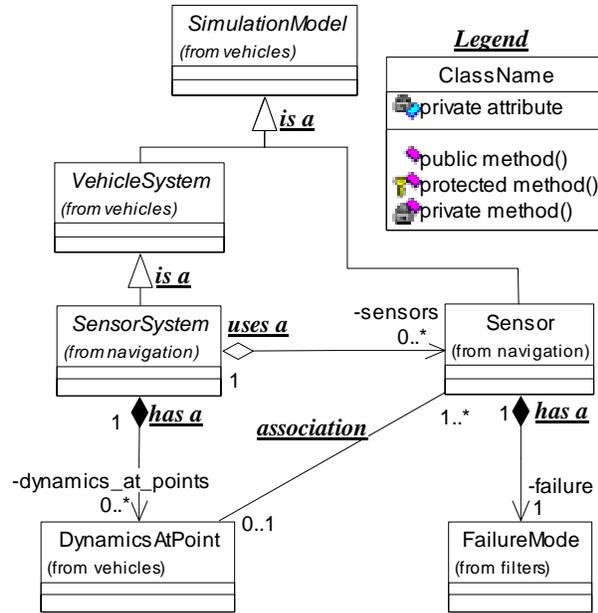


Figure 1 Class Overview

DynamicsAtPoint

The DynamicsAtPoint class was created to allow for the computation of vehicle dynamics at a point other than the center of gravity; or any other point at which the position, orientation, velocity, angular velocity, acceleration, and angular acceleration are known. Figure 2 shows the UML diagram for the DynamicsAtPoint class. The class makes use of three main points, a reference origin, reference point, and sample point.

The reference origin is the origin point of the reference frame. The reference origin for most aircraft is the reference center of gravity. This origin point should always be at a fixed point on the aircraft.

The reference point is the point at which the dynamic inputs (velocity, acceleration, etc.) are known, typically the center of gravity (CG). This point is specified

relative to the reference origin in the reference frame coordinate system. The reference frame coordinate system will typically be the body frame.

The sample point is the point at which the dynamic inputs will be translated. It can be specified as a distance from the reference origin or the reference point in the reference frame coordinate system.

The equations, listed below, assume that the frame is rigid, and therefore do not take into account rotations or accelerations of the sample point frame relative to the reference frame. The class allows the ability to specify an extra coordinate frame rotation from the reference frame to a frame used in the position calculation. This has no affect on the calculation of the orientation, velocity, acceleration, angular velocity, or the angular acceleration at the sample point. This extra frame rotation was added to allow the position calculation to be used in a different frame than the other calculations. For example, the position calculation is computed in either world relative coordinates or relative to a geographic reference point by the SensorSystem class.

$$\begin{aligned}\bar{x}_s &= \bar{x}_r + R_{rp} \cdot \bar{r}_{rs} \\ \bar{\theta}_s &= \bar{\theta}_r \\ \bar{v}_s &= \bar{v}_r + \bar{\omega}_r \times \bar{r}_{rs} \\ \bar{\omega}_s &= \bar{\omega}_r \\ \bar{a}_s &= \bar{a}_r + \bar{\alpha}_r \times \bar{r}_{rs} + \bar{\omega}_r \times (\bar{\omega}_r \times \bar{r}_{rs}) \\ \bar{\alpha}_s &= \bar{\alpha}_r\end{aligned}$$

where :

*_s = sample point value

*_r = reference point value

\bar{r}_{rs} = radius vector from the reference point to the sample point in the reference frame coordinate frame

\bar{x} = position vector

$\bar{\theta}$ = orientation vector

R_{rp} = rotation matrix from the reference frame to the frame used by the position input

\bar{v} = translational velocity vector

$\bar{\omega}$ = angular velocity vector

\bar{a} = translational acceleration vector

$\bar{\alpha}$ = angular acceleration vector

For the LaSRS++ Sensor System, the reference origin is located at the reference CG. The body frame is used as the reference frame. The reference point is kept at the CG and the sample point is fixed on a specified point on the aircraft (discussed further in the SensorSystem section).

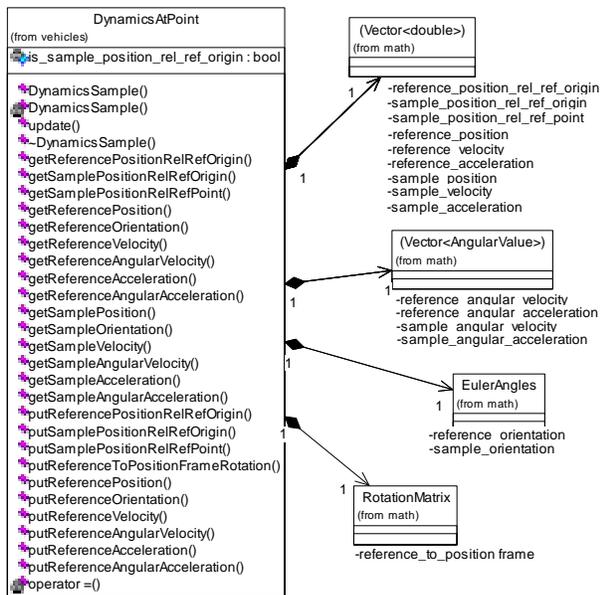


Figure 2 DynamicsAtPoint Class Diagram

Sensor

The Sensor class is used to model a sensor and provides a wide assortment of ways to model any associated errors of a particular sensor. It outputs a sensor signal as a function of an input value and any errors or failures that are set. Available errors include a scale factor, constant bias, random bias, and random noise. The sensor can also have a sampling rate, failure modes, and can use a first and/or second order filter to form the sensor output. Figure 3 shows a UML diagram of the Sensor class.

The scale factor introduces a scaling error. A constant bias adds a constant, non-random bias to the sensor signal. A random bias allows a mean, standard deviation, and random number seed to be specified to add a random (gaussian distribution) bias to the sensor signal. A random noise allows a standard deviation, lag, and random number seed to be specified. The lag is used to implement a first order Markov process to simulate the noise spectrum for the sensor model.

The sampling rate can be used in order to update a sensor slower than the simulation rate. The first and second order filters can be used to create the desired sensor model, introducing lag, phase shift, or oscillation to the sensor input.

Sensor failures are implemented using the FailureMode class.

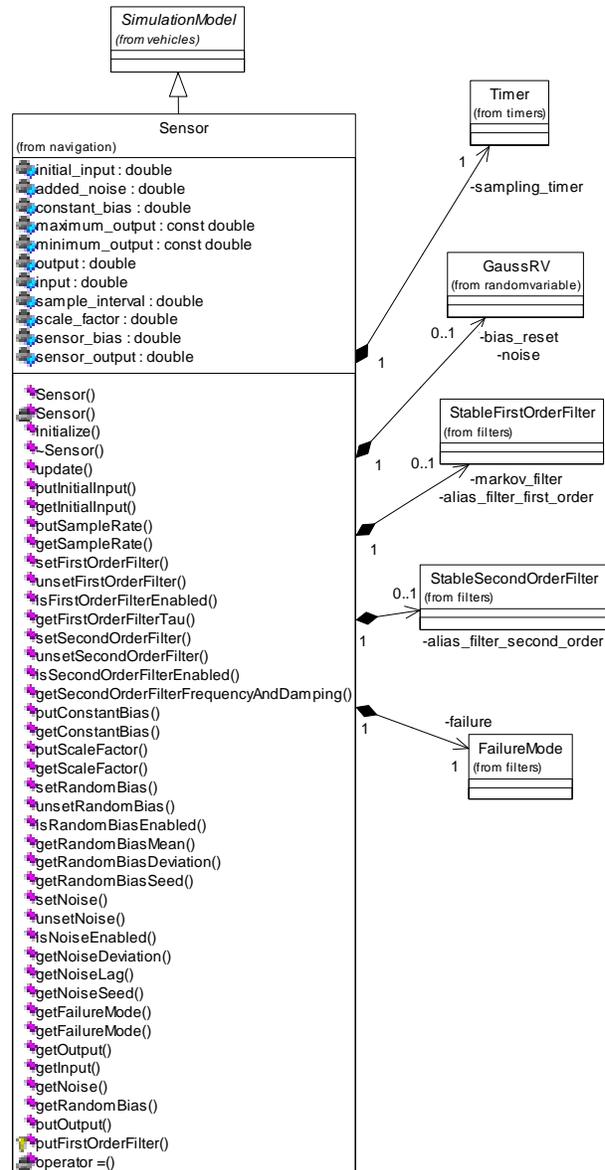


Figure 3 Sensor Class Diagram

FailureMode

The FailureMode class, shown in Figure 4, modifies an output value based on a selected failure mode, if any. There are nine basic failure types and six random failure types. The basic failure types are no fail, fail zero, fail frozen, fail high, fail low, fail reverse, fail bias, fail cycle, and fail random. The random failure types are random zero, constant interval zero, random glitch, constant interval random glitch, constant interval constant glitch, and constant random noise.

The no failure mode passes the input through without any failure. Fail zero always returns zero as the output. Fail frozen returns the previous output value. Fail high returns the maximum value of the output. Likewise,

fail low returns the minimum value of the output. Fail reverse returns the input value with the sign changed. Fail constant returns a user specified constant value. Fail bias returns the input value plus a constant bias. Fail cycle ramps the output constantly and limits the output by rolling over from minimum to maximum or visa versa. Fail random returns an output based on the random failure types selected.

When the basic failure type is set to fail random, the FailureMode class will base the output on the random failure type selected. The random zero failure returns a zero (dropout) at random time intervals. The constant interval zero failure outputs zero at specified periodic intervals. The random glitch failure outputs random values at random intervals. The constant interval random glitch failure outputs random values at specified periodic intervals. The constant interval constant glitch failure outputs a specified error value at specified periodic intervals. The constant random noise failure outputs random noise.

In addition to the failure modes and random failure types, the user may specify a persistence time and whether or not to ramp to the target value at a specified ramp rate. The persistence time specifies how long the output of a random failure is held while failing. The ramp to target can be used to gradually ramp the output value to the target value, dependent on the failure mode, at a specified rate.

SensorSystem

The SensorSystem class, shown in Figure 5, keeps track of sensors and sensor positions by using the Sensor and DynamicsAtPoint classes. When constructed, the SensorSystem class requires the number of sensors, location of the reference center of gravity in the sensor reference frame, and a rotation matrix from the sensor reference frame to the body frame to be specified. The sensor reference frame can be any frame, although for most aircraft this will be either the Airplane Reference System (ARS), specified by a Body Station, Butt Line, and Water Line, or centered at the reference CG. The reference CG is assumed to not move relative to the aircraft during a simulation run.

Sensors are registered to the system through one of the registration methods. These methods allow a sensor to be registered with or without a position specified.

When a sensor is registered without a position using the registerSensor() method, no DynamicsAtPoint instance is created for it. Registration without a position is

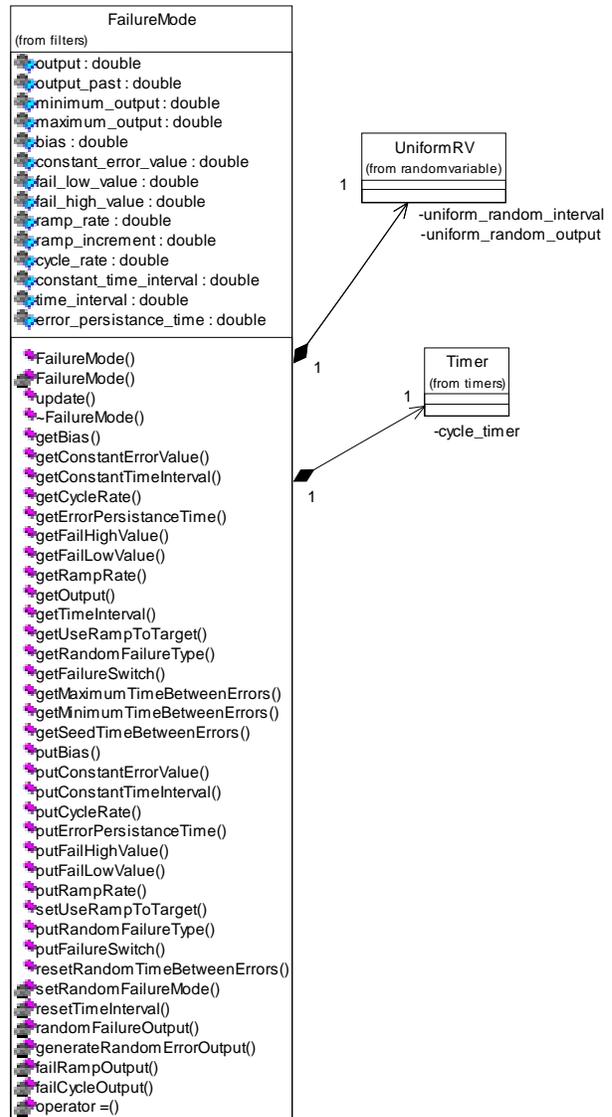


Figure 4 FailureMode Class Diagram

typically done for a sensor that is at the same position as another sensor, always located at the CG (unrealistic as it may be), or if the sensor already computes an input based on a position and does not wish the sensor system to handle the position shift. Registration with a position is done through the registerSensorAtPosition() method.

Registering a sensor with a position, specified in the sensor reference frame, allows the SensorSystem class to compute the position, orientation, velocity, etc. at the sensor location. An optional argument when registering the sensor at a specified position allows a Geographic Reference Relative Info Handle (GeoRefRelInfoHandle) to be specified. The GeoRefRelInfoHandle class allows calculations to be done for the aircraft relative to a fixed point on the ground, such as a

navigation transmitter or runway threshold. The sensor must also specify whether or not the velocity computed is air-relative, to include wind velocity, or body relative.

If two sensors are located at the same position, the linkSensors() method can be used to ensure the dynamics calculated for one sensor are the same for all sensors that are linked to it. Linking sensors is done to save some computation time by only updating one DynamicsAtPoint instance for each sensor location on the aircraft, instead of updating one for each physical sensor.

The update() method is used to update all of the DynamicsAtPoint instances for the sensor position specified. The values computed at the sample points are then used to calculate the inputs to the individual sensors.

The SensorSystem updates the individual DynamicsAtPoint instances differently if a GeoRefRelInfoHandle was specified for the corresponding sensor during registration. If the handle was specified, the reference point position is calculated relative to the geographic reference point, otherwise the position is calculated in world relative coordinates. The SensorSystem class provides a method for computing the latitude, longitude, and altitude of a point given in world relative coordinates using the same world shape data as the simulation. The orientation is input as the local frame to body if a GeoRefRelInfoHandle was specified, otherwise the orientation is input as the rotation angles from local vertical to body.

None of the other reference point inputs to the DynamicsAtPoint instances are dependent on whether or not a geographic handle was provided. Air relative velocities or body velocities are input for the velocity input based on whether the sensor specified that it wanted air relative velocities or not when it was registered. The inertial acceleration of the aircraft in body coordinates minus gravitational effects is used as the acceleration input. The body angular velocity and angular acceleration are used for the angular velocity and angular acceleration inputs. The position of the reference point is updated based on the current location of the CG relative to the reference CG.

Once all of the DynamicsAtPoint instances have been calculated, the calculateSensorInputs() method is called. This method allows the aircraft specific sensor system to calculate the inputs to the sensors based on

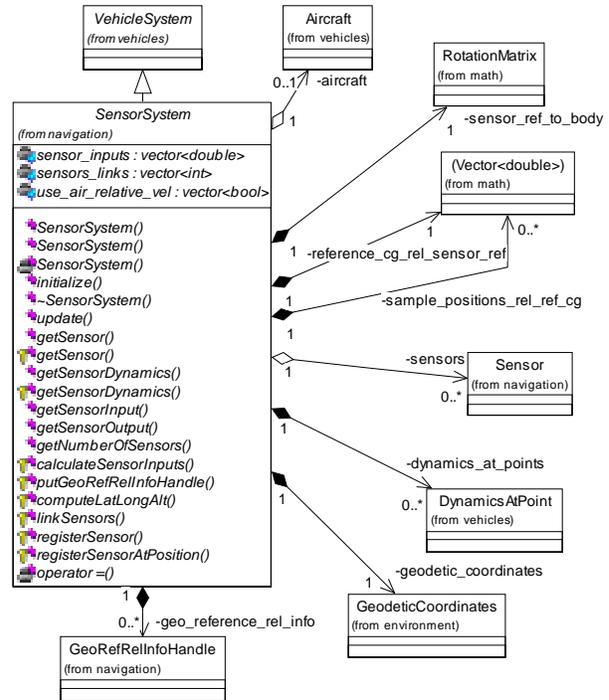


Figure 5 SensorSystem Class Diagram

the dynamics that have been calculated at the sensor locations. For example, an alpha (angle of attack) sensor would use the body x and z components of the air-relative velocity at the sensor location to compute the local angle of attack. A Global Positioning System (GPS) receiver would use the world relative position calculated, when registered without a geographic handle, to compute the latitude, longitude and altitude at the sensor location. A glideslope receiver would use the position relative to the transmitter, by using the transmitter as the geographic handle when registered, to compute the glideslope angle as seen by the receiver.

Miscellaneous Classes

This section discusses some of the classes that are used by the sensor system design, but are not otherwise described in this document.

The SimulationModel class is the base class for all models in the simulation. Its main purpose is to standardize the way models initialize and to allow access to a simulation mode and timer.

The simulation mode is used to determine what state the simulation is in. These states include reset, trim, hold, and operate.

The simulation timer is implemented with the Timer class. The Timer class is used for various timing

functions. It is often used when a class needs to keep a separate timer from the main simulation timer.

The `VehicleSystem` classes manage communications between an aircraft and the simulation models of its components. They are responsible for calculating and passing the expected inputs into the simulated components. They are also responsible for retrieving output from the component and manipulating it into the data that the aircraft expects. For example, the `ControlSystem` class derives from `VehicleSystem`. The control system gets sensor outputs from the sensor system (another class that derives from `VehicleSystem`) and passes them to a control law.

The `Aircraft` class is the aircraft object being simulated.

The `Vector<double>` template class is simply a three element array of doubles.

The `Vector<AngularValue>` template class is similar to the `Vector<double>` template class, but contains an array of angular values. Angular values are doubles that contain information about an angle and can return their value in either degrees or radians.

The `EulerAngles` class is a storage class for standard Euler angle orientations

The `RotationMatrix` class is used to perform coordinate rotations between different coordinate frames.

The `GaussRV` class is used to generate random values in a Gaussian distribution. The class allows a seed, mean, and standard deviation to be defined.

The `UniformRV` class is used to generate random values in a uniform distribution. The class allows a seed, minimum, and maximum value to be defined.

The `StableFirstOrderFilter` class implements a first order filter with additional checking to make sure the digital, z-transform, implementation of the filter is stable. If the time step is greater than 2τ , where τ is the filter time constant, the digital implementation of the filter is unstable. If the filter is unstable the filter will always return the steady state output for the filter.

The `StableSecondOrderFilter` class is similar to the `StableFirstOrderFilter` class except it implements a second order filter.

The `GeodeticCoordinates` class is used to specify positions relative to a reference ellipsoid as coordinates of latitude, longitude, and altitude.

The `GeoRefRelInfoHandle` class provides calculations and accessors for individually selectable position, velocity, and acceleration data of an aircraft relative to a geographic reference point (`GeoRefPoint`), for example, a navigation transmitter or runway threshold.

Results

In this section, example sensor placement and sensor errors are demonstrated. Figure 6 through Figure 10 show the dynamics of the aircraft at the CG for the 10 second run.

GPS Receiver

In this example, a GPS receiver is placed 50 feet forward and 20 feet up from the reference CG. Figure 11 through Figure 13 show the latitude, longitude, and altitude of the center of gravity versus the sensor position. This example includes no sensor errors or dynamics.

The GPS sensor outputs are affected only by the orientation of the aircraft and the latitude, longitude, and altitude of the CG. The aircraft begins heading north at 10,000 feet. During the 10 second run, the difference in latitude remains small, with the sensor remaining slightly north of the CG (Figure 14). The minimum difference between the sensor and CG latitudes occurs around 5 seconds when the pitch and yaw are at their maximum values, and the aircraft is rolled somewhat.

The longitude remains the same until shortly after 2 seconds when the aircraft heading changes (Figure 15). As the aircraft starts to turn east, the sensor reads a longitude slightly east of the CG.

The sensor altitude changes between approximately 25 and 38 feet higher than the CG throughout the run (Figure 16). The primary contribution to the change in altitude is the pitch angle, with the roll angle having a smaller adverse affect on the relative altitude.

Accelerometer

In this example, an accelerometer triad is placed 80 feet forward and 2 feet down from the reference CG. Figure 17 through Figure 19 show the sensed accelerations at the center of gravity versus the sensor position. This example includes no sensor errors or dynamics.

The body x axis accelerometer follows similar trends to the acceleration at the CG (Figure 17). The difference in the sensed acceleration between 2 and 4 seconds is primary due to the yaw rate peak. The yaw rate term ($-r^2 \cdot x$) adds an additional -2.5 foot per second squared of acceleration at just past 3 seconds. At 5 seconds, a small spike appears because of a jump in the body pitch acceleration.

The body y axis accelerometer is shown in Figure 18. The primary effect on the side acceleration is due to the large roll rates between 2 and 5 seconds.

The body z axis accelerometer is shown in Figure 19. The difference between the sensed and CG accelerations is due primarily to the pitch acceleration and the product of the roll and yaw rates. The spikes in acceleration at 1 and 5 seconds are primarily caused by the pitch acceleration.

Alpha Sensor

In this example, an angle of attack sensor is placed 5 feet aft and 60 feet to the right of the reference CG. Figure 20 shows the angle of attack at the CG, as well as the computed angle of attack at the sensor position. The angle of attack at the sensor position is calculated based on the air-relative velocity components calculated at the sensor position. This example includes no sensor errors or dynamics.

The sensor output was obtained by calculating the angle of attack using the body x and z axis velocities from an instance of the DynamicsAtPoint class. The angle of attack calculated at the sensor position was then passed as the input to the sensor. The primary difference in the angle of attack at the CG and the sensor position is caused by the body roll rate.

Alpha Sensor with Errors and Model Dynamics

This example copies the example above (Alpha Sensor), and adds a first order filter with a time constant of 0.025 seconds, a constant bias of 0.2 degrees, and noise. The noise is modeled with a standard deviation of 0.1 degrees, a lag of 0.03 seconds.

Figure 21 shows the difference in the sensor outputs between the previous sensor output with and without the errors listed above.

Alpha Sensor with Failures

This example duplicates the example above (Alpha Sensor with Errors and Model Dynamics) plus the implementation of some failures. The failures include a

fail high at 2 seconds, a fail frozen at 4 seconds, and a fail random/random zero failure at 6 seconds. The fail high and fail frozen failures each last 1 second. The fail random is held until the end of the sampling period. The sensor has a maximum value of 50 degrees, a minimum of -10 degrees, ramp to target enabled with a ramp rate of 20 degrees/sec, and an error persistence time of 0.0125 seconds. The random time interval was set to cause errors to repeat between 0.025 and 0.7 seconds.

Figure 22 shows the effects of adding the failures listed above, and the errors from the previous example to the alpha sensor output.

Conclusions

The Sensor and SensorSystem classes implemented in the LaSRS++ framework provide a flexible framework for the implementation of sensor models. Through the use of the SensorSystem, Sensor, DynamicsAtPoint, and FailureMode classes, the user has been given a well-featured framework to model sensors. These sensor models may be placed at any point on the aircraft to include the effects of the position relative to the center of gravity into the input of the sensor. The sensor models have the added ability to implement sensor dynamics, add errors to the sensor signal, and fail the sensor.

Future Work

Additional features that could be useful to the sensor system include the ability to remove the assumption that the aircraft is a rigid body. This can be accomplished by deriving a class from the DynamicsAtPoint class that can add in the effects of movement and orientation changes of the sample point frame to the calculations done in the DynamicsAtPoint class. In this way, an angle of attack sensor could be added to a wing flapping model to take into account the effects of a non-rigid wing on the sensor reading.

References

- 1 Booch, Grady. *Object-Oriented Analysis and Design With Applications*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.
- 2 Muller, Pierre-Alain. *Instant UML*. Wrox Press Ltd. Chicago, Illinois, 1997.

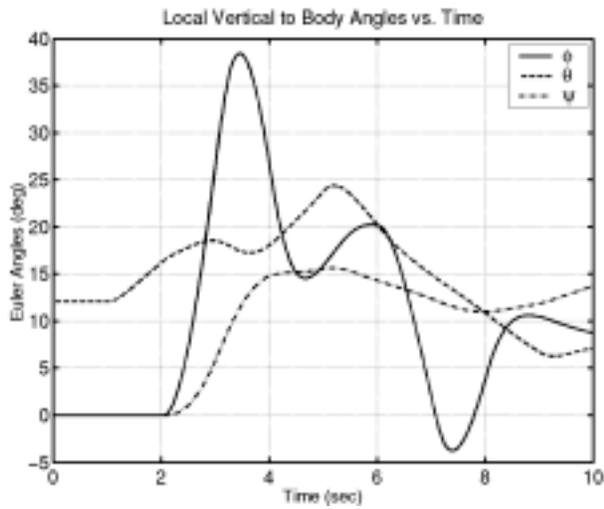


Figure 6 Euler Angles

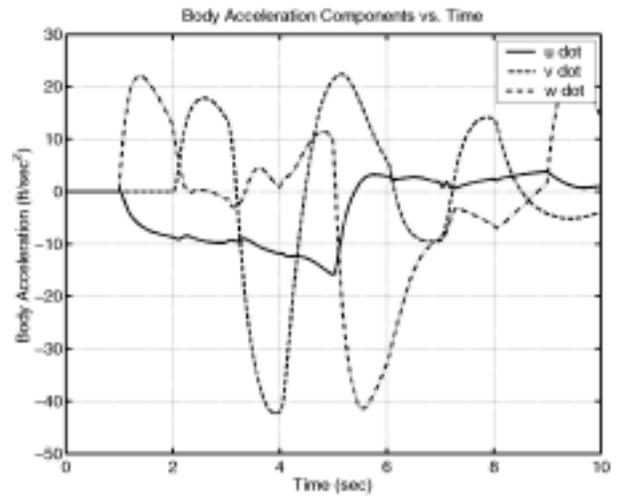


Figure 9 Acceleration Components

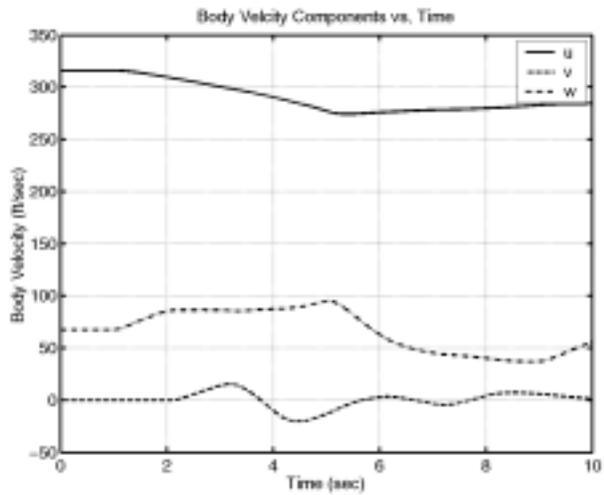


Figure 7 Velocity Components

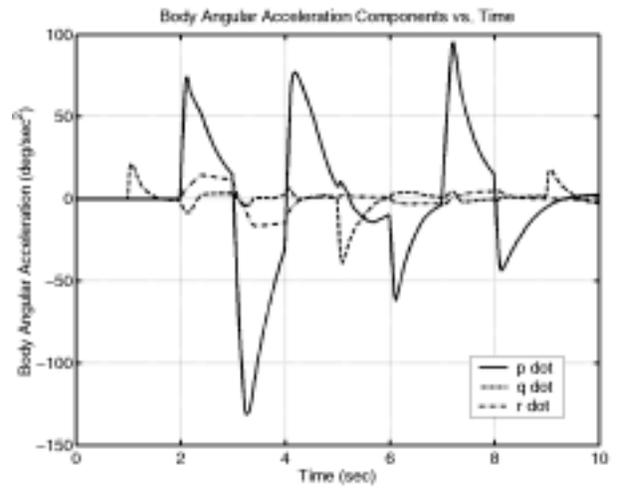


Figure 10 Angular Acceleration

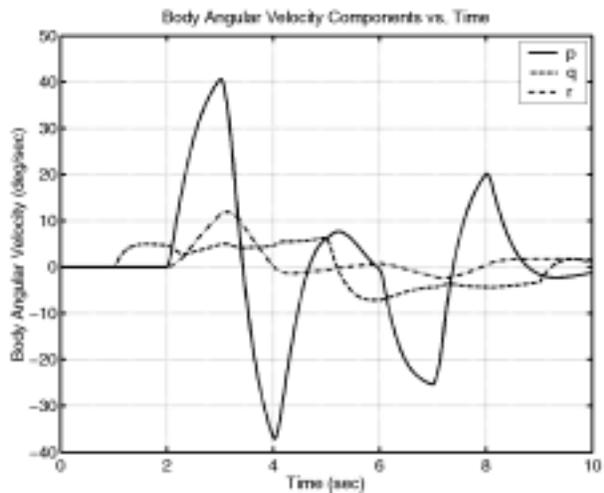


Figure 8 Angular Velocities

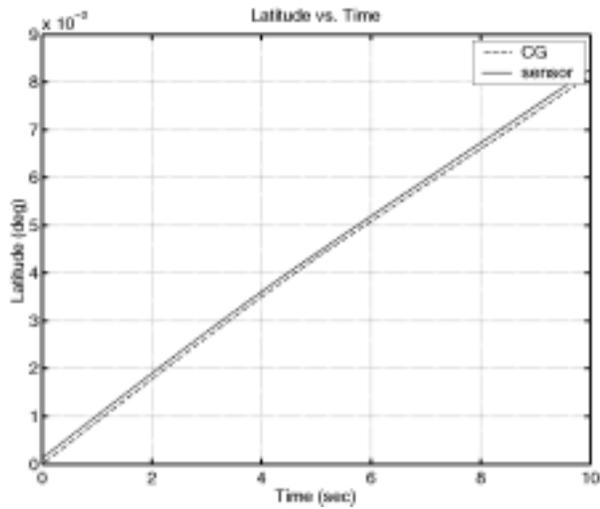


Figure 11 GPS Latitude

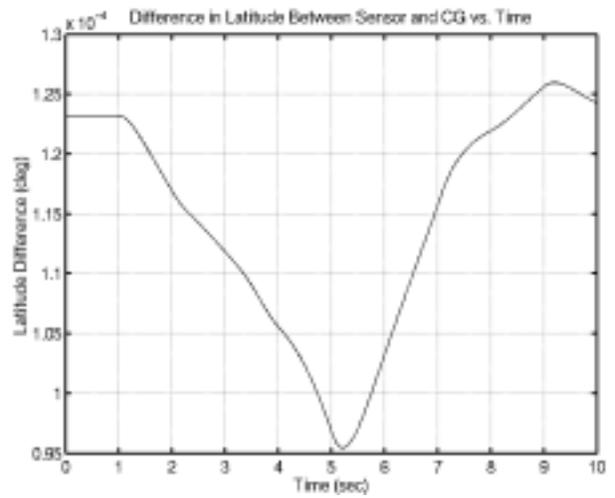


Figure 14 GPS Latitude Difference

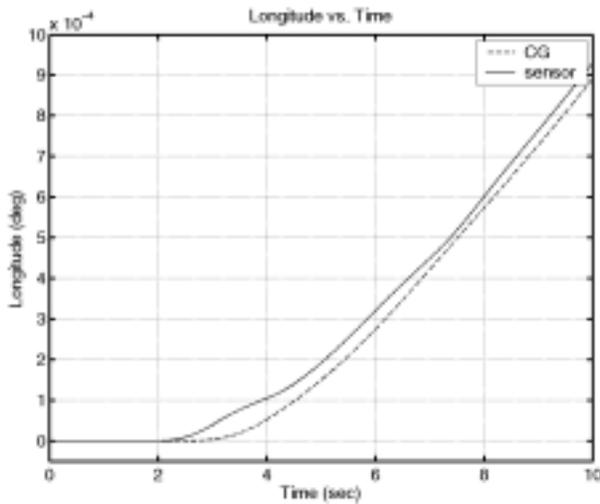


Figure 12 GPS Longitude

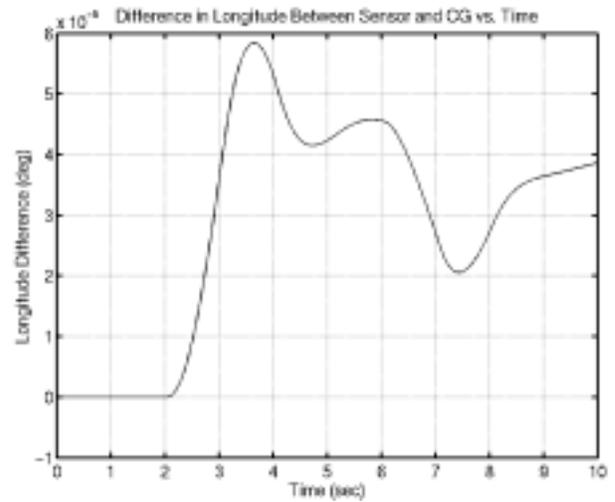


Figure 15 GPS Longitude Difference

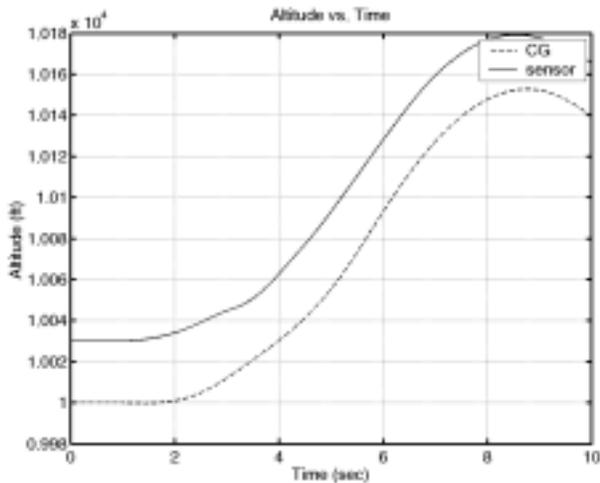


Figure 13 GPS Altitude

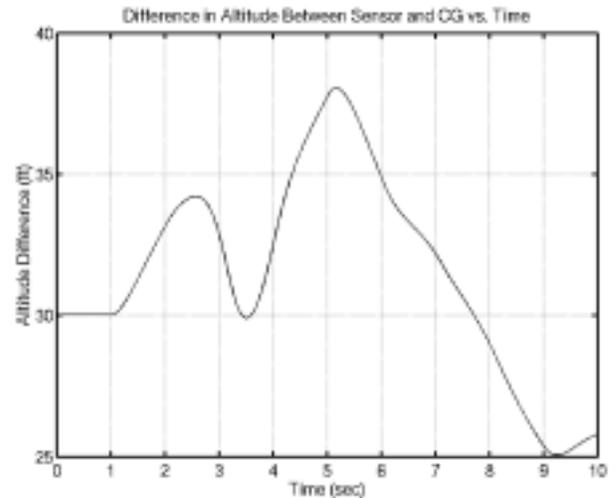


Figure 16 GPS Altitude Difference

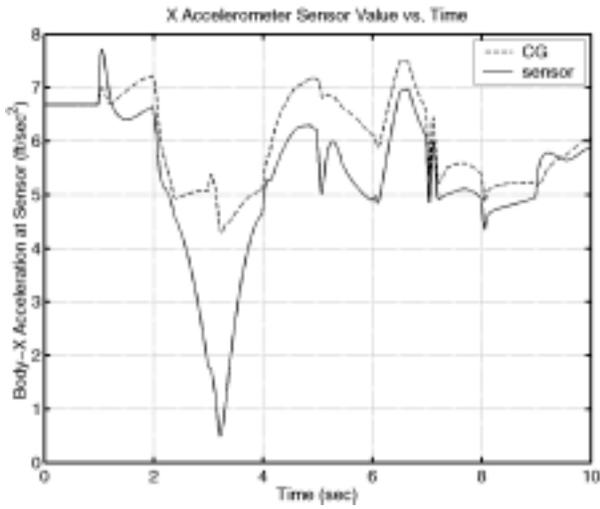


Figure 17 X Accelerometer

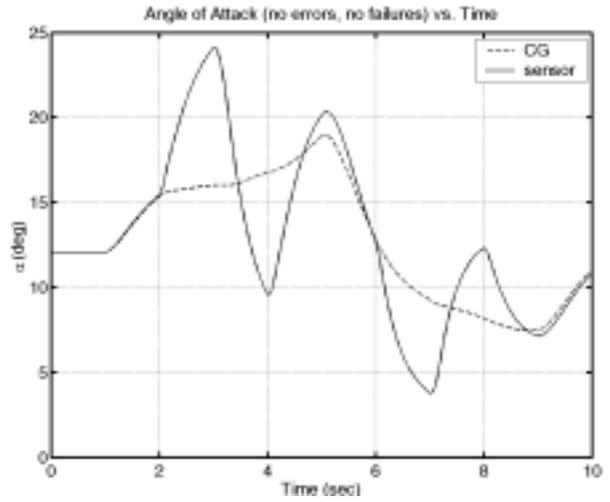


Figure 20 Angle of Attack

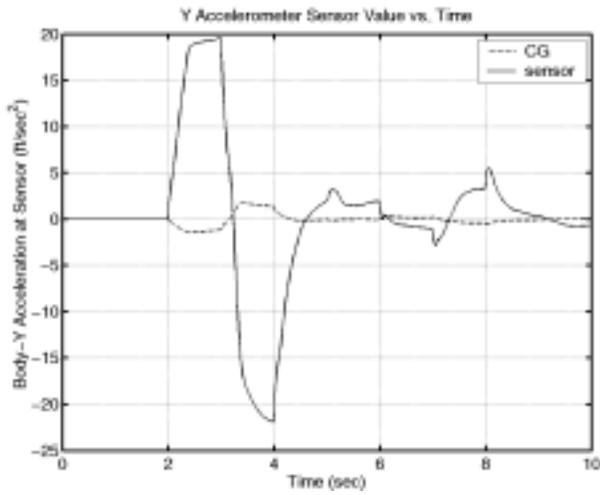


Figure 18 Y Accelerometer

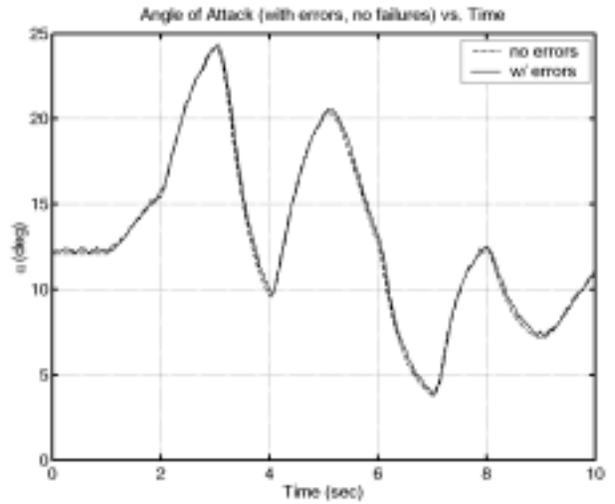


Figure 21 Angle of Attack with Errors

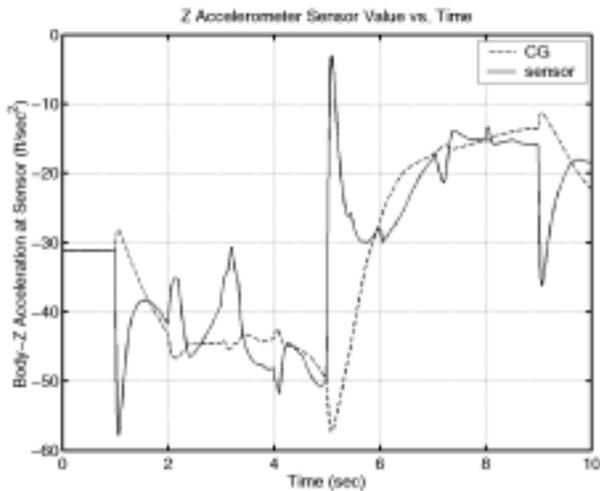


Figure 19 Z Accelerometer

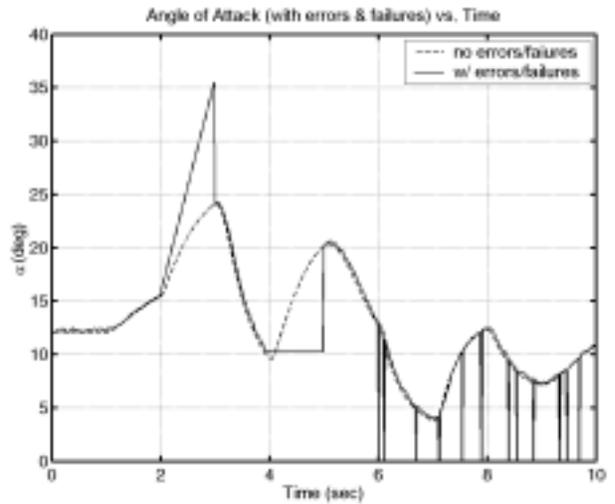


Figure 22 Angle of Attack with Errors and Failures