JSBSim

An open source, platform-independent, flight dynamics model in C++

Jon S. Berndt
and the JSBSim
Development Team

JSBSim

An open source, platform-independent, flight dynamics model in C++

Jon S. Berndt
& the JSBSim Development Team

[This version released on:6/9/2011]

ACKNOWLEDGEMENTS

This software is the result of work done by many people over the years. Tony Peden has been contributing to the growth of JSBSim almost from day 1. He is responsible for the initialization and trimming code. Tony also incorporated David Megginson's property system into JSBSim. Tony hails from Ohio State University, with a degree in Aero and Astronautical Engineering. David Culp developed the turbine model for JSBSim, and crafted several aircraft models that use it including the T-38. David has experience flying many types of military and commercial aircraft, including the T-38, and the Boeing 707, 727, 737, 757, 767, the SGS 2-32, and the OV-10. David is an aerospace engineer, a graduate from the U.S. Air Force Academy. David Megginson came from a long involvement as a core FlightGear developer. David correlated our flight dynamics with his general aviation flying experience to aid in maximum realism, among other things. David designed the property system that both FlightGear and JSBSim use. He is well known for his contributions to XML technology, and wrote the easyXML parser that both FlightGear and JSBSim use. Erik Hofman has done a bit of everything, hunting down aircraft data, creating flight models (F-16), and performing some programming. He also tests for IRIX compatibility. Erik has a degree in Computer Science. Mathias Frölich added a versatile per-gear ground elevation capability, and many other things. Mathias is a mathematician from Germany. Agostino De Marco has created a broadly capable cost/penalty trim analysis feature for JSBSim, and has used JSBSim by itself and together with FlightGear at the University of Naples. David Luff from the United Kingdom provided the original piston engine model. Ron Jensen has steadily refined it. Engineers with many years of simulation experience, Lee Duke and Bill Galbraith have contributed suggestions and ideas that have improved JSBSim. Bruce Jackson from NASA Langley Research Center – who has been involved in the development and use of a variety of simulations for many years – has been supportive and helpful, and the simulation code he wrote in C many years ago, (“LaRCSim”) was instructive in the early development of JSBSim. Curt Olson, who coordinates the development of FlightGear and some of its constituent parts (SimGear) has been a great help over the years in countless discussions of simulation, control theory, and many other topics. Working with the FlightGear community has made JSBSim a better tool. Finally, the user and developer community has worked well to bring JSBSim to where it is today. Thanks are due to anyone who has ever taken the time to report a bug or to ask for a feature.

PREFACE

JSBSim was conceived in 1996 as a lightweight, data-driven, non-linear, six-degree-of-freedom (6DoF), batch simulation application aimed at modeling flight dynamics and control for aircraft. Since the earliest versions, JSBSim has benefited from the open source development environment it has grown within and from the wide variety of users that have contributed ideas for its continued improvement.

About this document

This document is split up into several parts. This is because JSBSim can be viewed from several different perspectives: from that of a flight vehicle model developer, from that of an integrator who will incorporate JSBSim into a full flight simulation architecture with visuals, and from that of a software developer who wants to adapt or enhance JSBSim with additional capabilities.

There is a QuickStart section that explains how to get started with JSBSim quickly. That is followed by Section One, which is a User's Manual. The User's Manual explains how to use JSBSim to make simulation runs, to create aircraft models, to write scripts, and how to perform various other tasks that do not involve changes to program code in JSBSim itself. Section Two is a Programmer's Manual. The Programmer's Manual explains the architecture of JSBSim – how the code is organized and how it works. Section Three is the Formulation Manual which contains a description of the math model and algorithms present in JSBSim. Section Four is a collection of some examples and case studies showing how JSBSim has been used.

What this document is and what it is not

This document is not an exhaustive reference on the derivation of the equations of motion and flight dynamics. For a text on that, see (Stevens & Lewis, 2003), and (Zipfel, 2007). This document is meant to be the authoritative document about JSBSim.

Conventions used

When XML definitions are given, items in brackets ("[]") are optional.

TABLE OF CONTENTS

Quickstart

1.1 Getting the Source

JSBSim can be downloaded as source code in a "release" - a bundled set of source code files (a snapshot) that was taken at some point in time when the source code was thought to be stable. A release is made at random times as the developers see fit. The code can be downloaded as a release from the project web site at:

http://sourceforge.net/project/showfiles.php?group_id=19399

The source code can also be retrieved from the revision management storage location using a tool called cvs (concurrent versions system). CVS is available as an optional tool in the Cygwin environment under Windows, but is usually standard with Linux systems. Directions on downloading JSBSim can be found at the project web site at http://www.jsbsim.org. Briefly, the JSBSim code (including example files needed to run JSBSim) can be downloaded using cvs as follows (when prompted for a password from the login command – the first line - simply press the Enter key):

```
cvs -d:pserver:anonymous@jsbsim.cvs.sourceforge.net:/cvsroot/jsbsim \
  login
cvs -z3 -d:pserver:anonymous@jsbsim.cvs.sourceforge.net:/cvsroot/jsbsim \
  co -P JSBSim
```

Alternatively – and perhaps more easily – a tarball can be downloaded that contains the very latest version of the code as it exists in cvs at this page: http://jsbsim.cvs.sourceforge.net/jsbsim/. Click on the Download GNU tarball text. The tarball can be extracted using the tar program, or another extractor such as 7zip.

1.2 Getting the Program

At the same location, executable files for some platforms (Windows, Cygwin, Linux, IRIX, etc.) may also be downloaded. It should be noted that at present (and unless otherwise stated in the release notes for a release) the example aircraft and script files, etc. must be downloaded as part of the source code release. Another alternative is to check out the files from the JSBSim cvs repository.

1.3 Building the Program

JSBSim can be built using standard gnu tools, automake, autoconf, g++, make, etc. There also exists a project file for building JSBSim using the Visual C++ development environment from Microsoft. For a Unix environment (cygwin, Linux, etc.), once you have the source code files, you should be able to change directory ("cd") to the JSBSim base directory and type:

```
./autogen.sh
make
```

Note that the make command can be told to use all available cores if your CPU is multi-core. Specify the "-j #" option, where the "#" is replaced with the number of cores you have plus one. For a quad core of course this would lead to the make command:

```
make –j 5
```

The base directory is the location under which all other files and directories are found. When JSBSim source code is downloaded or checked out from cvs, the exact name of the directory may vary, and there may be intermediate directories under which the main JSBSim directory is placed. One thing is certain, the JSBSim root directory is the one where the autogen.sh files is to be found, and under which the src/ subdirectory will be found.

When the autogen and make commands are run as directed, this should result in an executable being created under the src/ subdirectory. If you are building JSBSim under Microsoft Visual C++, the executable will be placed under the Debug/ or Release/ subdirectory, depending on how you are building the executable. Project files for Microsoft Visual C++ are found in the JSBSim root directory, and are named JSBSim.vcproj (for Microsoft Visual C++ 2008 Express), and JBSim.vcxproj (for Microsoft Visual C++ 2010 Express).

Note: If you would like to optimize your executable for speed, and you know the architecture for your computer (for instance, "nocona"), you could do that as follows:

```
./autogen.sh CFLAGS="-O3 -march=nocona" \
             CPPFLAGS="-O3 -march=nocona" \
             CXXFLAGS="-O3 -march=nocona"
```

The "nocona" option supports compilation for 64-bit extensions, and for the MMX, SSE, SSE2, and SSE3 instruction sets. You can find out more about the Gnu compilers at gcc.gnu.org.

1.4 Running the Program

There may be several options specified when running the standalone JSBSim application that is provided with a JSBSim release or built using the instructions above:

```
Usage (items in brackets are optional):

  JSBSim [script name] [output directive files names] <options>

 Options:
 --help Returns a usage message
 --version Returns the version number
 --outputlogfile=<filename> Sets/replaces the name of the data log file
 --logdirectivefile=<filename> Sets name of data log directives file
 --root=<path> Sets the JSBSim root directory (where src/ resides)
 --aircraft=<filename> Sets the name of the aircraft to be modeled
 --script=<filename> Specifies a script to run
 --realtime Specifies to run in actual real world time
 --nice Directs JSBSim to run at low CPU usage
 --suspend Specifies to suspend the simulation after initialization
 --initfile=<filename> Specifies an initialization file to use
 --catalog Directs JSBSim to list all properties for this model
     (--catalog can be specified on the command line along with a
      --aircraft option, or by itself, while also specifying the
      aircraft name, e.g. --catalog=c172)
 --end-time=<time> Specifies the sim end time (e.g. time=20.5)
```

```
 --property=<name=value> Sets a property to a value.
     For example: --property=simulation/integrator/rate/rotational=1
NOTE: There can be no spaces around the = sign when an option is
followed by a filename
```

If you have built JSBSim from source code there will be an executable under the src/ subdirectory. You can run JSBSim by supplying the name of a script:

```
src/jsbsim scripts/c1723.xml
```

If you are running the program using the Microsoft Visual C++ IDE, you will need to go to Program Options and set the Command Arguments to run the script as shown above.[1]

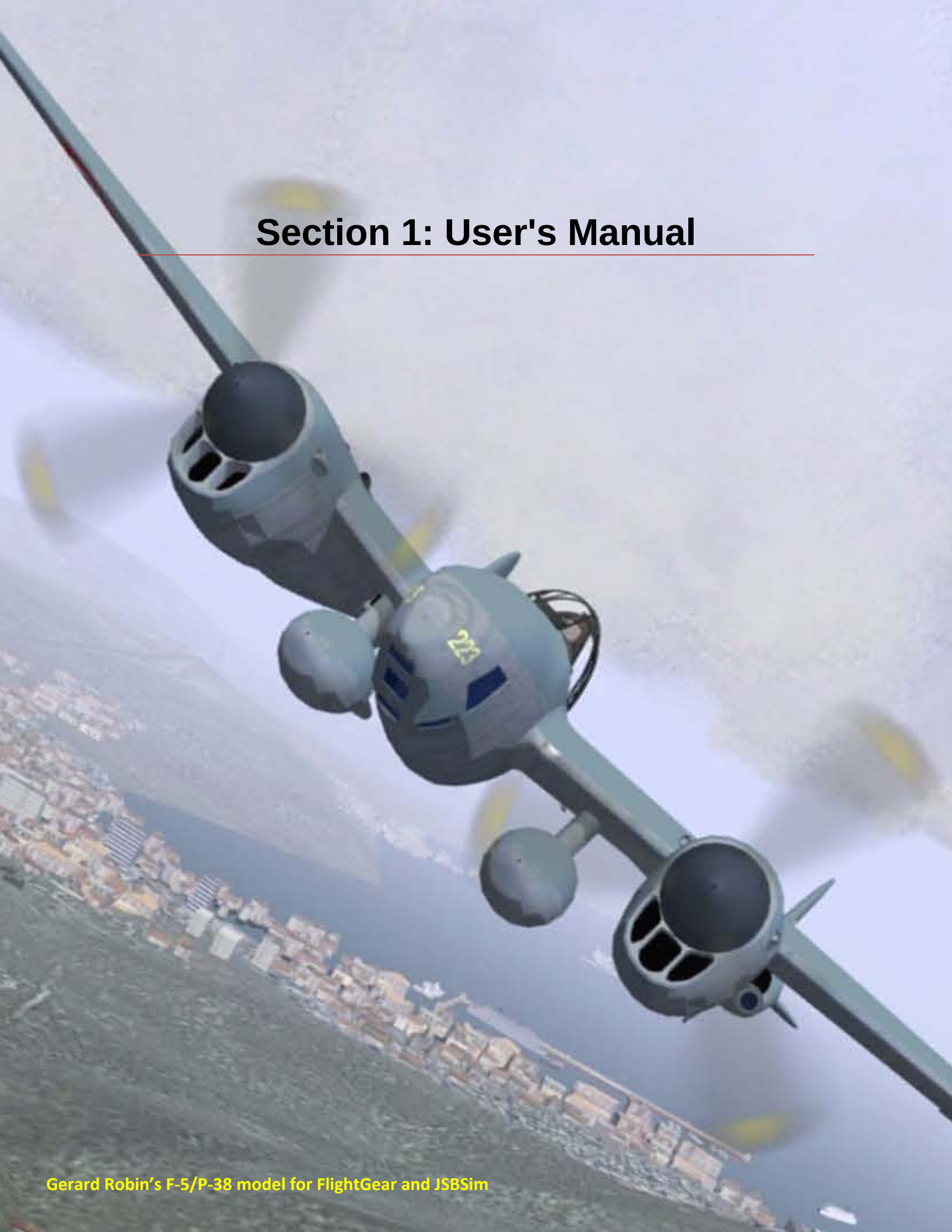
You may have simply downloaded the executable. In this case, you should create the scripts/, aircraft/, and engine, subdirectories. You should place aircraft models under the aircraft/ subdirectory, in another subdirectory that has the same name as the aircraft. For instance, if you create a B-2 flight model, you would place that in the aircraft/B2/ subdirectory, and the aircraft flight model name would be called B2.xml.

1.5 Getting Support

The best way to get support on JSBSim is to subscribe and post questions to the relevant mailing list. You can find out about the mailing lists here: www.sf.net/mail/?group_id=19399.

[1] If you are using an older version of Microsoft Visual C++, make sure you have the latest service packs installed for your version of Windows and for the Visual C++ tool you are using. Otherwise, you may get application crashes.

Section 1: User's Manual



Gerard Robin's F-5/P-38 model for FlightGear and JSBSim

1. Overview

1.1 What, exactly, is JSBSim?

From an application programming perspective, JSBSim is a collection of program code mostly written in the C++ programming language (but some C language routines are included). Some of the C++ classes that comprise JSBSim model physical entities such as the atmosphere, a flight control system, or an engine. Some of the classes encapsulate concepts or mathematical constructs such as the equations of motion, a matrix, quaternion, or vector. Some classes manage collections of other objects. Taken together, the JSBSim application takes control inputs, calculates and sums the forces and moments that result from those control inputs and the environment, and advances the state of the vehicle (velocity, orientation, position, etc.) in discrete time steps.

JSBSim has been built and run on a wide variety of platforms such as a PC running Windows or Linux, Apple Macintosh, and even the IRIX operating system from Silicon Graphics. The free GNU g++ compiler easily compiles JSBSim, and other compilers such as those from Borland and Microsoft also work well. See the Programmers Guide for more information.

From an end-user perspective (perhaps a student doing research), JSBSim can be viewed as sort of a “black box” which is supplied with input files in XML format. These XML files contain descriptions of an aerospace vehicle, engines, scripts, and so on. When these files are loaded into JSBSim, they direct it to model the flight of that vehicle in real-time as part of a larger simulation framework (such as FlightGear or OpenEaagles), or faster than real-time in a batch mode. Each run of JSBSim would result in data files showing performance and dynamics of the vehicle being simulated and studied.

From a software integrator perspective (such as someone integrating JSBSim within a larger simulation framework), JSBSim is a library that can be called, supplied with inputs (such as control inputs from the pilot), and returning outputs (describing where the vehicle is at any moment in time).

1.2 Who is it for, and how can it be used?

The JSBSim flight dynamics model (FDM) software library is meant to be reasonably easy to comprehend, and is designed to be useful to advanced aerospace engineering students. Due to the ease with which it can be configured, it has also proven to be useful to industry professionals in a number of ways. It has been incorporated into larger, full-featured, flight simulation

Figure 1. Aerocross Echo Hawk

applications and architectures (such as FlightGear, Outerra, and OpenEaagles), and has been used as a batch simulation tool in industry and academia.

1.2.1 *Examples of use*

1.2.1.1 Aerocross Echo Hawk

JSBSim has been used for Hardware-in-the-Loop (HITL) testing of the Aerocross Echo Hawk UAV. Custom code was written to interface with the flight hardware (PC/104-based system) via RS-232/422/485, proportional analog I/O, discrete I/O, and sockets, but the core simulation code was unaltered JSBSim code. Pilot/operator training also relies on JSBSim as the 6-DoF code.

1.2.1.2 DuPont Aerospace Company

JSBSim was used at duPont Aerospace Company along with Matlab for real-time HITL simulation and pilot/operator training. Rex duPont of duPont Aerospace Company explained the project,

In the 1990s duPont Aerospace Company was building an airplane to test its concept for a vertical takeoff fan jet transport plane. We had developed a Microsoft Windows-based flight simulator that we had used to test the flying qualities of the proposed craft. However, we needed a simulation that we could use in real time so that we could test the flight characteristics on a full-sized mockup with the flight actuators operating in the loop. We settled on the FlightGear simulator, using the JSBSim flight dynamics model because we could get the full code, it was nicely organized so that we could create new subprograms to match our aircraft, and support was readily available.

Figure 2. duPont Aerospace DP-1 in a tethered hover test. (Courtesy of duPont Aerospace Company)

We developed simultaneously a Matlab simulator for the use in developing more effective autopilot guidance systems, since our primary task became to take the aircraft off using the autopilot alone and to hover in place for 30 seconds. This would show definitively that the control system was sufficiently robust. Therefore, we built into each relevant module of the Matlab simulator and the JSBSim derivative simulator a series of unit tests that provided a sequence of inputs to each model that could be cross verified to ensure that the two systems stayed in sync.

We used the JSBSim system to test a number of dynamic issues that were not easily testable with the Matlab model, especially issues involving pilot feel and the controllability

during transition to and from hover. These issues are hard to evaluate in the pure control-system world of Matlab because during transition the underlying force structure is continuously varying as aerodynamic forces become more important and pure thrust control forces less.

We also did parametric studies on such issues as the sensitivity of the key parameters in the aerodynamic simulation to possible errors in estimation. These were done by having the pilot fly a series of standard maneuvers designed to test the aircraft's response when one or more parameter was degraded by as much as 50% (without the pilot knowing which one was changed).

We simulated various servo bandwidths as well, testing to see at what point the flying characteristics became unacceptable. This helped define the characteristics needed. What pilots desire from control systems is almost always different from the optimal theoretical parameters.

Additionally, we developed a number of HUD display systems that facilitated operations during hover, where very precise control of ground speed is required. Eventually we achieved a system that allowed a young engineer who did not even have a pilot's license to take off and hold a hover at constant altitude to within one foot for over 30 seconds.

We finally achieved our goal of autopilot controlled take-off and hover in two flights of approximately 45 seconds duration on September 30, 2007. Both flights were terminated because one of the engines ran out of fuel, rather than for any control problems.

1.2.1.3 MITRE Air Traffic Studies

JSBSim is being used at MITRE in developing a 6-DoF simulation of the FMS (Flight Management System) behavior during CDAs (Continuous Descent Arrival) and OPDs (Optimized Profile Descent). Both the standalone version of JSBSim (for batch runs) and a version integrated with FlightGear have been used. Additional control system components have been created to support specific lateral and vertical navigation studies. JSBSim has also been extended to handle output of messages over a socket to another application used at MITRE which provides a view similar to what an Air Traffic Control operator would see.

1.2.1.4 U.S. Department of Transportation

In work done with and for the U.S. D.O.T., a human pilot math model was developed using JSBSim as the 6-DoF (six degree-of-freedom) simulation core.

1.2.1.5 The University of Naples, Italy, Federico II

The University of Naples has a motion base flying/driving simulator that is driven by FlightGear and JSBSim. The simulator has a three-screen visual presentation that provides a 190 degree field-of-view. The JSBSim source code was modified to provide a force feedback capability.

JSBSim has been used at the University of Naples as a tool supporting risk level evaluations of near-ground flight operations. One of the practical problems considered in those collision risk studies consisted in the evaluation of threat posed to flight operations when a new obstacle (such as a building or radar tower) is placed inside the airport area. The risk evaluation has been performed by varying the obstacle geometry and location.

The risk assessment procedure has been based on the analysis of statistical deviations of aircraft trajectories from the “normal” flight path, evaluating the probability of a generic trajectory

to cross a given “protection” area enveloping the potential obstacle. Within this framework, the operational scenario has been formally described and implemented in order to run multiple computer simulations.

2. Concepts

2.1 Simulation

While the JSBSim user does not need to know some of the finer details of the flight simulator operation, it can be helpful to understand basically how JSBSim works. Some of the most important concepts are described in this section. Frames of reference are used to describe the placement and location of various items in a vehicle model. There is flexibility in how the units of measure can be specified when defining a vehicle model – both English and metric units are u supported. The use of "Properties" permits JSBSim to be a generic simulator, providing a way to interface the various systems with parameters (or variables). Properties are used throughout the configuration files that describe aircraft and engine characteristics. Obviously, math plays a big part in modeling flight physics. JSBSim makes use of data tables, as flight dynamics characteristics are often stored in tables. Arbitrary algebraic functions can also be set up in JSBSim, allowing broad freedom for describing aerodynamic and flight control characteristics.

2.2 Frames of Reference

Before moving into a description of the configuration file syntax, one must understand some basic information about some of the frames of reference used in describing the location of objects on the aircraft.

Structural Frame - This frame is a common manufacturer's frame of reference and is used to define points on the aircraft such as the center of gravity, the locations of all the wheels, the pilot eye-point, point masses, thrusters, etc. Items in the JSBSim aircraft configuration file are located using this frame. In the structural frame the X-axis increases from the nose towards the tail, the Y-axis increases from the fuselage out towards the right (when looking forward from the cockpit), and of course the Z-axis then is positive upwards. Typically, the origin for this frame is near the front of the aircraft (at the tip of the nose, at the firewall, or in front of the nose some distance). The X-axis is typically coincident with the fuselage centerline and passes through the propeller hub (thrust axis). Positions along the X axis are referred to as *stations*. Positions along the Z axis are referred to as *waterline* positions. Positions along the Y axis are referred to as *buttline* positions.

Note that the origin can be anywhere for a JSBSim-modeled aircraft, because JSBSim internally only uses the relative distances between the CG and the various objects – not the discrete locations themselves.

Body frame - In JSBSim, the body frame is similar to the structural frame, but rotated 180 degrees about the Y axis, with the origin coincident with the CG. In this frame the aircraft forces and moments are summed and the resulting accelerations are integrated to get velocities.

Stability frame - This frame is similar to the body frame, except that the X-axis points into the relative wind vector projected onto the XY plane of symmetry for the aircraft. The Y-axis still

points out the right wing, and the Z-axis completes the right-hand system.

Wind frame This frame is similar to the Stability frame, except that the X-axis points directly into the relative wind. The Z-axis is perpendicular to the X-axis, and remains within the aircraft body axis XZ plane (also called the reference plane). The Y-axis completes a right hand coordinate system.

2.3 Units

JSBSim uses English units for internal calculations almost exclusively. However, it is possible to input some parameters in the configuration file using different units. In fact, to avoid confusion, it is recommended that the unit always be specified. Units are specified using the "unit" attribute. For instance, the specification for the wingspan looks like this:

```
<wingspan unit="FT"> 35.8 </wingspan>
```

The above statement specifies a wingspan of 35.8 feet. The following statement specifying the wingspan in meters would result in the wingspan being converted to 35.8 feet as it was read in:

```
<wingspan unit="M"> 10.91 </wingspan>
```

The two statements for wingspan are effectively equivalent.

The following conversions are currently supported [using the abbreviations: "M" – Meters, "FT" – Feet, "IN" – Inches, "IN3" – Cubic inches, "CC" – Cubic centimeters, "M3" – Cubic meters, "FT3" – Cubic feet, "LTR" – Liter, "M2" – Square meters, "FT2" – Square feet, "LBS" – Pounds, "KG" – Kilogram, "SLUG*FT2" – Slug-ft^2, "KG*M2" – Kilogram-meter^2, "RAD" – Radian, "DEG" – Degree, "LBS/FT" – pounds per foot, "N/M" – Newtons per meter, "LBS/FT/SEC" – pounds per foot per second, "N/M/SEC" – Newtons per meter per second, "WATTS" – Watts, "HP" – Horsepower, "N" – Newtons, "LBS" – Pounds, "KTS" – Knots, "FT/SEC" – Feet per second, "M/S" – Meters per second, "FT*LBS" – Foot-pounds, "N*M" – Newton-meters, "INHG" – Inches of mercury, "PSF" – Pounds per square foot, "ATM" – atmospheres, "PSI" – Pounds per square inch, "PA" – Pascals]:

Length

M → FT
M → IN
KM → FT

Area

M2 → FT2

Volume

CC → IN3
M3 → FT3
LTR->IN3

Mass & Weight

KG → LBS

Moments of Inertia

KG*M2 → SLUG*FT2

Angles

RAD → DEG
DEG → RAD

Spring force

N/M → LBS/FT

Damping force

N/M/SEC → LBS/FT/SEC

Power

WATTS → HP
HP → WATTS

Force

N → LBS

Velocity

KTS → FT/SEC
M/S → FT/SEC

Torque

N*M → FT*LBS

Pressure

INHG → PSF
ATM → INHG
INHG → ATM
PSF → INHG
INHG → PA
PA → INHG
PSI → INHG

2.4 Properties

Simulation programs need to manage a large amount of state information. With especially large programs, the data management task can cause problems:

- Contributors find it harder and harder to master the number of interfaces necessary to make any useful additions to the program, so contributions slow down.
- Runtime configurability becomes increasingly difficult, with different modules using different mechanisms (environment variables, custom specification files, command-line options, etc.).
- The order of initialization of modules is complicated and brittle, since one module's initialization routines might need to set or retrieve state information from an uninitialized module.
- Extensibility through add-on scripts, specification files, etc. is limited to the state information that the program provides, and non-code-writing developers often have to wait too long for the developers to get around to adding a new variable.

The Property Manager system provides a single interface for chosen program state information, and allows the creation of new variables dynamically at run-time. The latter capability is especially important for the JSBSim flight control system (FCS) model because the various flight control components (PID controllers, switches, summer, gains, etc.) that make up the control law definition for an aircraft exist only in a specification file. At runtime, after parsing the component definitions, the components are instantiated, and the property manager creates a property to store the output value of each component.

Properties themselves are like global variables with selectively limited visibility (read or read/write) that are categorized into a hierarchical, tree-like structure that is similar to the structure of a Unix file system. The structure of the property tree includes a root node, sub nodes, (like subdirectories) and end-nodes (properties). Similar to a Unix file system, properties can be referenced relative to the current node, or to the root node. Nodes can be grafted onto other nodes similar to symbolically linking files or directories to other files or directories in a file system. Properties are used throughout JSBSim and FlightGear to refer to specific parameters in program code. Properties can be assigned from the command line, from specification files and scripts, and even via a socket interface. Property names look like: position/h-sl-ft, and aero/qbar-psf.

To illustrate the power of using properties and configuration files, consider the case of a high-performance jet aircraft model. Assume for a moment that a new switch has been added to the control panel for the example aircraft that allows the pilot to override pitch limits in the FCS. For

FlightGear, the instrument panel is defined in a configuration file, and the switch is defined there for visual display. A property name is also assigned to the switch definition. Within the flight control portion of the JSBSim aircraft specification file, that same property name assigned to the pitch override switch in the instrument panel definition file can be used to channel the control laws through the desired path as a function of the switch position. No code needs to be touched.

Specific simulation parameters are available both from within JSBSim and in configuration file specifications via properties. As mentioned earlier, “properties” are the term we use to describe parameters that we can access or set from within a configuration file, on the command line, etc.

Many properties are standard properties – i.e. those properties that are always present for all vehicles. The aerodynamic coefficients, engines, thrusters, and flight control/autopilot models will also have dynamically defined properties. This is because the whole set of aerodynamic coefficients, engines, etc. will not be known until after the relevant configuration file for an aircraft is read. One must know the convention used to name the properties for these parameters in order to access them. As an example, the flight control system for the X-15 model features the following components, among others:

```
<flight_control name="X-15">

    <channel name="Pitch">

        <summer name="fcs/pitch-trim-sum">
            <input>fcs/elevator-cmd-norm</input>
            <input>fcs/pitch-trim-cmd-norm</input>
            <clipto>
                <min>-1</min>
                <max>1</max>
            </clipto>
        </summer>

        <aerosurface_scale name="fcs/pitch-command-scale">
            <input>fcs/pitch-trim-sum</input>
            <range>
                <min>-50</min>
                <max>50</max>
            </range>
        </aerosurface_scale>

        <pure_gain name="fcs/pitch-gain-1">
            <input>fcs/pitch-command-scale</input>
            <gain>-0.36</gain>
        </pure_gain>
```

The first component above (“fcs/pitch-trim-sum”) takes input from two places, the known static properties, fcs/elevator-cmd-norm and fcs/pitch-trim-cmd-norm. The next component takes as input the output from the first component. The input property listed for the second component is fcs/pitch-trim-sum. Continuing with the above case shows that the last component, “fcs/pitch-gain-1”, takes as input the output from the preceding component, “fcs/pitch-command-scale”, which is given the property name, fcs/pitch-command-scale.

So, now we have a way to access many parameters inside JSBSim. We know how the FCS is assembled in JSBSim. The same components used in the FCS are also available to build an

autopilot, or other system.

2.5 Math

2.5.1 *Functions*

The function specification in JSBSim is a powerful and versatile resource that allows algebraic functions to be defined in a JSBSim configuration file. The function syntax is similar in concept to MathML (Mathematical Markup Language, www.w3.org/Math/), but it is simpler and more terse.

A function definition consists of an operation, a value, a table, or a property (which evaluates to a value). The currently supported operations are:

- sum (takes n arguments)
- difference (takes n arguments)
- product (takes n arguments)
- quotient (takes 2 arguments)
- pow (takes 2 arguments)
- exp (takes 2 arguments)
- abs (takes n arguments)
- sin (takes 1 arguments)
- cos (takes 1 arguments)
- tan (takes 1 arguments)
- asin (takes 1 arguments)
- acos (takes 1 arguments)
- atan (takes 1 arguments)
- atan2 (takes 2 arguments)
- min (takes n arguments)
- max (takes n arguments)
- avg (takes n arguments)
- fraction (takes 1 argument)
- mod (takes 2 arguments)
- random (Gaussian random number, takes no arguments)
- integer (takes one argument)

An operation is defined in the configuration file as in the following example:

```
<sum>
  <value> 3.14159 </value>
  <property> velocities/qbar </property>
  <product>
    <value> 0.125 </value>
    <property> metrics/wingarea </property>
  </product>
</sum>
```

In the example above, the sum element contains three other items. What gets evaluated is written algebraically as:

```
3.14159 + qbar + (0.125 * wingarea)
```

A full function definition, such as is used in the aerodynamics section of a configuration file includes the function element, and other elements. It should be noted that there can be only one non-optional (non-documentation) element - that is, one operation element - in the top-level function definition. The <function> element cannot have more than one immediate child operation, property, table, or value element. Almost always, the first operation within the function element will be a product or sum. For example:

```
<function name="aero/coefficient/Clr">
    <description>Roll moment due to yaw rate</description>
    <product>
        <property>aero/qbar-area</property>
        <property>metrics/bw-ft</property>
        <property>aero/bi2vel</property>
        <property>velocities/r-aero-rad_sec</property>
        <table>
            <independentVar>aero/alpha-rad</independentVar>
            <tableData>
                 0.000  0.08
                 0.094  0.19
            </tableData>
        </table>
    </product>
</function>
```

The "lowest level" in a function definition is always a value or a property, which cannot itself contain another element. As shown, operations can contain values, properties, tables, or other operations.

Some operations take only a single argument. That argument, however, can be an operation (such as sum) which can contain other items. The point to keep in mind is any such contained operation evaluates to a single value - which is just what the trigonometric functions require (except atan2, which takes two arguments).

Finally, within a function definition, there are some shorthand aliases that can be used for brevity in place of the standard element tags. Properties, values, and tables are normally referred to with the tags, <property>, <value>, and <table>. Within a function definition only, those elements can be referred to with the tags, <p>, <v>, and <t>. Thus, the previous example could be written to look like this:

```
<function name="aero/coefficient/Clr">
  <description>Roll moment due to yaw rate</description>
  <product>
    <p> aero/qbar-area </p>
    <p> metrics/bw-ft </p>
    <p> aero/bi2vel </p>
    <p> velocities/r-aero-rad_sec </p>
    <t>
      <independentVar> aero/alpha-rad </independentVar>
      <tableData>
         0.000  0.08
         0.094  0.19
      </tableData>
    </t>
  </product>
```

```
</function>
```

2.5.2 *Tables*

One, two, or three dimensional lookup tables can be defined in JSBSim for use in aerodynamics and function definitions. For a single "vector" lookup table, the format is as follows:

```
<table name="property_name">
  <independentVar lookup="row"> property_name </independentVar>
  <tableData>
    key_1 value_1
    key_2 value_2
    ...  ...
    key_n value_n
  </tableData>
</table>
```

The lookup="row" attribute in the independentVar element is optional in this case; it is assumed that the independentVar is a row variable. A real example is as shown here:

```
<table>
  <independentVar lookup="row"> aero/alpha-rad </independentVar>
  <tableData>
   -1.57  1.500
   -0.26  0.033
    0.00  0.025
    0.26  0.033
    1.57  1.500
  </tableData>
</table>
```

The first column in the data table represents the lookup index (or *breakpoints*, or *keys*). In this case, the lookup index is aero/alpha-rad (angle of attack in radians). If alpha is 0.26 radians, the value returned from the lookup table would be 0.033.

The definition for a 2D table, is as follows:

```
<table name="property_name">
  <independentVar lookup="row"> property_name </independentVar>
  <independentVar lookup="column"> property_name </independentVar>
  <tableData>
                {col_1_key   col_2_key   ...  col_n_key }
    {row_1_key} {col_1_data  col_2_data  ...  col_n_data}
    {row_2_key} {...         ...         ...  ...       }
    { ...     } {...         ...         ...  ...       }
    {row_n_key} {...         ...         ...  ...       }
  </tableData>
</table>
```

The data is in a gridded format. A real example is as shown below. Alpha in radians is the row lookup (alpha breakpoints are arranged in the first column) and flap position in degrees is split up in columns for deflections of 0, 10, 20, and 30 degrees):

```
<table>
  <independentVar lookup="row">aero/alpha-rad</independentVar>
  <independentVar lookup="column">fcs/flap-pos-deg</independentVar>
  <tableData>
```

```
                0.0         10.0        20.0         30.0
    -0.0523599  8.96747e-05 0.00231942  0.0059252    0.00835082
    -0.0349066  0.000313268 0.00567451  0.0108461    0.0140545
    -0.0174533  0.00201318  0.0105059   0.0172432    0.0212346
     0.0        0.0051894   0.0168137   0.0251167    0.0298909
     0.0174533  0.00993967  0.0247521   0.0346492    0.0402205
     0.0349066  0.0162201   0.0342207   0.0457119    0.0520802
     0.0523599  0.0240308   0.0452195   0.0583047    0.0654701
     0.0698132  0.0333717   0.0577485   0.0724278    0.0803902
     0.0872664  0.0442427   0.0718077   0.088081     0.0968405
  </tableData>
</table>
```

The definition for a 3D table in a coefficient would be (for example):

```
<table name="property_name">
  <independentVar lookup="row"> property_name </independentVar>
  <independentVar lookup="column"> property_name </independentVar>
  <independentVar lookup="table"> property_name </independentVar>
  <tableData breakpoint="table_1_key">
                {col_1_key   col_2_key   ...  col_n_key }
    {row_1_key} {col_1_data  col_2_data  ...  col_n_data}
    {row_2_key} {...         ...         ...  ...       }
    { ...     } {...         ...         ...  ...       }
    {row_n_key} {...         ...         ...  ...       }
  </tableData>
  <tableData breakpoint="table_2_key">
                {col_1_key   col_2_key   ...  col_n_key }
    {row_1_key} {col_1_data  col_2_data  ...  col_n_data}
    {row_2_key} {...         ...         ...  ...       }
    { ...     } {...         ...         ...  ...       }
    {row_n_key} {...         ...         ...  ...       }
  </tableData>
  ...
  <tableData breakpoint="table_n_key">
                {col_1_key   col_2_key   ...  col_n_key }
    {row_1_key} {col_1_data  col_2_data  ...  col_n_data}
    {row_2_key} {...         ...         ...  ...       }
    { ...     } {...         ...         ...  ...       }
    {row_n_key} {...         ...         ...  ...       }
  </tableData>
</table>
```

[Note the "breakpoint" attribute in the tableData element, above.] Here's an example:

```
<table>
  <independentVar lookup="row">fcs/row-value</independentVar>
  <independentVar lookup="column">fcs/column-value</independentVar>
  <independentVar lookup="table">fcs/table-value</independentVar>
  <tableData breakPoint="-1.0">
           -1.0     1.0
    0.0     1.0000  2.0000
    1.0     3.0000  4.0000
  </tableData>
  <tableData breakPoint="0.0000">
            0.0     10.0
    2.0     1.0000  2.0000
```

```
    3.0     3.0000  4.0000
  </tableData>
  <tableData breakPoint="1.0">
           0.0     10.0     20.0
     2.0   1.0000   2.0000   3.0000
     3.0   4.0000   5.0000   6.0000
    10.0   7.0000   8.0000   9.0000
  </tableData>
</table>
```

Note that table values are interpolated linearly, and no extrapolation is done at the table limits – the highest value a table will return is the highest value that is defined.

2.6 Forces and Moments

2.6.1 *Aerodynamics*

There are several ways to model the aerodynamic forces and moments (torques) that act on an aircraft. JSBSim started out by using the coefficient buildup method. In the coefficient buildup method, the lift force (for instance) is determined by summing all of the contributions to lift. The contributions differ depending on the aircraft and the fidelity of the model, but contributions to lift can include those from:

- Wing
- Elevator
- Flaps

Aerodynamic coefficients are numbers which, when multiplied by certain other values (such as dynamic pressure and wing area), result in a force or moment. The coefficients can be taken from flight test reports or textbooks, or they can be calculated using software (such as Digital DATCOM or other commercially available programs) or by hand calculations.

Eventually, JSBSim added support for aerodynamic properties specified as functions. Within the <aerodynamics> section of a configuration file there are six subsections representing the 3 force and 3 moment axes (for a total of six degrees of freedom). The basic layout of the aerodynamics section is as follows:

```
<aerodynamics>

  <axis name="DRAG">
    { force contributions …}
  </axis>

  <axis name="SIDE">
    { force contributions …}
  </axis>

  <axis name="LIFT">
    { force contributions …}
  </axis>

  <axis name="ROLL">
    { moment contributions …}
  </axis>
```

```
  <axis name="PITCH">
    { moment contributions …}
  </axis>

  <axis name="YAW">
    { moment contributions …}
  </axis>

</aerodynamics>
```

Individual axes are not all absolutely required. There are several standard grouped sets of axes that are supported in JSBSim:

- DRAG, SIDE, LIFT (wind axes)
- X, Y, Z (body axes)
- AXIAL, SIDE, NORMAL (body axes)

All three systems accept ROLL, PITCH, YAW axis definitions. The axial systems cannot be mixed.

Within the axis elements, functions are used to define individual force or moment contributions to the total for that axis. Functions are used throughout JSBSim. In defining a force or moment, functions can employ the use of tables, constants, trigonometric functions, or other standard C library functions. Simulation parameters are referenced via properties. Here is an example:

```
<function name="aero/force/CLDf">
  <description>Lift contribution due to flap deflection</description>
  <product>
    <property>aero/function/ground-effect-factor-lift</property>
    <property>aero/qbar-area</property>
    <table>
      <independentVar>fcs/flap-pos-deg</independentVar>
      <tableData>
         0.0  0.0
        10.0  0.20
        20.0  0.30
        30.0  0.35
      </tableData>
    </table>
  </product>
</function>
```

In this case, a description in words of what the above does is as follows: CLDf (the contribution of lift due to flap deflection) is the product of the ground-effect-factor-lift, qbar-area, and the value determined by the table, which is indexed as a function of flap position in degrees.

All of the functions in an <axis> section are summed and applied to the aircraft in the appropriate manner. There is some flexibility in this format, though. Functions that are specified outside of any <axis> section are created and calculated, but they do not specifically contribute to any force or moment total by themselves. However, they can be referenced by other functions that *are* in an <axis> section. This technique allows calculations that might be applied to several individual functions to be performed once and used several times. The technique can be taken even further, with actual aerodynamic coefficients being calculated outside of an <axis> definition, with the coefficients subsequently being multiplied within function definitions by the various

factors (properties) that turn them into forces and moments *inside* an <axis> definition.

As an example, let's examine the lift force due to angle of attack (alpha). We know that increasing the angle of attack increases lift – up to a point. Lift force is traditionally defined as the product of dynamic pressure (qbar), wing area (S_w), and lift coefficient (C_L). In this case, the lift coefficient is determined via a lookup table, using alpha as an index into the table:

```
<function name="aero/force/CLalpha">
  <description> Lift due to alpha </description>
  <product>
    <property> aero/qbar-psf </property>
    <property> metrics/Sw-sqft </property>
    <table name="CL">
      <independentVar lookup="row"> aero/alpha-rad </independentVar>
      <tableData>
        -0.20 -0.720
         0.00  0.240
         0.22  1.300
         0.60  0.664
      </tableData>
    </table>
  </product>
</function>
```

The above function results in "CLalpha" being calculated many times per second as JSBSim executes. The value of the function is the product of qbar, wing area, and lift coefficient as determined by the lookup table. For instance, if alpha (in radians) is 0 degrees, the lift coefficient is 0.240.

So, for the aircraft we are modeling, where do we get this information about the lift coefficient – or any other aerodynamic data? An early task in modeling any aircraft is to collect the aircraft characteristics information – a sort of "data farming" effort. One can begin with a search using the NASA Technical Report Server search engine at: http://ntrs.nasa.gov. For instance, a search phrase of "Beech 99" returned several documents, and one was available as a PDF file:

> **Stability and control derivative estimates obtained from flight data for the Beech 99 aircraft**, *Tanner, R. R.; Montgomery, T. D.*
> *Lateral-directional and longitudinal stability and control derivatives were determined from flight data by using a maximum likelihood estimator for the Beech 99 airplane. Data were obtained with the aircraft in the cruise configuration and with one-third flap deflection. The estimated derivatives show good agreement with the predictions of the manufacturer.*

Next, one might search for an FAA Type Certificate Data Sheet (TCDS) at:
http://www.airweb.faa.gov/Regulatory_and_Guidance_Library/rgMakeModel.nsf/
A careful search led to "TYPE CERTIFICATE DATA SHEET NO. A14CE" for the Beech 99 aircraft.

Documents such as these can be helpful in crafting an aircraft aerodynamic model. If you are unable to find stability and aerodynamic information about the aircraft you are interested in, try finding a comparable aircraft. For instance, the CV-880, B747, and C-5 all have data published in *Aircraft Handling Qualities Data*, by Heffley and Jewell.

First of all, the NASA *Beech 99* Technical Memo mentioned above contains specific

information about the aerodynamic qualities of the aircraft. Let's look at how to begin creating a JSBSim aerodynamics specification for a specific aircraft.

As mentioned, JSBSim uses the coefficient buildup method for calculating aerodynamic forces and moments on an aircraft. We have a nice report for the Beech 99 showing both the manufacturer's estimates and flight test data for the aerodynamic quantities:

Longitudinal coefficients (i.e. xz-plane symmetric coefficients, lift, pitch, drag)

(1) $C_N = C_{N_\alpha}\alpha + C_{N_{\dot\alpha}}\dot\alpha + C_{N_{\delta_e}}\delta_e + C_{N_0}$

(2) $C_m = C_{m_\alpha}\alpha + C_{m_q}\frac{q\bar{c}}{2V} + C_{m_{\dot\alpha}}\dot\alpha + C_{m_{\delta_e}}\delta_e + C_{m_0}$

(3) $C_D = C_{D_0} + KC_L^{\,2}$

Lateral coefficients (i.e. side force, roll, yaw)

(4) $C_Y = C_{Y_\beta}\beta + C_{Y_{\delta_a}}\delta_a + C_{Y_{\delta_r}}\delta_r + C_{Y_0}$

(5) $C_l = C_{l_\beta}\beta + C_{l_p}\frac{pb}{2V} + C_{l_r}\frac{rb}{2V} + C_{l_{\delta_a}}\delta_a + C_{l_{\delta_r}}\delta_r + C_{l_0}$

(6) $C_n = C_{n_\beta}\beta + C_{n_p}\frac{pb}{2V} + C_{n_r}\frac{rb}{2V} + C_{n_{\delta_a}}\delta_a + C_{n_{\delta_r}}\delta_r + C_{n_0}$

The above equations are for (in order), normal force coefficient, pitch moment coefficient, drag coefficient, side force coefficient, rolling moment coefficient, and yaw moment coefficient. There is no lift coefficient specified. For low alpha, the normal and lift coefficients are very close.

The manufacturer's estimates for the aerodynamic force and moment components are as follows:

(7) $C_{L_\alpha} = 0.096 + 0.151 \cdot T_c$ (per degree; no flaps)

(8) $C_{L_\alpha} = 0.101 + 0.151 \cdot T_c$ (per degree; 1/3 flaps)

(9) $C_D = 0.0275 + 0.0625 \cdot C_L^{\,2}$

(10) $C_{m_{\alpha,1/4}} = 0.010 + 0.010 \cdot T_c$ (per degree)

(11) $C_{m_{\delta_e}} = -0.034 - 0.032 \cdot T_c$ (per degree)

(12) $C_{m_{\dot\alpha}} = -43.1$ (per *radian*)

(13) $C_{m_q} = -3.40$ (per *radian*)

(14) $C_{n_\beta} = 0.014 - 0.0015 \cdot T_c$ (per degree)

(15) $C_{l_\beta} = -0.0023$ (per degree)

(16) $C_{Y_\beta} = -0.010$ (per degree)

(17) $C_{n_{\delta_r}} = -0.0014$ (per degree)

(18) $C_{l\delta_r} = 0.00017 - 0.000022 \cdot \alpha$ (per degree)

(19) $C_{Y\delta_r} = 0.0026$ (per *radian*)

(20) $C_{n_r} = -0.20 - 0.197 \cdot \alpha$ (per *radian*)

(21) $C_{l_r} = 0.05 + 0.623 \cdot \alpha$ (per *radian*)

(22) $C_{Y_r} = 0.394$ (per *radian*)

(23) $C_{n_p} = 0.0079 - 0.762 \cdot \alpha$ (per *radian*)

(24) $C_{l_p} = -0.515$ (per *radian*)

(25) $C_{Y_p} = -0.199 - 0.385 \cdot \alpha$ (per *radian*)

(26) $C_{l\delta_a} = 0.0027$ (per degree)

(27) $C_{n\delta_a} = -0.0015$ (per degree)

The subscripts *p*, *q*, *r*, refer to rates about the *x*, *y*, *z*, axes; δ_x (where *x* is *a*, *r*, *e*) refers to the deflection of the aileron, rudder, or elevator, respectively; *l*, *m*, *n*, refer to moments about the *x*, *y*, *z*, axes; *α*, *β*, refer to angle of attack and angle of sideslip; *L*, *D*, and *Y*, refer to lift, drag, and side force. T_c refers to thrust coefficient, and is defined as follows:

(28) $$T_c = \frac{Thrust}{\bar{q}S}$$

These coefficients and derivatives are not all necessarily the last word on aerodynamic modeling for the Beech 99 – the data comes from testing that was done for a particular purpose that is not necessarily supposed to result in a full aerodynamic database for simulation modeling. So, we are left with data obtained within a fairly narrow flight envelope. We can look at the coefficient and derivative data and make adjustments on a one-by-one basis. But, not only are there adjustments that can be made to the set of coefficients, but additional ones that can be added in.

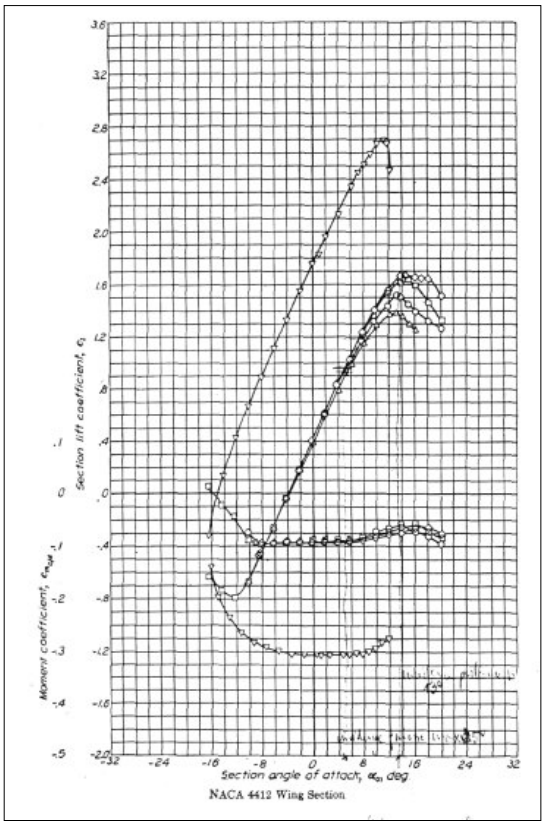


Figure 3. NACA 4412 wing section characteristics

Let's consider $C_{L\alpha}$ first. The symbol $C_{L\alpha}$ is shorthand for $\delta C_L/\delta_\alpha$, or *the change in lift coefficient with a change in alpha*. This is just the slope of the lift coefficient curve (see Figure 3 for an example). The lift curve slope for an ideal 2D airfoil section is 2π (per radian). For a cambered airfoil, the lift curve does *not* pass through zero – that is, there is an inherent lift to the airfoil even at zero angle of attack. Furthermore, a real wing (i.e. *not* a 2D airfoil section) is not as efficient as

an airfoil, and the slope of the lift curve will be somewhat less than 2π. Additionally, a real wing will eventually stall at a particular angle of attack, perhaps around 12-16 degrees. The information given in the Beech 99 report is valuable; however, some adjustments will have to be made to account for stalling. The value of $C_{L\alpha}$ defines a slope – a line that goes on forever! Knowing that the lift attributable to alpha is complex – it's not a straight line – we might instead consider creating a lookup table or function that includes the behavior at stall. Unfortunately, little or no information is given to us about stall in the document, but some searching suggests 73 kts (123 ft/sec) as the stall speed. Also, what about the post-stall behavior? Modeling post-stall and very high alpha regimes is effectively beyond the scope of this discussion, but suffice it so say we would like to provide *plausible* behavior in all regimes where possible. In this case, we can look to a study done by Sandia Labs some years ago. The study resulted in data being collected that showed the lift coefficient as it varied with angle of attack for symmetric airfoils of varying thickness, across the entire alpha range of 0 to 180 degrees. Figure 4 shows the lift coefficient for an airfoil for a full range of alpha. The thickness of the airfoil represented by the data in Figure 4 is 12%. The blended-airfoil wing used on the Beech 99 is 18% thick at the root and 12% thick at the tip. We will assert that we can use the data for a 15% thick airfoil as representative of the average characteristic of the wing.

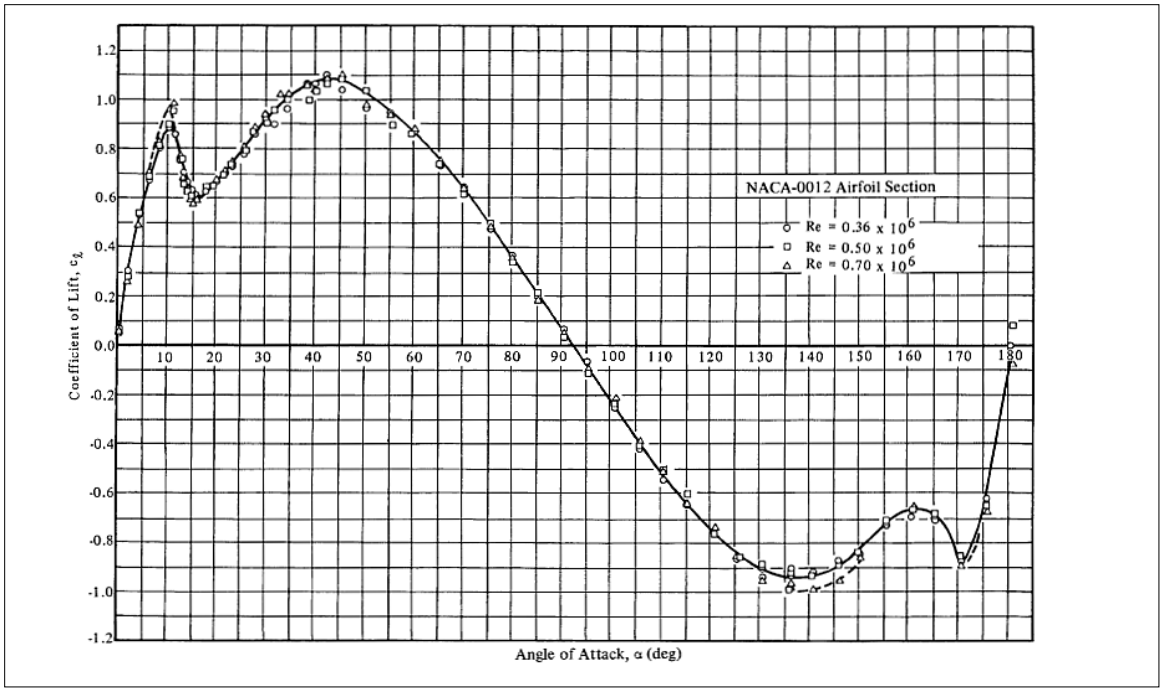


Figure 4. Full, 180° lift coefficient data.

Not including thrust effects at the moment, the lift coefficient of the wing can be written as:

```
Lift coefficient = 0.939 * {table value} + (0.25 * Df * 0.017) + 0.3
```

The lift force is just the above coefficient multiplied by qbar and area:

```
Lift force = qbar * wingarea * Lift_coefficient
```

The equation was developed by asserting that:

- the effect of camber on the airfoil data effectively adds 0.3 to the lift coefficient,
- for a 3-D wing the curve must be scaled (multiplied by 0.939),
- adding a flap setting in degrees (multiplied by 0.017 to convert it to a radian measurement) multiplied by 0.25 gives the increase in lift coefficient from flaps

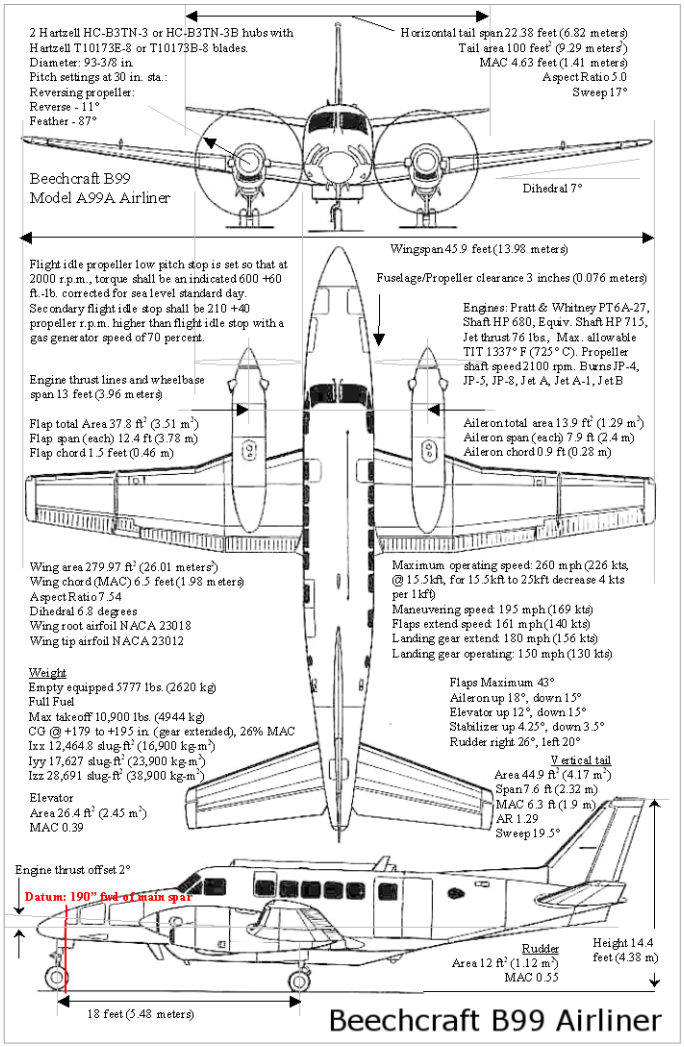
While this approach seems valid enough for modeling plausible wing lift characteristics, this exercise is meant more to show the kinds of things that the JSBSim <function> elements can be constructed to do. The JSBSim-ML representation of the lift of a wing using the 15% thick Sandia data and information from “Theory of Wing Sections”, NACA wing data, etc. is shown as:

```
<function name="aero/force/CLalpha">
  <description> Lift force due to alpha </description>
  <product>
    <property> aero/qbar-psf </property>
    <property> metrics/Sw-sqft </property>
    <sum name=”aero/coefficient/CLalpha”>
      <value> 0.3 </value> <!-- Lift curve slope camber shift -->
      <product>
        <value> 0.25 </value> <!-- flap setting curve shift -->
        <property> fcs/flap-pos-deg </property>
        <value> 0.017 </value> <!-- convert degrees to radians -->
      </product>
      <product>
        <value> 0.939583333 </value> <!-- Lift curve scaling value -->
        <table name=”aero/coefficient/Clalpha”>
          <independentVar lookup="row">aero/alpha-deg</independentVar>
          <tableData>
          -180   0.0000
          -170   0.8500
          -160   0.6350
          -150   0.7700
          -140   0.9800
          -130   0.8500
           ...   ...    <!-- intermediate data removed for brevity -->
           120  -0.6700
           130  -0.8500
           140  -0.9800
           150  -0.7700
           160  -0.6350
           170  -0.8500
           180   0.0000
          </tableData>
        </table>
      </product>
    </sum>
  </product>
</function>
```

This specification should produce a fairly smoothly changing lift force in a wide range of conditions. Note that this is a force calculation – the lift coefficient has been multiplied with qbar and the area of the wing.

This function defining the lift force on the wing due to angle of attack is but one contribution to the total lift force on the aircraft. When the elevator is moved, it also changes the lift force of the tail, which results in a pitch moment on the aircraft – the main purpose of the elevator – but the change in lift force must be taken into account in the total lift force calculation.

Developing an accurate aerodynamic model of an aircraft is an art form in itself, and crafting more accurate and detailed aerodynamic models can involve great expense and effort using wind tunnels and computational facilities. But the first step is to gather as much information as is possible for the vehicle being modeled.



2.6.2 *Propulsion*

Several different engine types are modeled in JSBSim,

- Piston
- Rocket
- Turbine
- Turboprop
- Electric

They are generic models, because the characteristics that define a specific engine must be entered in an engine configuration file. Any number of engines (the same or different) can be included in an aircraft configuration file by referring to the name of the engine file, and specifying placement information in the aircraft configuration file in the <engine> element. At runtime, the forces and moments generated by each engine are calculated and summed together.

2.6.3 *Ground reactions*

2.6.4 *External reactions*

2.7 Flight Control and Systems modeling

2.7.1 *System components*

Flight control laws, stability augmentation systems, autopilots, and arbitrary aircraft systems

(avionics, electrical systems, etc.) can be modeled in JSBSim by creating chains of individual control components. A suite of configurable components is available in JSBSim that includes gains, filters, switches, etc. An aircraft system is specified as a string of components within the <channel> element of a <system>, <autopilot>, or <flight_control> specification in the aircraft configuration file. Groupings of components which perform a related task are placed in a <channel> element. For example,

```
<autopilot name="C172X Autopilot">

  <!-- Wing leveler -->

  <channel name="Roll wing leveler">

    <pid name="fcs/roll-ap-error-pid">
      <input>attitude/phi-rad</input>
      <kp> 1.0 </kp>
      <ki> 0.01</ki>
      <kd> 0.1 </kd>
    </pid>

    <switch name="fcs/roll-ap-autoswitch">
      <default value="0.0"/>
      <test value="fcs/roll-ap-error-pid">
        ap/attitude_hold == 1
      </test>
    </switch>

    <pure_gain name="fcs/roll-ap-aileron-command-normalizer">
       <input>fcs/roll-ap-autoswitch</input>
       <gain>-1</gain>
    </pure_gain>

  </channel>

  <channel name="Pitch attitude hold">
    … components …
  </channel>

  … additional channels …

</autopilot>
```

A component will calculate its own output, and that value will be passed to another component later on.

Historically, the flight control section was the first section to be implemented in JSBSim. Later, the autopilot section was added, and then the system section, to support any arbitrary system. Note that it is possible that the system section may replace the autopilot and flight control sections. There can be any number of system sections. Also, any of the three sections may include a reference to a filename, loading the system definition from a file rather than defining the system inline by specifying a value for the file attribute. For example:

```
<autopilot file="C310ap"/>
```

The file "C310ap.xml" will be searched for in the same directory as the aircraft file is found in.

Common *system* files are searched for in the systems/ directory.

In the example above, the autopilot must be completely defined for the aircraft (C310) that is referring to it. That doesn't really support reuse, because the PID gains (for instance) used in the autopilot may not be the same for other aircraft that might wish to use the C310ap file.

However, we can also specify the PID gains to use in the <autopilot> element itself, or override default gains specified in the autopilot file (in this example, C310ap.xml). We do this as follows:

```
  <autopilot file="c310ap">

    <!-- Roll channel A/P gains -->

    <property value="50.0"> ap/roll-pid-kp </property>
    <property value="5.0"> ap/roll-pid-ki </property>
    <property value="17.0"> ap/roll-pid-kd </property>

  </autopilot>
```

For reference, here is the relevant specification in the C310ap.xml file:

```
<autopilot name="C-310 Autopilot">

<!-- INTERFACE PROPERTIES -->

  <property>ap/attitude_hold</property>
  <property>ap/altitude_hold</property>
  ...

  <property value="50.0"> ap/roll-pid-kp </property>
  <property value="5.0"> ap/roll-pid-ki </property>
  <property value="17.0"> ap/roll-pid-kd </property>

<!-- Wing leveler -->

<channel name="AP roll wing leveler">

  ...

  <pid name="fcs/roll-ap-error-pid">
    <input>attitude/phi-rad</input>
    <kp> ap/roll-pid-kp </kp>
    <ki> ap/roll-pid-ki </ki>
    <kd> ap/roll-pid-kd </kd>
    <trigger> fcs/wing-leveler-ap-on-off </trigger>
  </pid>

  ...
```

This opens up the ability to create standard, generic autopilot and system files to refer to and then simply override gains and values in a specific aircraft config file.

Each of the system components is described in Section, 4.1.9.

2.7.2 *Automatic Flight in JSBSim*

One long-time goal of JSBSim has been to support automatic, scripted flights. Scripted flights

refer to the ability of JSBSim to run in a standalone mode (apart from visuals) and fly in a stable manner to various targets, be they altitude and heading, or latitude and longitude, etc. This is a useful feature for many reasons, among them being regression testing of JSBSim, aircraft flight model performance testing, and control system development.

Some of the features that make capability possible include the incorporation of switch and function components, sensors, and autopilot-related properties.

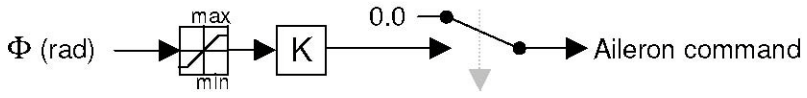
In implementing automatic flight in JSBSim, several files are involved:

- A script file directs the aircraft to turn on its engine, advance the throttle, and fly to a target heading, altitude, and/or velocity. The script file and processing capability takes the role of guidance.
- The aircraft configuration file defines the aircraft properties – including the flight control system, and the interface to the autopilot (or it could even include the autopilot itself).
- The autopilot definition file (if separate).

The job of an autopilot is to attain a state such as wings level, or a heading, or an altitude, and hold it. Designing an autopilot for an aircraft is a science in itself – we will only gloss over the design aspect, paying the most attention on how to implement one and use it in JSBSim.

As an example, we can set out to build a wing-leveler autopilot. Wings-level is by definition a roll angle of zero (phi=0). Many forces will tend to disrupt a state of wings-level, such as engine torque, atmospheric turbulence, fuel slosh, etc. To get to a phi of zero (assuming phi is initially non-zero) we will need to attain a non-zero phi *rate*, which drives us towards wings-level. To get the phi rate, we will need to attain a phi *acceleration*, which is controlled by aileron deflection.

One possibility is to simply command the ailerons based on the roll angle. This is called proportional control, because the output is simply the input multiplied by a value – the output is proportional to the input. The autopilot aileron command is sent to the main flight control system and summed in to the aileron command channel. There are a couple of nuances to this autopilot arrangement, however. For instance, it is always on. A better arrangement is shown on the next page.



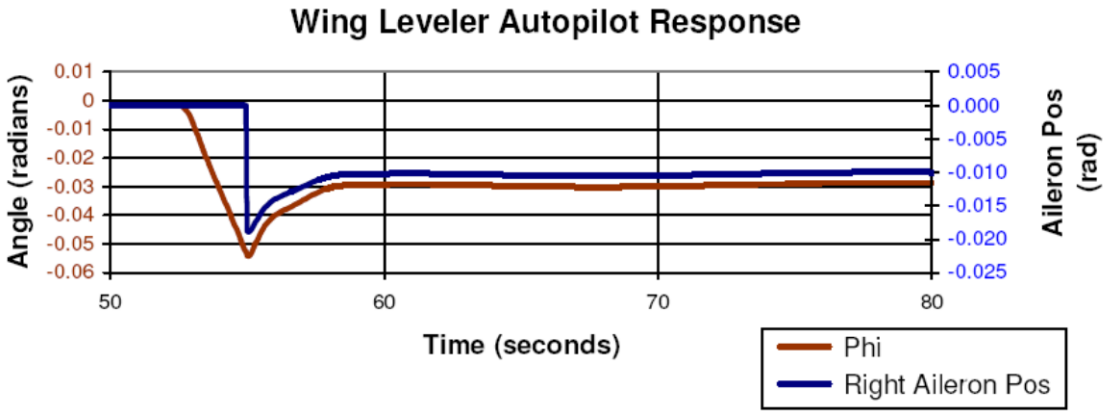
Roll A/P ON

Also, the min, max, and K values must be chosen properly. As an example, we set up JSBSim to run with this autopilot and see what the response is. Here is the autopilot we use, in JSBSim format:

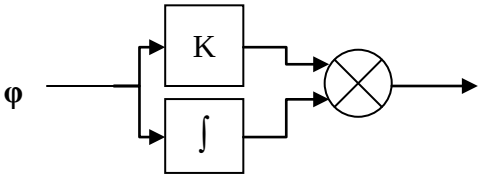
```
<channel name="AP Roll Wing Leveler">

  <pure_gain name="ap/roll-ap-wing-leveler">
    <input> attitude/phi-rad </input>
    <gain>2.0</gain>
    <clipto>
      <min>-0.255</min>
      <max>0.255</max>
    </clipto>
```

```
  </pure_gain>

  <switch name="ap/roll-ap-autoswitch">
    <default value="0.0"/>
    <test logic="AND" value="ap/roll-ap-wing-leveler">
      ap/attitude_hold == 1
    </test>
  </switch>

  <pure_gain name="ap/roll-ap-aileron-command-normalizer">
     <input>ap/roll-ap-autoswitch</input>
     <gain>-1</gain>
  </pure_gain>
</channel>
```

There are a couple of points to make about the above listing. First, for the *ap/roll-ap-wing-leveler* component, there could be a "<gain> 1 </gain>" line in the definition, however, gain is 1 by default for the pure gain component. Second, the output of the control is limited to ± 0.255 radians (15 degrees). This is about all the ailerons can produce anyhow. The *ap/attitude_hold* property in the ap/r*oll-ap-autoswitch* is functionally the "Roll A/P ON" switch shown in the previous diagram.



It is seen in the graph that once the transient damps out there is a bias in the roll angle (phi). That bias can be removed using an integrator:

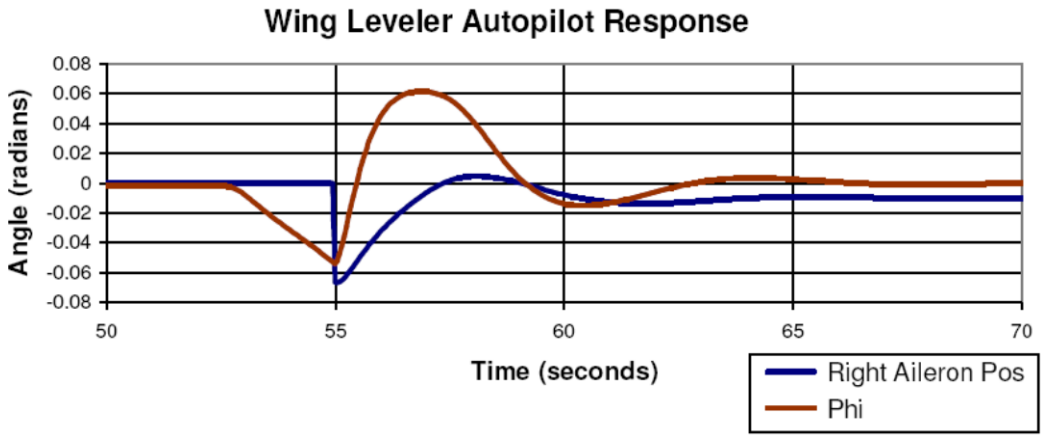


This block diagram of a wing leveler using PI control is implemented in JSBSim-ML as follows:

```
<channel name="AP Roll Wing Leveler">

  <pure_gain name="ap/limited-phi">
    <input> attitude/phi-rad </input>
    <clipto>
      <min>-0.255</min>
      <max>0.255</max>
```

```
    </clipto>
  </pure_gain>

  <pure_gain name="ap/roll-ap-wing-leveler">
    <input> ap/limited-phi </input>
    <gain>2.0</gain>
  </pure_gain>

  <integrator name="ap/roll-ap-error-integrator">
    <input> ap/limited-phi </input>
    <c1> 0.125 </c1>
  </integrator>

  <summer name="ap/roll-ap-error-summer">
     <input> ap/roll-ap-wing-leveler</input>
     <input> ap/roll-ap-error-integrator</input>
     <clipto>
       <min>-1.0</min>
       <max> 1.0</max>
     </clipto>
  </summer>

  <switch name="ap/roll-ap-autoswitch">
    <default value="0.0"/>
    <test logic="AND" value="ap/roll-ap-error-summer">
      ap/attitude_hold == 1
    </test>
  </switch>

  <pure_gain name="ap/roll-ap-aileron-command-normalizer">
     <input>ap/roll-ap-autoswitch</input>
     <gain>-1</gain>
  </pure_gain>
</channel>
```

Making the same test run as before gives the following graph of phi and aileron command against time:

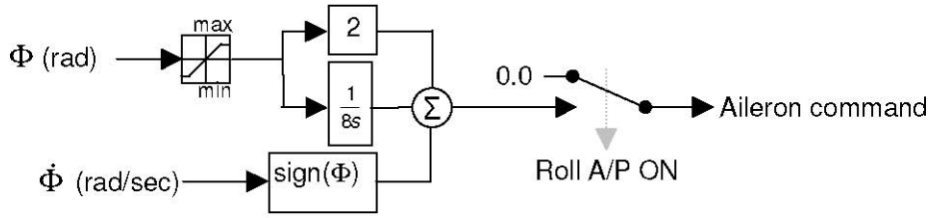


The roll angle is seen in the above graph to settle out, now, to wings level. There is, however, a good degree of overshoot initially. This is where the tweaking comes into play. One can vary the relative contributions of the proportional and integral parts of the control command to achieve

different results. However, in turbulence, for instance, the response may be different than what is expected, perhaps even unstable. Often, it is expected that the autopilot will be turned off in bad weather.

There is yet another kind of control action we can use, called derivative control. Derivative control action produces a control command that is proportional to the rate of change of the error. In our wing leveler case – where we seek zero roll angle – the rate of change of the error is simply the roll rate, "p". If that parameter is summed in, the resulting controller is a PID controller (Proportional-Integral-Derivative).

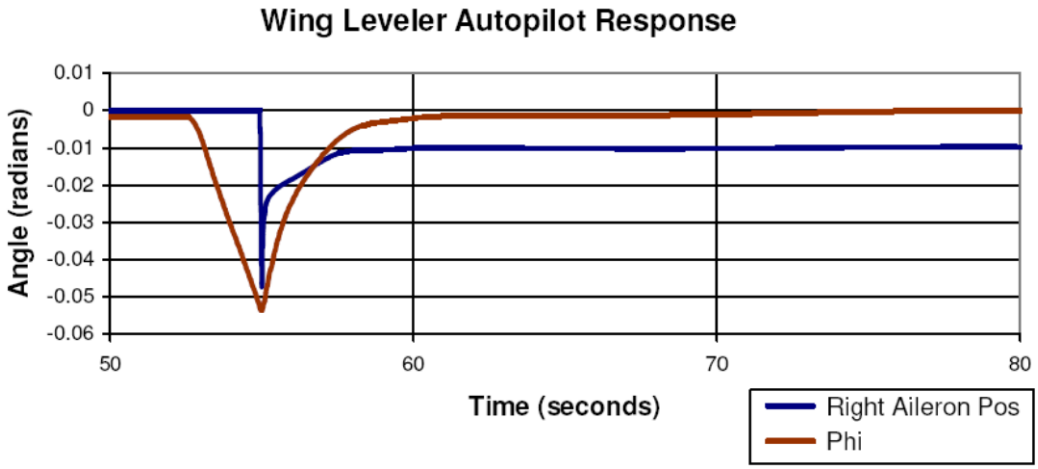
If we play around with the gains we can tailor the wing leveler to give the best response. The final control system block diagram for the wing leveler is:



This is represented in JSBSim as follows:

```
<channel name="AP Roll Wing Leveler">

  <pure_gain name="ap/limited-phi">
    <input> attitude/phi-rad </input>
    <clipto>
      <min> -0.255 </min>
      <max>  0.255 </max>
    </clipto>
  </pure_gain>

  <pure_gain name="ap/roll-ap-wing-leveler">
    <input> ap/limited-phi </input>
    <gain> 2.0 </gain>
  </pure_gain>

  <integrator name="ap/roll-ap-error-integrator">
    <input> ap/limited-phi </input>
    <c1> 0.125 </c1>
  </integrator>

  <summer name="ap/roll-ap-error-summer">
     <input> velocities/p-rad_sec</input>
     <input> ap/roll-ap-wing-leveler</input>
     <input> ap/roll-ap-error-integrator</input>
     <clipto>
       <min> -1.0 </min>
       <max>  1.0 </max>
     </clipto>
  </summer>

  <switch name="ap/roll-ap-autoswitch">
    <default value="0.0"/>
    <test logic="AND" value="ap/roll-ap-error-summer">
      ap/attitude_hold == 1
```

```
    </test>
  </switch>

  <pure_gain name="ap/roll-ap-aileron-command-normalizer">
     <input> ap/roll-ap-autoswitch </input>
     <gain> -1 </gain>
  </pure_gain>
</channel>
```



We see now that our wing leveler controller is performing as desired.

3. Authoring Configuration Files

3.1 Aircraft

3.1.1 *File Header Information*

The fileheader section of the configuration file includes information about who made the aircraft model, when, which version it is, what the license is, which references were used in creating the model, notes, and limitations. For example:

```
<fileheader>
  <author> Joe Public </author>
  <email> joe.public@hotmail.com </email>
  <organization>
    Department of Mechanical Engineering,
    University of Bath,
    UK
  </organization>
  <filecreationdate> 2003-01-01 </filecreationdate>
  <version> 1.0 </version>
  <description> Models a 1970 B747-100 with Pratt & Whitney JT9D-3
                turbofan engines.
  </description>
  <note> Since a moving stabilizer property is not available the
         speedbrake property is being used to emulate the Boeing
         747 trimmable horizontal stabilizer. </note>
  <note> Aircraft origin for measurements is the nose.</note>
  <limitation>
    Undercarriage Aerodynamic effects not modeled
  </limitation>
  <limitation> Undercarriage data approximated </limitation>
  <limitation> Spoilers not modeled </limitation>
  <reference
    refID="None"
    author="Hanke, C.R."
    title="The Simulation of a Large Jet Transport Aircraft, Vol I."
    date="1970"/>
  <reference
    refID="None"
    author="Hanke, C.R."
    title="The Simulation of a Large Jet Transport Aircraft, Vol. II"
    date="1970"/>
  <reference
    refID="None"
    author="Roskam, J."
    title="Airplane Flight Dynamics & Automatic Flight Control, Vol. I"
    date="1979"/>
  <reference
    refID="None"
    author="Jane's"
```

```
    title="Jane's All The World's Aircraft 1980-1981"
    date="1981"/>
</fileheader>
```

3.1.2 *Metrics*

The metrics section of the configuration file defines the characteristic measurements of the vehicle and the locations of key points. For example:

```
<wingarea unit="FT2"> 174.0 </wingarea>
<wingspan unit="FT"> 35.8 </wingspan>
<chord unit="FT"> 4.9 </chord>
<htailarea unit="FT2"> 21.9 </htailarea>
<htailarm unit="FT"> 15.7 </htailarm>
<vtailarea unit="FT2"> 16.5 </vtailarea>
<vtailarm unit="FT"> 15.7 </vtailarm>
<location name="AERORP" unit="IN">
  <x> 43.2 </x>
  <y> 0.0 </y>
  <z> 59.4 </z>
</location>
<location name="EYEPOINT" unit="IN">
  <x> 37.0 </x>
  <y> 0.0 </y>
  <z> 48.0 </z>
</location>
<location name="VRP" unit="IN">
  <x> 42.6 </x>
  <y> 0.0 </y>
  <z> 38.5 </z>
</location>
```

The Vehicle Reference Point (VRP) is not by itself important to flight dynamics. If you are using JSBSim within a larger simulation framework that includes a visual system, then it matters. When placing a 3D model in the scene, you might ask yourself the question: given that my 3D model can have it's origin at any arbitrary spot, how do I make sure I am placing the aircraft exactly where the flight model specifies that it should be? When you are flying and are at high altitude, the question might not seem all that critical. However, when you are on the ground, if model placement is not correct, then the model may be seen to "hover", or to be embedded in the ground. Or, when you rotate at takeoff, you may rotate into the ground.

So, correct placement is important.

The flight model calculates motion about the aircraft CG, and reports the current position of the CG. The CG might be thought of as a good "origin" for the 3D model, but the CG moves as fuel burns off.

The VRP is simply an agreed upon point on the aircraft, for which the flight model will provide the latitude/longitude/altitude. The VRP is defined in JSBSim in the same structural frame in which the landing gear, empty weight CG, etc. are defined. By convention, the nose of the aircraft is usually taken to be the point to be reported as the VRP.

If you are using JSBSim as a standalone application, you do not need to define the VRP.

3.1.3 *Mass/Balance*

Within this section the mass properties of the aircraft are specified. Included are the empty

weight of the craft, the moments and products of inertia, the location of the center of gravity, and definitions for any point masses that are included. Here's an example:

```
<mass_balance>
    <documentation>
       The Center of Gravity location, empty weight, in aircraft's own
       structural coord system.
    </documentation>
    <ixx unit="SLUG*FT2"> 2.31442e+06 </ixx>
    <iyy unit="SLUG*FT2"> 1.34649e+06 </iyy>
    <izz unit="SLUG*FT2"> 3.59608e+06 </izz>
    <ixy unit="SLUG*FT2"> 0 </ixy>
    <emptywt unit="LBS"> 48400 </emptywt>
    <location name="CG" unit="IN">
        <x> 459.2 </x>
        <y> 0 </y>
        <z> -31.8 </z>
    </location>
    <pointmass name="payload">
        <weight unit="LBS"> 1500 </weight>
        <location name="POINTMASS" unit="IN">
            <x> 460.0 </x>
            <y> 0 </y>
            <z> -31.8 </z>
        </location>
    </pointmass>
</mass_balance>
```

Here's the general format:

```
<mass_balance>
    <ixx unit="{SLUG*FT2 | KG*M2}"> {number} </ixx>
    <iyy unit="{SLUG*FT2 | KG*M2}"> {number} </iyy>
    <izz unit="{SLUG*FT2 | KG*M2}"> {number} </izz>
    <ixy unit="{SLUG*FT2 | KG*M2}"> {number} </ixy>
    <ixz unit="{SLUG*FT2 | KG*M2}"> {number} </ixz>
    <iyz unit="{SLUG*FT2 | KG*M2}"> {number} </iyz>
    <emptywt unit="{LBS | KG"> {number} </emptywt>
    <location name="CG" unit="{IN | M}">
        <x> {number} </x>
        <y> {number} </y>
        <z> {number} </z>
    </location>
    <pointmass name="{string}">
        <weight unit="{LBS | KG}"> {number} </weight>
        <location name="POINTMASS" unit="{IN | M}">
            <x> {number} </x>
            <y> {number} </y>
            <z> {number} </z>
        </location>
    </pointmass>
    ... other point masses ...
</mass_balance>
```

3.1.4 *Buoyant Forces*

Hot air balloons, buoyancy-assisted vehicles, and zeppelins can be modeled through the use of gas cells and ballonets. The general format of a gas cell and ballonet definition is shown as:

```
<buoyant_forces>
  <gas_cell type="{HYDROGEN | HELIUM | AIR}">
    <location unit="{M | IN}">
      <x> {number} </x>
      <y> {number} </y>
      <z> {number} </z>
    </location>
    <x_{radius|width} unit="{M | IN}"> {number} </x_{radius|width}>
    <y_{radius|width} unit="{M | IN}"> {number} </y_{radius|width}>
    <z_{radius|width} unit="{M | IN}"> {number} </z_{radius|width}>
    <max_overpressure unit="{PA | PSI}"> {number} </max_overpressure>
    [<valve_coefficient unit="{M4*SEC/KG | FT4*SEC/SLUG}"> {number}
                                                  </valve_coefficient>]
    [<fullness> {number} </fullness>]
    [<heat>
      {heat transfer coefficients} [lbs ft / sec]
    </heat>]
    [<ballonet>
      <location unit="{M | IN}">
        <x> {number} </x>
        <y> {number} </y>
        <z> {number} </z>
      </location>
      <x_{radius|width} unit="{M | IN}"> {number} </x_{radius|width}>
      <y_{radius|width} unit="{M | IN}"> {number} </y_{radius|width}>
      <z_{radius|width} unit="{M | IN}"> {number} </z_{radius|width}>
      <max_overpressure unit="{PA | PSI}"> {number} </max_overpressure>
      <valve_coefficient unit="{M4*SEC/KG | FT4*SEC/SLUG}"> {number}
                                                  </valve_coefficient>
      [<fullness> {number} </fullness>]
      [<heat>
       {heat transfer coefficients} [lb ft / (sec R)]
      </heat>]
      [<blower_input>
       {input air flow function} [ft^3 / sec]
      </blower_input>]
    </ballonet>]
  </gas_cell>
</buoyant_forces>
```

The required attribute *type* in the <gas_cell> element defines the type of gas contained in the gas cell, one of HYDROGEN, HELIUM or AIR. Presently the gas is always 100% pure.

The <location> element sets the placement of the cell center in the aircraft's structural frame. Currently this is where the buoyancy and gravity forces of the cell are applied.

The elements <x_ radius>, <y_ radius>, <z_ radius>, <x_width>, <y_width>, and <z_width> define the shape and volume of the fully inflated cell in the structural frame. The supported shapes are ellipsoid (using x_ radius, y_radius, z_radius) and cylindrical along X axis (using x_width and y_radius, z_radius). The cell shape is used to compute the inertia tensor due to the mass of the contained gas.

The <fullness> element defines the initial fullness fraction of the cell, normally in the interval (0-1). A fullness value greater than 1.0 initialize the cell at higher than ambient pressure.

The element <max_overpressure> defines the maximum allowed cell overpressure with respect to the surrounding atmosphere. If the cell pressure is about to exceed this limit the excess gas is automatically and instantly valved off. This models a pressure relief valve of sufficient capacity mounted at the bottom of the cell.

The <valve_coefficient> element defines the capacity of the manual valve. The valve is considered to be located at the top of the cell. The valve coefficient determine the flow out of the cell according to:

$$\frac{dVolume}{dt} = ValveOpen * ValveCoefficient * DeltaPressure$$

where *DeltaPressure* is the difference between the internal pressure at the top of the cell and the surrounding atmosphere and *ValveOpen* is a non-negative number controlled by the user via the property *buoyant_forces/gas-cell[x]/valve_open*.

The element <heat> can contain zero or more FGFunction elements describing the heat flow from the atmosphere and surrounding environment into the gas cell. The unit is lb-ft/(sec-degR).

If there are no heat transfer functions at all the gas cell temperature will equal that of the surrounding atmosphere. A constant function returning 0 results in adiabatic behavior (i.e. no heat exchange at all with the environment).

The following is an example of how the heat flow due to conduction and radiation can be modeled:

```
<heat>
  <function name="buoyant_forces/gas-cell/dU_conduction">
   <product>
    <value> 6282.25 </value> <!-- Surface area [ft2] -->
    <value> 0.05 </value>    <!-- Conductivity [lb / (R ft sec)] -->
    <difference>
     <property> atmosphere/T-R </property>
     <property> buoyant_forces/gas-cell/temp-R </property>
    </difference>
   </product>
  </function>
  <function name="buoyant_forces/gas-cell/dU_radiation">
   <product>
    <value> 0.1714e-8 </value> <!-- Stefan-Boltzmann's constant
                                    [Btu / (h ft^2 R^4)] -->
    <value>      0.05 </value> <!-- Emissivity [0,1] -->
    <value>   6282.25 </value> <!-- Surface area [ft2] -->
    <difference>
     <pow>
      <property> atmosphere/T-R </property>
      <value> 4.0 </value>
     </pow>
     <pow>
      <property> buoyant_forces/gas-cell/temp-R </property>
      <value> 4.0 </value>
     </pow>
    </difference>
   </product>
  </function>
 </heat>
```

A <gas_cell> element may contain zero or more <ballonet> elements. A ballonet is an air bag inside the gas cell in non-rigid or semi-rigid airships and is used to maintain the shape and volume of the gas cell/envelope and keep its internal pressure higher than that of the surrounding environment. It is common for such airships to have more than one ballonet, e.g. with one ballonet in the forward part and one in the aft part of the envelope the pitch attitude of the airship can be trimmed via the relative inflation of the two ballonets (as this moves part of the contained air forward or aft, respectively).

In JSBSim ballonets are modeled as being completely contained inside the enclosing gas cell (except for the inflow and outflow valves which serves air from and to the outside atmosphere).

3.1.5 *Ground Reactions*

Contacts between the aircraft and the ground can be modelled in JSBSim so that the aircraft can realistically take off and land. Contacts can also be used to model the interaction between the ground and any part of the aircraft structure such as the wing tip.

Contacts are managed at discrete locations of the aircraft, and JSBSim does not make any guess for the contact points location from the geometry data provided in the <metrics> section. Instead the contact points must be listed in the <ground_reactions> section of the aircraft configuration file. The typical layout of the ground reactions section is as follows:

```
<ground_reactions>

  <contact type="BOGEY" name="NOSE_GEAR">
    { nose gear description ... }
  </contact>

  <contact type="BOGEY" name="LEFT_MAIN">
    { left main gear description ... }
  </contact>

  <contact type="BOGEY" name="RIGHT_MAIN">
    { right main gear description ... }
  </contact>

  <contact type="STRUCTURE" name="LEFT_WING_TIP">
    { left wing tip description ... }
  </contact>

  <contact type="STRUCTURE" name="RIGHT_WING_TIP">
    { right wing tip description ... }
  </contact>

  <contact type="STRUCTURE" name="TAIL_SKID">
    { tail skid description ... }
  </contact>

  ... additional contacts ...

</ground_reactions>
```

The number of contacts may vary from an aircraft model to the other but it is advisable to have at least 3 unaligned contacts so that the aircraft can have a stable position when resting on ground.

JSBSim can model two types of contacts:

- BOGEY which is used for landing gears
- STRUCTURE which is used for any location on the aircraft other than the landing gears (typically wing tips, nose and tail)

Both of these contact types basically result in a force which resists the penetration of the ground by the aircraft. The main difference between the two types of contacts is how the ground reaction force is computed. Furthermore the BOGEY type includes features which are typical to landing gears such as brake and steering.

Within the contact element, several characteristics are given in order to describe the ground reaction properties. Here is an example:

```
<contact type="BOGEY" name="NOSE_GEAR">
  <location unit="IN">
    <x> -6.8 </x>
    <y> 0 </y>
    <z> -20 </z>
  </location>
  <spring_coeff unit="LBS/FT"> 1800 </spring_coeff>
  <damping_coeff unit="LBS/FT/SEC"> 600 </damping_coeff>
  <static_friction> 0.8 </static_friction>
  <dynamic_friction> 0.5 </dynamic_friction>
  <rolling_friction> 0.02 </rolling_friction>
  <max_steer unit="DEG"> 10 </max_steer>
  <brake_group> NONE </brake_group>
  <retractable>0</retractable>
</contact>
```

The <location> element gives the position of the contact point which is to be compared to the ground elevation. For a landing gear, the location is generally chosen as the point on the wheel tire that is in contact with the ground when the aircraft is at rest. The coordinates are given in the structural coordinate system of the aircraft.

As mentioned earlier, the ground reactions are computed as forces that support the aircraft above the ground, and affect the motion over the ground. These forces can thus be split into two components:

- the ground normal reaction
- the ground tangential reaction

JSBSim computes the ground normal reaction by a spring/damper model. The normal reaction force is therefore obtained by the formula

$F_n = k*u + b*v$

where k is the spring stiffness, b is the spring damping, u is the displacement due to compression, and v is the rate of compression. [This is a simplification of the actual model, more details can be found in section 2.5.8 Ground Reactions in this manual.] Parameters u and v are computed by JSBSim and k and b are given respectively by the <spring_coeff> and <damping_coeff> properties in the <contact> element definition.

To compute the ground tangential force, JSBSim uses the Coulomb friction law. This means that JSBSim assumes that the ground tangential reaction is proportional to the ground normal reaction. The ground tangential reaction is thus obtained by the formula

$F_t = \mu * F_n$

where μ is the friction coefficient and F_n is the normal force described above. JSBSim uses a different value for μ depending on the state of the contact point. Three properties describe these different values: <static_friction>, <dynamic_friction> and <rolling_friction>. The static coefficient is used when the contact point is at rest, while the dynamic coefficient is used when the contact point is sliding. The last coefficient, <rolling_friction> is only relevant for the BOGEY contact type; it models the resistance of the wheel to its rolling. Experience shows that, in the general case, the static friction coefficient is higher than the dynamic coefficient which, in its turn, is much higher than the rolling coefficient, which typically has a value of about 0.02.

Landing gears can be steerable if needed. JSBSim distinguishes 3 steering configurations: fixed, steerable and castered. The steering configuration of a landing gear is described by the property <max_steer>. For a fixed landing gear (typically main landing gears), the <max_steer> element must be set to zero:

```
<max_steer> 0 </max_steer>
```

Castered wheels can be used in JSBSim by defining the <max_steer> element by

```
<max_steer unit="DEG"> 360 </max_steer>
```

Castered wheels are free to rotate and their angle is computed by JSBSim at any time from the local tangential speed. The steering angle of the castered wheel can be read from the property gear/unit[n]/steering-angle-deg where n is the number of the castered gear.

Castered wheels are generally used as tail wheels for stability of the aircraft. However, at high speed, the weight distribution of the aircraft is modified and the normal force may not be high enough to prevent the castered wheel from jittering, therefore leading to difficulties for the pilot in handling the aircraft. In order to prevent such a situation, some aircraft manufacturers have included a command to lock the castered wheel at an angle of 0 degrees. In JSBSim this feature can be obtained by setting the property gear/unit[n]/castered to 0.

For any value of <max_steer> other than 0 and 360 degrees, the gear is assumed to be steerable. By default, a steerable wheel is driven by the property fcs/steer-cmd-norm. The steering angle of such a wheel is obtained by the product of fcs/steer-cmd-norm by the value given in the <max_steer> element. For example, let's consider a gear which has been defined by

```
<max_steer unit="DEG"> 10 </max_steer>
```

if fcs/steer-cmd-norm is equal to -0.8, then JSBSim assumes the wheel is rotated by 10*(-0.8) = -8 degrees. This default behavior can however be redefined in the FCS.

In JSBSim, the landing gears are divided into braking groups. This allows independent actuation of each of the braking groups. There are 4 braking groups : NONE, CENTER, LEFT and RIGHT. Each landing gear is affected to a braking group by including the following element in the <contact> description

```
<brake_group> {NONE | LEFT | RIGHT | CENTER} </brake_group>
```

By default, a landing gear is assigned to the braking group NONE which means that there is no braking system on the gear. Gears assigned to the LEFT braking group will be driven by the property fcs/left-brake-cmd-norm, those assigned to the RIGHT braking group will be driven by

fcs/right-brake-cmd-norm and those assigned to the CENTER braking group will be driven by fcs/center-brake-cmd-norm.

The strength of the brakes is driven by the static friction coefficient.

The <retractable> property is only applicable to the BOGEY type. As its name implies it tells whether a landing gear is retractable or not. The state of a landing gear retraction is only tracked by JSBSim to enable or disable the ground interpenetration checking of this contact point. For example, this allows simulating a "belly" emergency landing following the failure of one or several landing gears failure to deploy. It has no effect on the gear aerodynamics drag. Such a calculation needs to be handled separately in the <aerodynamics> section.


```
<contact type="{BOGEY | STRUCTURE}" name="{string}">
  <location unit="{IN | M}">
    <x> {number} </x>
    <y> {number} </y>
    <z> {number} </z>
  </location>
  <static_friction> {number} </static_friction>
  <dynamic_friction> {number} </dynamic_friction>
  <rolling_friction> {number} </rolling_friction>
  <spring_coeff unit="{LBS/FT | N/M}"> {number} </spring_coeff>
  <damping_coeff unit="{LBS/FT/SEC | N/M/SEC}">
    {number}
  </damping_coeff>
  <damping_coeff_rebound unit="{LBS/FT/SEC | N/M/SEC}">
    {number}
  </damping_coeff_rebound>
  <max_steer unit="DEG"> {number | 0 | 360} </max_steer>
  <brake_group> {NONE|LEFT|RIGHT|CENTER|NOSE|TAIL} </brake_group>
  <retractable>{0 | 1}</retractable>
  <table type="{CORNERING_COEFF}">
  </table>
</contact>
```

3.1.6 *External Reactions*

JSBSim can model externally or arbitrarily applied forces and moments. Such a capability might be needed to model a catapult, hook and wire capture device, tow rope, or parachute. Similar to the ground reactions specification, any external forces are defined in an external_reactions section:

```
<external_reactions>

    <!-- Interface properties, a.k.a. property declarations -->
    [<property> ... <property>]

    <force name="name" frame="BODY|LOCAL|WIND" unit="unit">

      [<function> ... </function>]

      <location unit="units"> <!-- location -->
        <x> value </x>
        <y> value </y>
        <z> value </z>
      </location>
```

```
      [<direction> <!-- optional for initial direction vector -->
        <x> value </x>
        <y> value </y>
        <z> value </z>
      </direction>]
    </force>

</external_reactions>
```

Following initialization, the following properties are available:

external_reactions/{force name}/magnitude

external_reactions/{force name}/x

external_reactions/{force name}/y

external_reactions/{force name}/z

The x, y, and z properties define a unit vector in the chosen frame. You can thus define a force magnitude and direct it an in any direction within the chosen frame (BODY, LOCAL, or STABILITY).

An example of a parachute modeled using the external reactions specification is as follows:

```
<external_reactions>
  <!-- "Declare" the reefing term -->
  <property>fcs/parachute_reef_pos_norm</property>

  <force name="parachute" frame="WIND">
    <function>
      <product>
        <p> aero/qbar-psf </p>
        <p> fcs/parachute_reef_pos_norm </p>
        <v> 1.0 </v>  <!-- Full drag coefficient -->
        <v> 20.0 </v> <!-- Full parachute area -->
      </product>
    </function>
    <!-- The location below is in structural frame (x positive
         aft), so this location describes a point 1 foot aft
         of the origin. In this case, the origin is the center. -->
    <location unit="FT">
      <x>1</x>
      <y>0</y>
      <z>0</z>
    </location>
    <!-- The direction describes a unit vector. In this case, since
         the selected frame is the WIND frame, the "-1" x component
         describes a direction exactly opposite of the direction
         into the wind vector. That is, the direction specified below
         is the direction that the drag force acts in. -->
    <direction>
      <x>-1</x>
      <y>0</y>
      <z>0</z>
    </direction>
  </force>
</external_reactions>
```

3.1.7 *Propulsion*

The propulsion section of an aircraft configuration file contains references to specific engines, where those engines are located, the thruster that turns the engine power into a force, and tank definitions for fuel or propellant. Engine and thruster specifics are contained in an external file, and specific engine types are discussed later.

The propulsion definition looks like this:

```
<propulsion>
  <engine file="filename">
    <location unit="IN">
      <x> number </x>
      <y> number </y>
      <z> number </z>
    </location>
    <orient unit="DEG">
      <roll> number </roll>
      <pitch> number </pitch>
      <yaw> number </yaw>
    </orient>
    <feed> number </feed>
    [<feed> number </feed>
     …]
    <thruster file="filename">
      <location unit="IN">
        <x> number </x>
        <y> number </y>
        <z> number </z>
      </location>
      <orient unit="DEG">
        <roll> number </roll>
        <pitch> number </pitch>
        <yaw> number </yaw>
      </orient>
      <sense> +/-1 </sense>
      <p_factor> number </p_factor>
    </thruster>
  </engine>
  [… additional engines if present …]
  <tank type="type name">
    <location unit="IN">
      <x> number </x>
      <y> number </y>
      <z> number </z>
    </location>
    <capacity unit="LBS"> number </capacity>
    <contents unit="LBS"> number </contents>
  </tank>
```

3.1.7.1 Tanks, Fuel, Propellants

Each engine defines a list of feed tanks using the <feed> tag. Each tank has a priority value, with 1 being the highest priority. The default value is 1. Since the priority is owned by the tank, all engines will see the same priority from a given tank. Setting the priority to zero is like closing a valve "at the tank". This is the only valve you have control over in the fuel system. The tank list belonging to each engine can be thought of as a set of valves too. Each engine is connected to all tanks, but a valve "at the engine" is closed for each tank list member with a value of zero.

When refueling is activated (by setting the property "propulsion/refuel" to 1 or true) fuel is added at the same rate to ALL tanks at a total transfer rate of 6000 lbs per minute, regardless of tank priority. It also ignores what fluid is supposed to be in a particular tank, so don't use this with a rocket. When dumping fuel (by setting the property "propulsion/fuel_dump" to 1 or true) fuel is removed from ALL tanks down to any standpipe level previously defined at load-time for each tank. The dump rate is defined at load-time, and is the rate for the entire fuel system. This code ignores tank priority. If no standpipe is defined, all fuel will be dumped. Default dump rate is zero.

Fuel temperature is calculated for tanks that define an initial temperature.

The definition of a fuel tank is as follows:

```
<tank type="{FUEL | OXIDIZER}">
 [<grain_config type="{CYLINDRICAL | ENDBURNING}">
    <length unit="{IN | FT | M}"> {number} </radius>
  </grain_config>]
  <location unit="{FT | M | IN}">
    <x> {number} </x>
    <y> {number} </y>
    <z> {number} </z>
  </location>
  <drain_location unit="{FT | M | IN}">
    <x> {number} </x>
    <y> {number} </y>
    <z> {number} </z>
  </drain_location>
  <radius unit="{IN | FT | M}"> {number} </radius>
  <capacity unit="{LBS | KG}"> {number} </capacity>
  <contents unit="{LBS | KG}"> {number} </contents>
  <temperature> {number} </temperature> <!-- must be deg Fahrenheit -->
  <standpipe unit="{LBS | KG"}> {number} </standpipe>
  <priority> {integer} </priority>
  <density unit="{KG/L | LBS/GAL}"> {number} </density>
  <type> {string} </type> <!-- overrides previous density setting -->
</tank>
```

The tank *type* element represents a fuel type name:

FUEL	DENSITY (LBS/GAL)	YEARS USED	NOTES
AVGAS	6.02		
JET-A	6.74		
JET-A1	6.74		a.k.a. F-35
JET-B	6.48		designed for cold climates, similar to JP-4
JP-1	6.76	1944-1945	
JP-2	6.38	1945-1947	experimental
JP-3	6.34	1947-1951	designed for aircraft carrier use
JP-4	6.48	1951-1995	a.k.a. F-40, AVTAG
JP-5	6.81	1952-	a.k.a. F-44, AVCAT, designed for aircraft carrier use
JP-6	6.55	1956	used only in XB-70 Valkyrie
JP-7	6.61		used only in SR-71
JP-8	6.66	1996-	a.k.a. F-34, used since 1978 in NATO
JP-8+100		1994-	
JP-9		1979-	designed for cruise missiles

```
JPTS                            1956-       used only in U-2
RP-1        6.73                            rocket fuel
T-1         6.88                            Russian, similar to RP-1
ETHANOL     6.58
HYDRAZINE   8.61                            a.k.a. B-Stoff, density is
                                            for hydrazine hydrate
```

3.1.8 *Aerodynamics*

Aerodynamic forces and moments in JSBSim are defined in the <aerodynamics> section of an aircraft configuration file, or in a separate file that is included in the <aerodynamics> section through the use of the "file" attribute. Within the <aerodynamics> section there are six <axis> sections corresponding to three translational and three rotational axes. Within those <axis> sections, aerodynamic forces or moments are described using <function> constructs that define the calculation for an actual force or moment. All of the forces or moments defined within a single <axis> section are contributions to the total force or total moment for an axis. All of the contributions are summed within an <axis> to arrive at the total force or moment for that axis.

The full specification for the aerodynamics section is as follows:

```
<aerodynamics>

  <alphalimits unit="{RAD | DEG}">
    <min> {number} </min>
    <max> {number} </max>
  </alphalimits>

  <hysteresis_limits unit="{RAD | DEG}">
    <min> {number} </min>
    <max> {number} </max>
  </hysteresis_limits>

  <aero_ref_pt_shift_x>
    <function>
      {function contents}
    </function>
  </aero_ref_pt_shift_x>

  <function>
    {function contents}
  </function>

  <axis name="{LIFT | DRAG | SIDE | ROLL | PITCH | YAW}">
    {force coefficient definitions}
  </axis>

  {additional axis definitions}

</aerodynamics>
```

Optionally two other coordinate systems may be used.

1. Body coordinate system:

```
<axis name="{X | Y | Z}">
```

2. Axial-Normal coordinate system:

```
<axis name="{AXIAL | NORMAL | SIDE}">
```

Systems may NOT be combined, or a load error will occur.

3.1.9 *Systems, Autopilot, and Flight Control*

3.1.9.1 Mechanization of Components

Note that in many cases “properties” are referred to in the text and in the definition of the component. Properties are explained elsewhere, but suffice it to say for the moment that properties are like variables that are categorized into a tree structure, and accessible from the configuration file. Properties refer to various values within the simulation which represent such physical parameters as roll rate, air density, drag force, etc.

Also, the control system components that are described in this section almost all have some common features. Unless otherwise specified, the common elements are:

- <input> This element specifies the input property to the component. For some components, such as an accelerometer any input specified will have no effect. For an fcs_function, the input is optional.
- <output> Some components – particularly the final component in a channel (a chain of linked components) – are designed to set the value of a predefined property or properties. The pitch control channel results in a final elevator position (usually), and so one would expect to place in the final component definition an element such as: <*output*> *fcs/elevator-pos-norm* </*output*>. More than one <output> element can be specified in a component definition. For instance, a set of components that model the roll control system for an aircraft might set both the right-hand aileron and the left hand spoiler. In this case, there would be two <output> elements.
- <delay> Some control system components might represent real-world components that have an inherent processing latency (delay). The <delay> element can simulate this latency. There is a *type* attribute for this element that will take one of the values, *time*, or *frames*. The *frames* type is assumed, if the type attribute is omitted. The delay specifies how many frames or seconds to time-shift the output of the component.
- <clipto> The <clipto> element permits limiting of the output of a component. There are two sub-elements for the <clipto> element, <max>, and <min>. If either one is omitted, the output of the component is not limited in that direction.

3.1.9.1.1 Filter Component

The filter component can simulate any first or second order filter, as well as an integrator. The Tustin substitution is used to take filter definitions from Laplace space to the time domain. The general format for a filter specification is:

```
<typename name="name">
  <input> property </input>
  <c1> number|property </c1>
```

```
   [<c2> number|property </c2>]
   [<c3> number|property </c3>]
   [<c4> number|property </c4>]
   [<c5> number|property </c5>]
   [<c6> number|property </c6>]
   [<clipto>
      <min> number|property </min>
      <max> number|property </max>
    </clipto>]
   [<output> property </output>]
</typename>
```

For a lag filter of the form,

$$\frac{C_1}{s + C_1}$$

the corresponding filter definition is:

```
<lag_filter name="name">
  <input> property </input>
  <c1> number|property </c1>
  [<clipto>
     <min> number|property </min>
     <max> number|property </max>
   </clipto>]
  [<output> property </output>]
</lag_filter>
```

As an example, for the specific filter:

$$\frac{600}{s + 600}$$

the corresponding filter definition could be:

```
<lag_filter name="filter1">
  <input> fcs/pitch-cmd </input>
  <c1> 600 </c1>
</lag_filter>
```

For a lead-lag filter of the form:

$$\frac{C_1 s + C_2}{C_3 s + C_4}$$

The corresponding filter definition is:

```
<lead_lag_filter name="name">
  <input> property </input>
  <c1> number|property </c1>
  <c2> number|property </c2>
  <c3> number|property </c3>
  <c4> number|property </c4>
  [<clipto>
     <min> number|property </min>
     <max> number|property </max>
   </clipto>]
  [<output> property </output>]
```

```
</lead_lag_filter>
```

For a washout filter of the form:

$$\frac{s}{s + C_1}$$

The corresponding filter definition is:

```
<washout_filter name="name">
  <input> property </input>
  <c1> number|property </c1>
  [<clipto>
     <min> number|property </min>
     <max> number|property </max>
   </clipto>]
  [<output> property </output>]
</washout_filter>
```

For a second order filter of the form:

$$\frac{C_1 s^2 + C_2 s + C_3}{C_4 s^2 + C_5 s + C_6}$$

The corresponding filter definition is:

```
<second_order_filter name="name">
  <input> property </input>
  <c1> number|property </c1>
  <c2> number|property </c2>
  <c3> number|property </c3>
  <c4> number|property </c4>
  <c5> number|property </c5>
  <c6> number|property </c6>
  [<clipto>
     <min> number|property </min>
     <max> number|property </max>
   </clipto>]
  [<output> property </output>]
</second_order_filter>
```

For an integrator of the form:

$$\frac{C_1}{s}$$

The corresponding definition is:

```
<integrator name="name">
  <input> property </input>
  <c1> number|property </c1>
  [<trigger> property </trigger>]
  [<clipto>
     <min> number|property </min>
     <max> number|property </max>
   </clipto>]
  [<output> property </output>]
</integrator>
```

The trigger element shown in the integrator definition is a device used to prevent integrator

wind-up. When the value of the property specified in the trigger element takes a non-zero value, the integrator ceases to integrate.

3.1.9.1.2 Switch Component

The switch component models a switch – either an on/off or a multi-choice rotary switch. The switch can represent a physical cockpit switch, or can represent a logical switch, where several conditions might need to be satisfied before a particular state is reached. The value of the switch – the output value – is chosen depending on the state of the switch. Each switch is comprised of two or more tests. Each test has a value associated with it. The first test that evaluates to true will set the output value of the switch according to the value parameter belonging to that test. Each test contains one or more conditions, which each must be logically related (if there are more than one) given the value of the logic parameter, and which takes the form:

```
property conditional property|value
```

e.g.

```
qbar GE 21.0
```

or,

```
roll_rate < pitch_rate
```

Within a test, additional tests can be specified, which allows for complex groupings of logical comparisons. The format of the switch component is as follows:

```
<switch name="name">
  <default value="{property|number}/>"
  <test logic="{AND|OR}" value="{property|number}">
    {property} {conditional} {property|number}
    …
    [<test logic="{AND|OR}" value="{property|number}">
       {property} {conditional} {property|number}
        …
     </test>]
  </test>
  …
  [<clipto>
     <min> number|property </min>
     <max> number|property </max>
   </clipto>]
  [<output> property </output>]
</switch>
```

Here's an example:

```
<switch name="fcs/roll_ap_autoswitch">
  <default value="0.0"/>
  <test value="fcs/roll-ap-error-summer">
    ap/attitude_hold == 1
  </test>
</switch>
```

The above example specifies that the default value of the component (i.e. the output property of the component, addressed by the property, ap/roll-ap-autoswitch) is 0.0 if or when the attitude

hold switch is selected (property ap/attitude_hold takes the value 1), the value of the switch component will be whatever value fcs/roll-ap-error-summer has. There is no input element for the switch component.

3.1.9.1.3 Sample and Hold Component

The switch component supports a sample-and-hold function. To effect this, an example is presented:

```
<switch name="fcs/roll-sample-and-hold-test">
  <default value="attitude/psi-rad"/>
  <test value="fcs/roll-sample-and-hold-test">
    ap/heading_hold == 1
  </test>
</switch>
```

Look carefully and you will see that the test takes on a value of *itself* if the condition is satisfied. In the example above, the default value is the heading (psi) in radians. Once the condition is satisfied (ap/heading_hold == 1), then the component holds the heading value at the time the condition was satisfied, and outputs that until such a time when the test condition is no longer satisfied. Restated, the default value (attitude/psi-rad) is output unless the condition is satisfied. If it is, then the test value (fcs/roll-sample-and-hold-test) is output. See the previous section for more information on the switch component.

3.1.9.1.4 Summer Component

The Summer component sums two or more inputs. A bias can also be added in using the bias keyword. The form of the summer component specification is:

```
<summer name="name">
  <input> [-]property </input>
  …
  <bias> number </bias>
  [<clipto>
     <min> number|property </min>
     <max> number|property </max>
   </clipto>]
  [<output> property </output>]
</summer>
```

Note that in the case of an input property the property name may be immediately preceded by a minus sign.

Here's an example of a summer component specification:

```
<summer name="fcs/pid_sum">
  <input>  velocities/p-rad_sec </input>
  <input> -fcs/roll-ap-wing-leveler </input>
  <input>  fcs/roll-ap-error-integrator </input>
  [<clipto>
     <min> -1.0 </min>
     <max> 1.0 </max>
   </clipto>]
</summer>
```

Note that there can be only one bias statement per component.

3.1.9.1.5 Gain Component

The gain component merely multiplies the input by a gain. The form of the gain component specification is:

```
<pure_gain name="name">
  <input> [-]property </input>
  [<gain> {property|number} </gain>]
  [<clipto>
     <min> number|property </min>
     <max> number|property </max>
   </clipto>]
  [<output> property </output>]
</pure_gain>
```

The gain defaults to a value of 1.0 if not supplied.

Note: as is the case with the Summer component, the input property name may be immediately preceded by a minus sign to invert that signal.

3.1.9.1.6 Aerosurface Scaling Component

This is a modified version of the simple gain component. The normal purpose for this component is to take control inputs from a known domain and map them to a specified range. The form of the aerosurface scaling component specification is:

```
<aerosurface_scale name="name">
  <input> [-]property </input>
  <zero_centered> {true|false} </zero_centered>
  [<domain>
     <min> number </min>
     <max> number </max>
   </domain>]
   <range>
     <min> number </min>
     <max> number </max>
   </range>
  [<gain> {property|number} </gain>]
  [<clipto>
     <min> number|property </min>
     <max> number|property </max>
   </clipto>]
  [<output> property </output>]
</aerosurface_scale >
```

The gain is optional and defaults to 1.0. If the domain is not supplied, min and max default to –1.0 and +1.0, respectively. The “zero_centered” element specifies how the domain will be mapped to the range. If the zero_centered element is set to true (which is the default if it is not supplied) then the domain from the minimum (negative) value to zero is mapped to the range minimum (negative) to zero. The domain from zero to maximum is linearly mapped into the range from zero to the maximum range. Otherwise, the mapping is linear from the domain min-to-max span into the range min-to-max span. The center value may not correspond to zero.

For instance, the normal and expected ability of a pilot to push or pull on a control stick is about 50 pounds. The input to the pitch channel block diagram of a flight control system is in units of pounds. Yet, the joystick control input is usually in a range from –1 to +1. The following

component definition takes joystick inputs from –1 to +1 and maps them to ±50.

```
<aerosurface_scale name="fcs/pitch_command">
  <input> fcs/pitch-cmd-norm </input>
  <range>
    <min> -50 </min>
    <max>  50 </max>
  </range>
</aerosurface_scale >
```

Note: as is the case with the Summer component, the input property name may be immediately preceded by a minus sign to invert that signal.

3.1.9.1.7 Scheduled Gain Component

The scheduled gain component multiplies the input by a variable gain that is dependent on another property (such as qbar, altitude, etc.). The lookup mapping is in the form of a table. This kind of component might be used, for example, in a case where aerosurface deflection must only be commanded to acceptable settings – i.e at higher qbar the commanded elevator setting might be attenuated. The form of the scheduled gain component specification is:

```
<scheduled_gain name="name">
  <input> {[-]property} </input>
  <table>
    <tableData>
      …
    </tableData>
  </table>
  [<clipto>
    <min> {[-]property|value} </min>
    <max> {[-]property|value} </max>
  </clipto>]
  [<gain> {property | value} </gain>]
  [<output> {property} </output>]
</scheduled_gain>
```

An overall <gain> may be supplied that is multiplicative with the scheduled gain.

Note: as is the case with the Summer component, the input property name may be immediately preceded by a minus sign to invert that signal.

Here is an example of a scheduled gain component specification:

```
<scheduled_gain name="fcs/pitch_scheduled_gain_1">
  <input>fcs/pitch-gain-1</input>
  <gain>0.017</gain>
  <table>
    <independentVar>fcs/elevator-pos-rad</independentVar>
    <tableData>
      -0.68  -26.548
      -0.595 -20.513
      -0.51  -15.328
       …
       0.527 -20.513
       0.612 -26.548
       0.697 -33.433
    </tableData>
  </table>
```

```
</scheduled_gain>
```

In the example above, there is a table – a function of elevator angle in radians – which outputs a command in degrees. The overall GAIN value effects a degrees-to-radians conversion on the output of the table.

3.1.9.1.8 Deadband Component

This is a component that allows for some "play" in a control path, in the form of a dead zone, or deadband. The form of the deadband component specification is:

```
<deadband name="fcs/windup_trigger">
  <input> [-]property </input>
  <width> number </width>
  [<clipto>
    <min> {[-]property|value} </min>
    <max> {[-]property|value} </max>
  </clipto>]
  [<output> {property} </output>]
</deadband>
```

The width value is the total deadband region within which an input will produce no output. For example, say that the width value is 2.0. If the input is between –1.0 and +1.0, the output will be zero.

3.1.9.1.9 Limiter Component

There is currently not a unique limiter component. However, the functionality of a limiter can be achieved by using a gain component with the clipto argument set with the minimum and maximum limits.

3.1.9.1.10 Positive or Negative Value Component

There exists no unique positive or negative value component. However, the functionality of such a component can be achieved by using a gain component with the clipto argument set to 0.0 and a maximum limit (for a positive value component), or having the clipto argument set to the minimum limit and 0.0 (for a negative value component). In this way, only half of the input domain will be accepted (either the positive domain or the negative domain).

3.1.9.1.11 Absolute Value Component

The functionality of an absolute value component could be implemented by a modified gain component, though such a modification has not yet been made. An absolute value could also be achieved through the use of an FCS Function component.

3.1.9.1.12 Kinematic Component

This component models the action of a moving effector, such as an aerosurface or other mechanized entity such as a landing gear strut for the purpose of effecting vehicle control or configuration. The form of the component specification is:

```
<kinematic name="fcs/gear_control">
  <input> [-]property </input>
  <traverse>
    <setting>
      <position> number </position>
      <time> number </time>
```

```
    </setting>
    …
  </traverse>
  [<clipto>
    <min> {[-]property|value} </min>
    <max> {[-]property|value} </max>
  </clipto>]
  [<gain> {property | value} </gain>]
  [<output> {property} </output>]
</kinematic>
```

The position is the position that the component takes, and the lag is the time it takes to get to that position from an adjacent setting. For example:

```
<kinematic name="fcs/gear_control">
  <input>gear/gear-cmd-norm</input>
  <traverse>
    <setting>
      <position>0</position>
      <time>0</time>
    </setting>
    <setting>
      <position>1</position>
      <time>5</time>
    </setting>
  </traverse>
  <output>gear/gear-pos-norm</output>
</kinematic>
```

In this case, it takes 5 seconds to get to a 1 setting. As this is a software mechanization of a servo-actuator, there will often be an OUTPUT specified, such as to an elevator, gear actuator, aileron, or other mechanism.

3.1.9.1.13 FCS Function Component

One of the most recent additions to the FCS component set is the FCS Function component. This component allows a function to be created when no other component is suitable. The function component is defined as follows:

```
<fcs_function name="fcs/windup_trigger">
  [<input> [-]property </input>]
  <function>
    …
  </function>
  [<clipto>
    <min> {[-]property|value} </min>
    <max> {[-]property|value} </max>
  </clipto>]
  [<output> {property} </output>]
</ fcs_function >
```

The function definition itself can include a nested series of products, sums, quotients, etc. as well as trig and other math functions. Here's an example of a function (from an aero specification):

```
<function name="aero/coefficient/CDo">
```

```
    <description>Drag_at_zero_lift</description>
    <product>
        <property>aero/qbar-psf</property>
        <property>metrics/Sw-sqft</property>
        <table>
            <independentVar>velocities/mach</independentVar>
            <tableData>
                0.0000  0.0220
                0.2000  0.0200
                0.6500  0.0220
                0.9000  0.0240
                0.9700  0.0500
            </tableData>
        </table>
    </product>
</function>
```


3.1.9.1.14 Actuator Component

The actuator can be modeled as a "perfect actuator", with the Output being set directly to the input. The actuator can be made more "real" by specifying any/all of the following additional effects that can be applied to the actuator. In order of application to the input signal, these are:

System lag (input lag, really)

Rate limiting

Deadband

Hysteresis (mechanical hysteresis)

Bias (mechanical bias)

Position limiting ("hard stops")

There are also several malfunctions that can be applied to the actuator by setting a property to true or false (or 1 or 0).

Syntax:

```
<actuator name=”name”>
  <input> {[-]property} </input>
  <lag> number </lag>
  <rate_limit> number <rate_limit>
  <bias> number </bias>
  <deadband_width> number </deadband_width>
  <hysteresis_width> number </hysteresis_width>
  [<clipto>
    <min> {[-]property|value} </min>
    <max> {[-]property|value} </max>
  </clipto>]
  [<output> {property} </output>]
</actuator>
```

Example:

```
<actuator name=”fcs/gimbal_pitch_position”>
  <input> fcs/gimbal_pitch_command </input>
  <lag> 60 </lag>
  <rate_limit> 0.085 <rate_limit> <!-- 5 degrees/sec -->
  <bias> 0.002 </bias>
  <deadband_width> 0.002 </deadband_width>
```

```
  <hysteresis_width> 0.05 </hysteresis_width>
  <clipto> <!-- +/- 10 degrees -->
    <min> -0.17 </min>
    <max>  0.17 </max>
   </clipto>
</actuator>
```

3.1.9.1.15 Sensor Component

When the flight control system uses “sensed” values such as qbar or orientation, the sensed values are taken from the environment model, and are modeled as “perfect sensors”. That is, the values were taken directly from the equations of motion, without attempting to make them look as though they had come from a sensing device (gyro, pitot tube, etc.) – with the associated imperfections that those devices can introduce. Imperfections can include noise, drift, and truncation of precision, etc. A bias can be introduced; a bias is simply an offset introduced to the sensor.

The sensor component models imperfections and increases the realism of the control system. The only required element in the sensor definition is the input element. In that case, no degradation would be modeled, and the output would simply be the input.

For adding noise, if the type is PERCENT, then the value supplied is understood to be a percentage variance. That is, if the number given is 0.05, the variance is understood to be +/-0.05 percent maximum variance. So, the actual value for the sensor will be *anywhere* from 0.95 to 1.05 of the actual "perfect" value at any time - even varying all the way from 0.95 to 1.05 in adjacent frames - whatever the delta time.

The format of the sensor specification is:

```
<sensor name=”name” >
  <input> property </input>
  <lag> number </lag>
  <noise variation={”PERCENT|ABSOLUTE”}
      distribution={“UNIFORM|GAUSSIAN”}> number </noise>
  <quantization name="name">
    <bits> number </bits>
    <min> number </min>
    <max> number </max>
  </quantization>
  <drift_rate> number </drift_rate>
  <delay> number </delay>
  <bias> number </bias>
</sensor>
```

Example:

```
<sensor name=”aero/sensor/qbar” >
  <input> aero/qbar </input>
  <lag> 0.5 </lag>
  <noise variation=”PERCENT”> 2 </noise>
  <quantization name="aero/sensor/quantized/qbar">
    <bits> 12 </bits>
    <min> 0 </min>
    <max> 400 </max>
  </quantization>
  <bias> 0.5 </bias>
```

```
</sensor>
```

3.1.9.1.16 Translational Accelerometer

The translational accelerometer component is based on the sensor component. This component accepts no inputs – it is a dedicated translational accelerometer component. It adds a <location> element, and an <orientation> element. The location specifies where the accelerometer is located in the structure (since it is not often possible to place the accelerometer exactly at the center of gravity). If the location is not at the center of gravity, the output from the accelerometer will include the effects caused by rotational motion of the vehicle. Also, the orientation of the accelerometer may be specified relative to vehicle body frame (X positive forward, Y positive out the right side), where the rotations take place in Yaw-Pitch-Roll order.

3.1.9.1.17 PID (Proportional-Integral-Derivative) Component

The PID component has a very simple specification:

```
<pid name=”name”>
  <input> property </input>
  <kp> number </kp>
  <ki> number </ki>
  <kd> number </kd>
  [<trigger> property </trigger>]
  [<clipto>
     <min> number </min>
     <max> number </max>
   </clipto>]
  [<output> property </output>]
</pid>
```

For example,

```
<pid name="fcs/roll-pi-controller">
  <input> fcs/heading-roll-error-switch </input>
  <kp> 1.0 </kp>
  <ki> 0.10 </ki>
</pid>
```

The pid component also has the ability to implement integrator wind-up protection. If the property specified by the trigger element is non-zero, then the input to the integrator is set to zero, and the integrator will not continue to build up.

For example, if the elevator for an aircraft extends outside a band of +/- 23 degrees (for a total width of 46 degrees), then this deadband will pass out a non-zero value. The value of the deadband component is used as the trigger for the wind-up protection in the pid component definition that follows:

```
<deadband name="fcs/windup-trigger">
  <input> fcs/elevator-pos-deg </input>
  <width>46.0</width>
</deadband>

<pid name="fcs/altitude-hold-pid">
  <input> fcs/ap-alt-hold-switch </input>
  <kp> 0.031 </kp>
  <ki> 0.000028 </ki>
```

```
  <trigger> fcs/windup-trigger </trigger>
  <clipto> <min>-1.0</min>
           <max> 1.0</max> </clipto>
</pid>
```

3.1.10 *Input*

JSBSim has the capability to *output* data at runtime via a socket. The capability also exists to connect *to* JSBSim via socket for data *input*. The capability is more suited for interaction and debugging or investigations more than for interprocess communications. The aircraft configuration file may contain the optional element that defines how an outside process can connect to a running instance of JSBSim:

```
<input port="port number"/>
```

The JSBSim input command interpreter can understand several commands:

```
get 'property_name'
set 'property_name'
hold
resume
info
help
quit
```

In the case of the "get" command, if the property name supplied is not found, JSBSim will attempt to return the names of all properties that contain the supplied string – which provides a rudimentary search capability.

One way that this capability can be used with JSBSim in a standalone mode is through the use of the "telnet" command. The JSBSim standalone application can be started with some the options: "--realtime", and "--suspend". These options are useful when running with the server (Input) capability. The --realtime option causes JSBSim to run at realtime speed. The --suspend option causes JSBSim to go into hold after initialization. This allows the user to connect from another shell to the JSBSim "server", using telnet. Running JSBSim with a script and for use with the server capability can be done as follows (all on one line):

```
src/jsbsim --script=scripts/c1723.xml --realtime --suspend
```

With the above command line, JSBSim will initialize and hold. A connection via telnet to JSBSim from a different shell can then be made:

```
telnet host port
```

For example:

```
telnet localhost 1137
```

JSBSim responds over the socket:

```
$ telnet localhost 1137
Trying 127.0.0.1...
Connected to zeus.
Escape character is '^]'.
Connected to JSBSim server
JSBSim>
```

Typing "info" gives important information about the simulation run:

```
JSBSim> info
JSBSim version: 0.9.13
Config File version: 2.0
Aircraft simulated: Cessna C-172
Simulation time: 0.008
```

Property values can be set or retrieved using the get and set commands. If the exact property name is unknown, the get command can also be used to search for all properties that contain the given string (this capability can only be used while in hold):

```
JSBSim> get /inertia
inertia/mass-slugs
inertia/weight-lbs
inertia/cg-x-ft
inertia/cg-y-ft
inertia/cg-z-ft
```

The value for a specific property can be retrieved using the get command:

```
JSBSim> get inertia/cg-x-ft
inertia/cg-x-ft = 43.886700
```

Setting a property value is done as follows (for example):

```
JSBSim> set ap/elevator_cmd 1
```

Note that the "=" symbol is not used.

3.1.11 *Output*

JSBSim can output data in a number of ways. Configuring which data to output has been made as simple as possible. Output data is configured within the <output> element. An <output> specification in a vehicle configuration file is optional, but in order to get any data logged during a run, the simulation must be told what to output, where to send the data, and at what rate to output data. The <output> section can optionally be included in the vehicle configuration file, or it can be specified in a standalone file, the name of which can be supplied on the command line for the standalone executable. This approach can be useful when different sets of variables are to be logged on different occasions. The format of the <output> section is the same regardless of which approach is taken.

There are groups of related parameters that can be selected for logging. One can also select any *property* for output. Here is the format of the <output> element:

```
<output name="name" type="type" [port="int" protocol="UDP|TCP"]
rate="number">
  [<group> OFF|ON </group>]
  [<property> name </property>]
</output>
```

The <output> element has several required and optional attributes.

name is required, and is the name of the file you want the output to go to. If you select a type of "SOCKET", then the *name* attribute should be the IP address that you want to send the data to (e.g. 192.168.0.10, or *machine_name.mycompany.com*).

type is required, and can be one of:

- CSV Comma separated data. If a filename is supplied for the *name* attribute, then the data goes to that file. If "COUT" or "cout" is instead specified, the data goes to stdout. If *name* is blank the data goes to stdout, as well.
- SOCKET Sends data to another machine via a socket, where the value of the *name* attribute would then be the IP address of the machine the data should be sent to.
- FLIGHTGEAR A socket is created for sending binary data packets to an external instance of FlightGear for visuals. Parameters defining the socket (name, port, etc.) are given on the <output> line.
- TABULAR Columnar data, tab-delimited.
- NONE Specifies to do nothing. This setting makes it easy to turn on and off the data output without having to change anything else.

rate is a rate in Hz that the data is output. This value may not be *exactly* what you want, due to the dependence on the frame rate.

port is used only for socket output, and *protocol* is currently only used for socket output to FlightGear.

protocol is one of either “UDP” or “TCP”.

There are several groups of data that can be output. It is a common practice to list all of the data groups and just turn them ON or OFF as needed. The following groups are currently defined:

- simulation
- atmosphere
- massprops
- aerosurfaces
- rates
- velocities
- forces
- moments
- position
- coefficients
- ground_reactions
- fcs
- propulsion

If, for instance, the position data is wanted, it will be logged if turned on as follows:

```
<output name="data.csv" rate="20">
  <position> ON </position>
</output>
```

Properties can also be specified in the <output> element if they are not included in an output group or simply if only a selected item or subset is desired to be output. Any number of properties can be specified in the <output> element. For instance, if qbar was desired to be output by itself, one could specify that as follows:

```
<output name=”data.csv” rate=”20”>
  <property> aero/qbar-psf </propery>
</output>
```

Socket output can be performed similarly, except that the attributes have a slightly different meaning, as mentioned above. For example, if it is desired to send output over a socket to a machine with IP address 192.168.0.10, to port 6055, at 20 Hz, one would specify the output element as follows:

```
<output name=”192.168.0.10” type=”SOCKET” port=”6055” rate=”20”>
  <simulation> OFF </simulation>
  <atmosphere> OFF </atmosphere>
  <massprops> OFF</massprops>
  <rates> ON </rates>
  <velocities> ON </velocities>
  <forces> OFF </forces>
  <moments> OFF </moments>
  <position> OFF </position>
  <propulsion> OFF </propulsion>
  <aerosurfaces> OFF </aerosurfaces>
  <fcs> OFF </fcs>
  <ground_reactions> OFF </ground_reactions>
  <coefficients> OFF </coefficients>
</output>
```

In the above example, only the rates and velocities are selected to be output.

It is also important to note that any number of <output> elements can be specified in an aircraft file, and they do not have to be the same type. One could specify more than one <output> element to send selected groups of data to different files (none of the various <output> elements should send data to the same file), and at the same time send selected sets of data over sockets to different machines or to one machine at different ports – all at the same time and perhaps at different rates.

3.1.11.1 Utilities for Receiving Data over a Socket

The data that JSBSim sends over a socket when the *type* is “SOCKET” (but not “FLIGHTGEAR”) is of data type character and not numeric. This avoids problems with different ways of storing numbers in different ways across systems. There are two applications that will accept this data sent by JSBSim over a socket and display it. One is the *netcat* application that is usually present in Linux distributions and can be installed in the Cygwin environment for Windows. If you have the netcat tools you can run the program nc.exe on the target machine to listen for traffic on a particular port. For instance, the following <output> element in a JSBSim aircraft config file (or in a file specified on the command line for the JSBSim standalone application) sets up socket output from JSBSim,

```
<output name="localhost" type="SOCKET" port="1138" rate="20">
  <simulation> ON </simulation>
  <atmosphere> ON </atmosphere>
  <massprops> OFF</massprops>
  <rates> ON </rates>
  <velocities> OFF </velocities>
  <forces> ON </forces>
  <moments> ON </moments>
```

```
  <property> velocities/vc-kts </property>
  <property> attitude/phi-rad </property>
</output>
```

This causes JSBSim to send data to the local machine in port 1138 at 20 Hz. The netcat application can be commanded to listen for this data in a separate shell as follows,

```
nc -l -p 1138
```

When the command line is entered (before starting JSBSim), it listens for traffic. When JSBSim is run, data will begin scrolling by.

Another tool that can be downloaded from the JSBSim web site is a stripchart application for Windows.[2] If you download and unzip this file into a folder, it should result in an application that – when run – will sit there and wait until it begins receiving data. Selecting File|Options, one can select a different port number to listen at (the default is 1138). When ready, one can click on the Start button and the application will begin listening.

JSBSim can then be started, running an aircraft model that has an <output> spec that looks like the <output> element shown above. Of course, this works best when JSBSim is run in realtime mode (either from the standalone application, or when JSBSim is incorporated into another application). JSBSim can be run like this when using the stripchart application,[3]

```
src/jsbsim --script=scripts/c1723 --realtime --nice
```

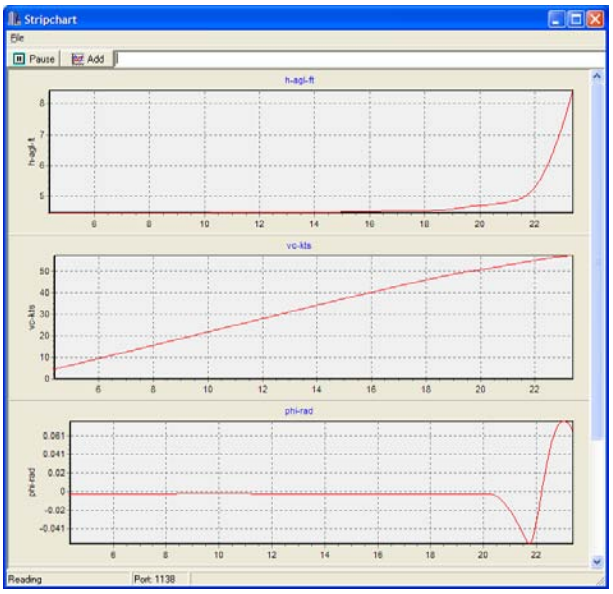
3.1.11.2 Runtime Control of Logging

It is possible at runtime to change the log rate for any output set. You can even set the log rate to zero (which effectively disables data logging). Changing the data logging rate can be done from a script, or from within the telnet interface, or through any process that can change a property. The property that controls the data logging rate is,

```
simulation/output[#]/log_rate_hz
```

In the property name, the "#" represents the number of the output set specified. Each <output> element in the aircraft configuration file is numbered sequentially beginning with zero (0). The 0 index is really not required, Any output directives files mentioned on the command line (if the standalone version of JSBSim is being used) continue the numbering after the <output> elements in the aircraft configuration file have been read and stored.

Setting the value of the log_rate_hz property to 0 turns off logging.



[2] Download the stripchart application at www.jsbsim.org/stripchart.zip

[3] The shutdown of the stripchart application is not very graceful at this time.

3.1.11.3 Post-processing

There are many spreadsheet and plotting programs that can post-process the data files that JSBSim produces. The data files will either be tab or comma-delimited data files, with the first row containing the names of the data elements, and with the first column always being time in seconds.

The JSBSim source distribution contains a project called prep_plot. The prep_plot program processes comma-separated data files and outputs a set of directives that serve as an input to the open source Gnuplot program (www.gnuplot.info). The application takes several options:

```
prep_plot filename
  [--plot=directives_file]       // Can be present multiple times
  [--comp]
  [--title=title]
  [--start=time]
  [[--end=time]
```

The *filename* argument is the name of the data file (myfile.csv, for instance). The *--plot* option allows an XML format plot layout file to be given. That file format is described in the appendix. [Not yet added.] The *--comp* argument is short for "comprehensive," which causes the application to plot everything in the data file, whether it is mentioned in any file supplied in the *--plot* option or not. The *–title* option allows a title to be supplied, which is printed at the top of each plot.

Briefly, the plot layout file contains a description of which data items to plot. For example,

```
<page>
 <plot>
   <title>Temperature vs. Time</title>
   <label axis="y">Temperature (R)</label>
   <label axis="x">Time (sec)</label>
   <scale>auto</scale>
   <parameter axis="x">Time</parameter>
   <parameter axis="y">Temperature (R)</parameter>
 </plot>
 <plot>
   <title>Pressure vs. Time</title>
   <label axis="x">Time (sec)</label>
   <label axis="y">Pressure (psf)</label>
   <scale>auto</scale>
   <parameter axis="x">Time</parameter>
   <parameter axis="y">P_{Ambient} (psf)</parameter>
 </plot>
 <plot>
   <title>Density vs. Time</title>
    <label axis="y">Rho (slugs/ft^3)</label>
    <label axis="x">Time (sec)</label>
    <scale>auto</scale>
    <parameter axis="y">Rho (slugs/ft^3)</parameter>
    <parameter axis="x">Time</parameter>
 </plot>
</page>
```

In this case, the prep_plot program will emit gnuplot commands to generate a page that has three plots on it, plotting time versus temperature, pressure, and density.

There are two subdirectories in the JSBSim source code distribution that deal with data

output: data_output/ and data_plot/.

3.2 Engines

3.2.1 *Piston*

3.2.2 *Turbine*

3.2.3 *Turboprop*

3.2.4 *Rocket*

Both liquid fuel and solid fuel rockets can be modeled in JSBSim. The characteristic of a rocket engine known as *Specific Impulse* is a key parameter in the calculation of thrust for both types of engine. Specific Impulse (known as I_{sp}, hereafter) is defined as follows ("*w dot*" is the weight rate of consumption of the propellants):

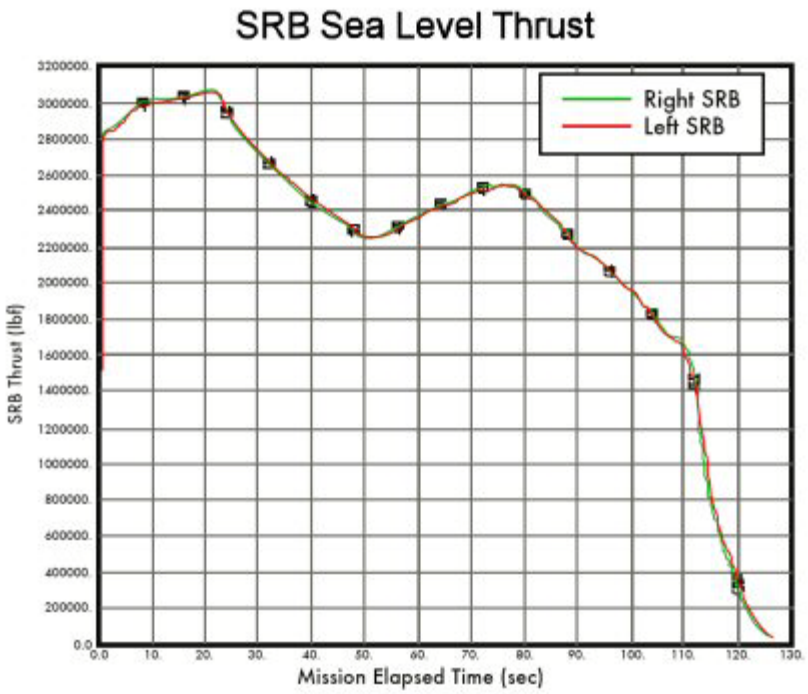
$$I_{sp} = \frac{Thrust}{\dot{w}}$$

3.2.4.1 Solid Rocket Motors

Solid rockets have fixed thrust versus time performance characteristics. Usually, one will see thrust versus time curves given to describe the burn characteristics of a solid rocket. Knowing the I_{sp} of the particular solid rocket propellant, a running total of the amount of fuel burned can be calculated as a function of time. With JSBSim, the solid rocket motor thrust is defined in a table of thrust versus the amount of propellant burned. This approach allows easier modification of the thrust characteristics when it is desired to disperse the performance of a motor. The temperature of the propellant at ignition can affect how fast the propellant burns. Manufacturing variations can change the total impulse provided by a specific copy of a rocket motor.

Here's an example of a thrust (sea level) versus time curve – this is for the space shuttle solid rocket booster (SRB)[4]:

[4] Image from Wikipedia



Using a tool such as Plot Digitizer5 a plot such as the one above can be digitized and columns of data representing the trace can be produced. Such data can be loaded into a spreadsheet program such as Excel and manipulated. A column for fuel expended can be added, with the column values determined by the relationship between Isp, thrust, and propellant flow rate (see above). Once these values are all known, the solid rocket engine table and the entire definition can be specified in XML.

The items that need to be known – apart from the thrust and propellant flow rate mentioned above – also include Isp and thrust ramp-up time. [There are some additional characteristics that need to be known for the "tank" definition so that mass properties can be properly calculated. Those are discussed in the tank section, 3.3.4.1 below.] One can also supply a thrust/burn time dispersion as well as a total Isp dispersion. Such dispersions might be applied if one wishes to make a series of runs representing a wide range of conditions that might be encountered. For instance, on a hotter day, the propellant will burn faster, and produce greater thrust. Also, manufacturing variations can result in slight changes in the propellant grain makeup, resulting in above or below average total energy (total impulse) for the engine.

The solid propellant motor specification is as follows:

```
<?xml version="1.0"?>
<rocket_engine name="solid rocket name">
  <isp> number </isp>
  <builduptime> time </builduptime>
  [<variation>
    [<thrust> percent </thrust>]
    [<total_isp> percent </total_isp>]
  </variation>]
  <thrust_table name="name" type="internal">
```

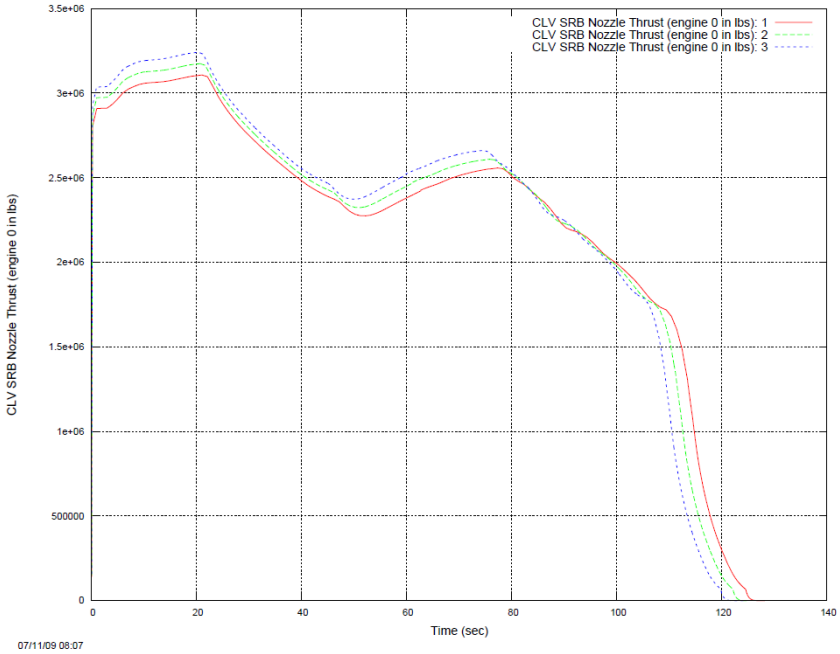
[5] http://plotdigitizer.sourceforge.net/

```
    <tableData>
            …         …
    </tableData>
  </thrust_table>
</rocket>
```

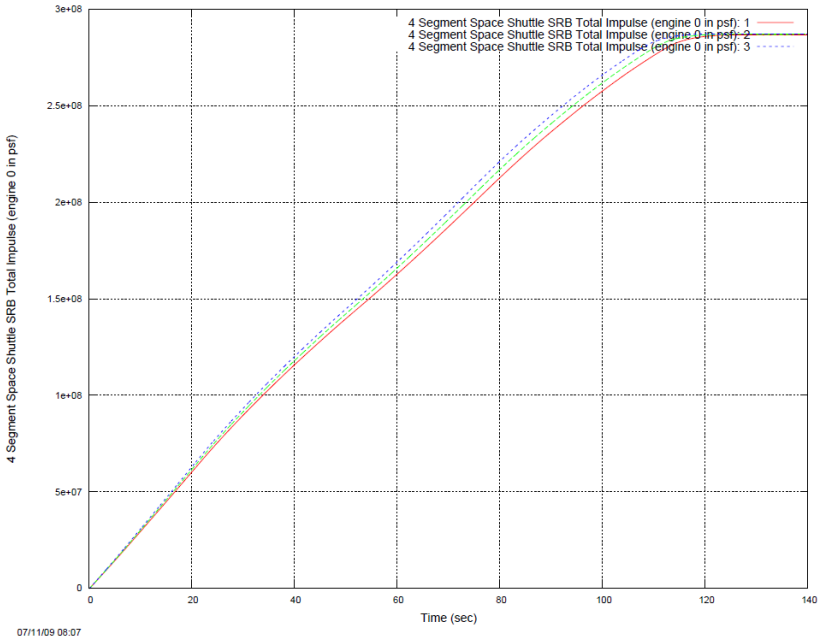
An example is shown below for the four segment space shuttle solid rocket booster:

```
<?xml version="1.0"?>
<rocket_engine name="4 Segment Space Shuttle SRB">
  <!--
    This Space Shuttle Solid Rocket Booster (SRB) model is a standard
    four segment SRB, based on information publicly available at
    http://www.sworld.com.au/steven/space/shuttle/sim/,
    As well as on Wikipedia, and at www.astronautix.com.

    This table starts with a thrust of 3,060,000 lbs at no mass burned
    - which is an immediate thrust. However, the buildup of 0.2 seconds
    gives a sine wave ramp up of thrust from 0.0 to whatever the
    calculated value of thrust is.
  -->

  <isp> 268 </isp>
  <builduptime> 0.2 </builduptime>
  <variation>
    <thrust>0.02</thrust>
    <total_isp>0.00</total_isp>
  </variation>
  <thrust_table name="propulsion/thrust_prop_remain" type="internal">
    <tableData>
<!--     =========  =========      ===========  =========    =====
         Prop Burn   Thrust        Total Impls    wdot        Time
         =========  =========      ===========  =========    =====  -->
           0.0      3060000.0 <!--    316699.9    11861.4      0.2  -->
           1186.1   3060000.0 <!--    316699.9    11861.4      0.2  -->
           10675.3  3166999.3 <!--   2850299.4    11861.4      1.0  -->
           22519.8  3167999.3 <!--   6012798.7    11827.7      2.0  -->
           …        …
         1113965.3    68000.0 <!-- 297428734.3      254.7    124.0  -->
         1114092.6        0.0 <!-- 297462734.3        0.0    125.0  -->
    </tableData>
  </thrust_table>
</rocket_engine>
```

In operation, the rocket motor performance can be seen in the plot of thrust versus time:



The thrust versus time plot shows the nominal thrust level trace (the solid red line), and two additional traces for a thrust variation of +0.02% and +0.04% (the dashed green and blue traces, respectively). The boost given to thrust from the positive variation results in a corresponding shorter burn time, and identical final total impulse values, as shown in the plot:



Note that thrust and total impulse variations can be set at runtime through the properties:

- propulsion/engine/thrust-variation_pct
- propulsion/engine/total-isp-variation_pct

The engine number may vary to include a subscript ("[1]", for example) if there is more than

one engine.

3.2.4.2 Liquid Fuel Rocket Motors

3.2.5 *Electric*

3.3 Thrusters

3.3.1 *Propeller*

3.3.2 *Nozzle*

3.3.3 *Direct*

3.3.4 *Tanks*

3.3.4.1 Solid Rocket Propellant

3.3.4.2 Liquid Rocket Propellant

3.3.4.3 Hybrid Solid/Liquid Rocket

3.3.4.4 Aircraft Fuel

3.4 Initialization

4. Scripting

4.1 Overview and Example

JSBSim can be scripted to make runs automatically. Commands are specified using the Scripting Directives for JSBSim. The script file is in XML format. A test condition (or conditions) can be set up in an event in a script and when the condition evaluates to true, the specified action[s] is/are taken. An event can be persistent, meaning that at all times when the test condition evaluates to true the specified set actions take place. When the set of tests evaluates to true for a given condition, an item may be set to another value. This value may be a value, or a delta value, and the change from the current value to the new value can be either via a step function, a ramp, or an exponential approach. The speed of a ramp or approach is specified via the time constant. Here is an example that shows the format of the script file:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
  href="http://jsbsim.sourceforge.net/JSBSimScript.xsl"?>

<runscript xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
     xsi:noNamespaceSchemaLocation=
     "http://jsbsim.sf.net/JSBSimScript.xsd"
     name="C172-01A takeoff run">

  <!--
    This run is for testing the C172 altitude hold autopilot
  -->

  <use aircraft="c172x" initialize="reset00"/>
  <run start="0.0" end="3000" dt="0.0083333">

    <property> simulation/notify-time-trigger </property>

    <event name="engine start">
      <description>Start the engine</description>
      <notify/>
      <condition>
        sim-time-sec >= 0.25
      </condition>
      <set name="fcs/throttle-cmd-norm" value="1.0" action="ramp" tc ="0.5"/>
      <set name="fcs/mixture-cmd-norm" value="0.87" action="ramp" tc ="0.5"/>
      <set name="propulsion/magneto_cmd" value="3"/>
      <set name="propulsion/starter_cmd" value="1"/>
    </event>

    <event name="Set heading hold">
      <description>Set Heading when 5 ft AGL is reached</description>
      <notify/>
```

```
      <condition>
        position/h-agl-ft >= 5
      </condition>
      <set name="ap/heading_setpoint" value="200"/>
      <set name="ap/attitude_hold" value="0"/>
      <set name="ap/heading_hold" value="1"/>
    </event>

    <event name="Set autopilot for 20 ft.">
      <description>Set Autopilot for 20 ft</description>
      <notify/>
      <condition>
        aero/qbar-psf >= 4
      </condition>
      <set name="ap/altitude_setpoint" value="100.0"/>
      <set name="ap/altitude_hold" value="1"/>
      <set name="fcs/flap-cmd-norm" value=".33"/>
    </event>

    <event name="Set autopilot for 4,000 ft.">
      <description>Set Autopilot for 4000 ft</description>
      <notify/>
      <condition>
        sim-time-sec >= 1500
      </condition>
      <set name="ap/altitude_setpoint" value="4000.0"/>
    </event>

    <event name="Set autopilot for 7000 ft.">
      <description>Set Autopilot for 7000 ft</description>
      <notify/>
      <condition>
        sim-time-sec >= 2500
      </condition>
      <set name="ap/altitude_setpoint" value="7000.0"/>
    </event>

    <event name="Time Notify" persistent="true">
      <description>Output message at 100 second intervals</description>
      <notify>
        <property>velocities/vc-kts</property>
        <property>position/h-agl-ft</property>
      </notify>
      <condition>
        sim-time-sec >= simulation/notify-time-trigger
      </condition>
      <set name="simulation/notify-time-trigger" value="100" type="delta"/>
    </event>

  </run>

</runscript>
```

The first line must always be present - it identifies the file as an XML format file. The second line (*runscript* element) identifies this file as a script file, and gives a descriptive name to the script file. Comments are next, delineated by the <!-- and --> symbols. The aircraft and initialization files

to be used are specified in the "use" lines. Next, comes the *run* section, where the conditions are described in "event" elements.

4.2 Events

Events control which actions take place and when. An event element includes a condition or conditions that must be satisfied for the event to be triggered, actions that are taken when the conditions are satisfied, and whether to print a message stating that the event has been triggered. The overall layout of an event is as follows:

```
<event name="text" [[persistent="true|false"] |
                    [continuous="true|false"]]>
  [<description> text </description>]
  <condition [logic="AND|OR"]>
    text
    [<condition [logic="AND|OR"]> … </condition>]
  </condition>

  <!-- Form 1 for the set element -->
  [<set name="text" value="number"
     [type="value|delta"]
     [action="step|ramp|exp" [tc="number"]]/>

  <!-- Form 2 for the set element -->
  [<set name="text"
     [type="value|delta"]
     [action="step|ramp|exp" [tc="number"]]>
    <function>
      … <!-- Function definition --> …
    </function>
  </set>]

  [<delay> number </delay>]
  [<notify>
    [<property> name </property>]
    [ … ]
  </notify>]
</event>
```

If the event is to execute every single frame (while the condition evaluates to true) then the *continuous* attribute should be set to *true*. This could be useful for instance if a set of wind direction and magnitude was given – perhaps as lookup tables as a function of altitude. Such an event would look like this,

```
<event name="Winds aloft" continuous="true">
  <condition>sim-time-sec  ge  0.05</condition>
  <set name="atmosphere/psiw-rad">
    <function>
      <table>
        <independentVar lookup="row"> position/h-agl-ft </independentVar>
        <tableData>
        <!-- Alt (ft)  Dir (rad) -->
              0.0    0.80
           5000.0    0.90
          10000.0    1.00
```

```
            15000.0    0.90
            20000.0    0.70
            25000.0    0.80
            30000.0    0.90
            35000.0    1.00
            40000.0    1.10
            45000.0    1.20
            50000.0    1.30
            55000.0    1.40
            60000.0    1.30
            65000.0    1.20
            70000.0    1.30
           100000.0    1.40
           200000.0    1.50
         </tableData>
       </table>
     </function>
   </set>
   <set name="atmosphere/wind-mag-fps">
     <function>
       <table>
         <independentVar lookup="row"> position/h-agl-ft </independentVar>
         <tableData>
         <!-- Alt (ft)  Speed (fps) -->
                0.0     10.0
             5000.0     40.0
            10000.0     60.0
            15000.0    120.0
            20000.0    135.0
            25000.0    140.0
            30000.0    130.0
            35000.0    120.0
            40000.0    110.0
            45000.0    110.0
            50000.0    100.0
            55000.0     90.0
            60000.0     85.0
            65000.0     60.0
            70000.0     40.0
            100000.0    10.0
            200000.0     5.0
         </tableData>
       </table>
     </function>
   </set>
   <notify/>
</event>
```

If an event is to execute each time the condition evaluates to true after having been false, set the *persistent* attribute to true. The event will be reset and the event actions will be performed once each time the condition toggles to true. The condition will evaluate to false when the condition is no longer true. By default – without the *persistent* or *continuous* attribute set – event actions will be executed exactly once when first triggered, and not again.

4.2.1 *Conditions*

The conditional tests are modeled using a "standard" JSBSim conditional construct that is also used in the flight control switch component. Conditions are specified to determine when the listed actions (if any) should take place. Conditions can be complex. For instance, a condition can include various groupings of condition groups, such as (A and (B or C)). A single, basic, conditional check would look like this,

```
<condition>
  sim-time-sec gt 0.50
</condition>
```

This condition merely checks to see that the simulation time is greater than 0.50 seconds. If it is, then the condition evaluates to true and the event is triggered. We can expand that to include a condition group,

```
<condition>
  sim-time-sec gt 0.50
  <condition logic="OR">
    aero/qbar-psf gt 4.5
    gear/unit/WOW eq 0
  </condition>
</condition>
```

This conditional test will evaluate to true if the simulation time is greater than 0.50 seconds and either qbar is greater than 4.5 or the nose gear WOW (Weight-On-Wheels) flag is 0 (meaning that the nose gear has lifted off). Note that the logic attribute defaults to "AND."

The set of conditional test operators that can be used is as follows:

EQ (equal to)
NE (not equal to)
LT (less than)
LE (less than or equal to)
GT (greater than)
GE (greater than or equal to)

The conditional operators can also be written in lower case. The standard mathematical operators could also be used (i.e. "<=" instead of "LE"), however they can cause problems with some XML applications, since the angle brackets have special meaning. It is best to avoid their use.

4.2.2 *Actions (the "set" element)*

When the condition evaluates to true, any actions specified will be carried out. There are several attributes that control the action taken for each *set* element.

name – The name attribute is the property that will be set.

value – The value that the property will be set to – this can be a numeric value or another property.

action – Specifies the way in which a new value will be assigned to the named property. The allowed values are: step, ramp, exp. The *step* value is the default, and is assumed if the action is not given in the *set* element.

type – Specifies whether the new value to be assigned is an actual value, or a delta to the

existing value. The allowed values are: value, delta. The *value* choice is the default and is assumed if the *type* attribute is not given.

tc – Specifies the numeric time constant that controls how fast an exponential or ramp action will be applied to a property value.

A second form of the *set* element permits the specification of a *function* that supplies the value to apply to the named property. In this case, the *value* attribute is not given, but the function is provided. See the "Winds Aloft" example in section 4.2 above for an example of the second form.

4.2.3 *Delay*

A delay can be specified for an event, so that it executes a certain number of seconds after it is triggered.

4.2.4 *Notifications*

An event can be a notify-only event. That is, you do not have to specify actions for an event, it can exist to notify the user of some kind of event. When coupled with the property declaration capability some interesting effects can be achieved. One of those is a repeating notification that is printed at time or altitude intervals. See the Time Notify event in the above example.

Formal documentation for the JSBSim Script Language can be found at, http://jsbsim.sourceforge.net/JSBSim/JSBSimScript.xsd.html

Section 2: Programmer's Manual

1. Overview

One of the specific goals of the design of JSBSim is to make sure it is relatively easy to incorporate into a small or large simulation program[6]. As often happens with simulation programs (and software applications in general) "feature creep" sets in and the program code blossoms into a larger-than-intended product. In the case of JSBSim, we have attempted to maintain the code base as small as possible, but at first glance it can still be overwhelming.

Unlike some simulator applications, JSBSim (at the time of this writing) has no external dependencies. That is, all code that is needed to build JSBSim is packaged along with it. This greatly simplifies the building of the JSBSim executable. There is also no automatically generated code within JSBSim. All code is straight C++ (with some C code from the included eXpat XML parsing code). Included with JSBSim source code are a couple of project files for Microsoft Visual C++ Express IDEs (2008 and 2010 edition at this time). JSBSim code can also be built under Linux, Cygwin, or other Unix-like environments that have the gnu g++ compiler suite and development tools (automake, libtool, configure, etc.). Makefiles are included for that purpose.

What does the JSBSim source code consist of? Let's think for a moment about what is present in a 6 degree of freedom rigid body software library. We want to control a vehicle, and want to see where it goes. That means we have to take inputs, calculate the forces and moments that act on the vehicle, and propagate the location and vehicle dynamics over time. The aerodynamic characteristics of the vehicle need to be modeled – and that in turn depends on the atmosphere. Winds can be modeled, too. Gravity needs to be modeled. Clearly, the *environment* needs to be modeled. As for the aircraft itself, apart from the aerodynamic characteristics which cause forces and moments to be applied, propulsive effects must be modeled, buoyancy, ground reactions, ground contact, and other externally applied forces can be modeled. Flight control characteristics must be modeled, and other systems can be modeled as well (such as an autopilot, other Guidance, Navigation, and Control (GNC) systems, or even electric or hydraulic systems). Finally, the forces and moments cause the aircraft or vehicle to move, and so the location and orientation of the vehicle must be tracked and propagated over time. There are of course a lot of models to keep track of, so some kind of model management or executive is needed. It's important to be able to input data to and extract data from the simulation, so an input and output capability is needed. Lastly, there are convenience or utility functions such as functions, tables, matrices, and quaternions, which facilitate the functioning of the models.

For JSBSim, the C++ programming language was chosen to implement the above mentioned models and ideas in program code. Over the years, this author has fielded questions from people who ask why that language was chosen instead of something else. Some old-timers have sworn

[6] Feedback from various integrators who have incorporated JSBSim into a larger simulation framework indicate that JSBSim can be integrated quickly – on the order of 10 to 15 hours for the initial integration, and a week or two following that for attending to details.

by Fortran as being the language of choice for engineering applications. Some have suggested that Java would be good for implementing a flight dynamics model. One could use any language but given the current state of the standard C++ language across many platforms, the ready availability of excellent compilers and IDEs (Integrated Development Environments) – even free ones – and the similarity of C++ to the C programming language, the choice of C++ seems natural. It is one of the more widely known and used languages today. Object oriented programming is well supported by C++ - and C++ is fun!

The C++ programming language is ideal for use in flight simulation software in part because of its support of the primary object-oriented concepts: *polymorphism*, *inheritance*, *encapsulation*, and *abstraction*.

Polymorphism (i.e. “having many forms”) is a characteristic of objects that operate differently internally, but have the same interface. For example, the JSBSim implementation of a flight control system (FCS) consists of various classes of filters, switches, and gains that each have a unique internal implementation. The public interface to the components consists of the Run() and GetOutput() methods, which are both virtual member functions of the component base class. These two methods are overridden in derived classes that represent specific components with unique behavior. Each instance of a component is referred to by the base class pointer, but the behavior is determined by the logic defined within the Run() method of the specific derived component class.

Inheritance is the derivation of a class from another in order to provide base class characteristics to the derived class. Using the example of the JSBSim FCS component classes again, each of the specific component classes (switch, summer, filter, etc.) is derived from a more generic component class that features characteristics that all components share. Inheritance is a common way to implement polymorphism.

Encapsulation is the concept of hiding and protecting data in a class, preventing corruption by outside processes. For example, the output of the FCS components may only be retrieved, and not modified. In addition, some of the JSBSim FCS components store past values. These should never be accessible by any other part of the program. Abstraction is “the elimination of the irrelevant and the amplification of the essential”. The FCS component base class, for instance, provides the two simple functions Run() and GetOutput() that are used to operate a component. The details of how each specific component operates do not need to be exposed.

The Standard Template Library (STL) is a library of containers and algorithms. The STL provides containers such as the vector class that operate like arrays that shrink or grow as needed. This capability is particularly useful in JSBSim, because the number of engines, coefficients, FCS components, etc. is unknown until the program is executed.

In the simplest use, where JSBSim is to be called upon from a tiny application that simply runs a command script for an aircraft, here is an example of how JSBSim could be used:

```
#include <FGFDMExec.h>

int main(int argc, char **argv)
{
  JSBSim::FGFDMExec FDMExec;

  FDMExec.LoadScript(argv[1]);
```

```
  while (FDMExec.Run());
}
```

The above code can actually be compiled and run. If the above code replaces the JSBSim.cpp file in the basic JSBSim distribution, typing "make" should make the code. It could subsequently be run by simply typing (assuming you have a script called “myscript.xml” in the scripts/ subdirectory):

```
src/JSBSim.exe scripts/myscript.xml
```

The "Executive" class object, "FGFDMExec" does most of the work in setting up and running JSBSim. In words, the above program – when compiled and run – will create an instance of an FGFDMExec object, load a script (which in turn loads an aircraft file and an initial conditions file), and cyclically run the simulation until the simulation completes as specified in the script.

For creating an application that runs inside a larger simulation program, where command inputs are made via joystick and outputs drive a visual scene, the following code excerpt shows how this can be done:

```
int initialize_flight_dynamics(char *name) {
  FDMExec = new FGFDMExec();
  FDMExec->LoadAircraft(name);
  FDMExec->DoTrim(mode);
}

int run_flight_dynamics() {
  copy_inputs_to_jsbsim();
  FDMExec->Run();
  copy_outputs_from_jsbsim();
}
```

The above two functions would be linked into the main program code, with the initialization function being called (as the name suggests) at initialization, and the latter function being called cyclically during runtime. The "copy_inputs_to_jsbsim" and "copy_outputs_from_jsbsim" functions are separate functions that would need to be written. Those two functions would take care of providing inputs to and extracting outputs from JSBSim. That process will be discussed later.

In the following sections, the layers and processes will be looked at from a high level, followed by a more detailed look.

2. Class Heirarchy

There are roughly 70 C++ classes that comprise JSBSim. The code files themselves are distributed into subdirectories based on function. There are math classes, classes that aid input/output and initialization, model classes, and basic classes. There are some key inheritance relationships that should be understood. For instance, there is a model class (FGModel), from which specific model classes inherit, and base FGEngine, FGFCSComponent, FGThruster, and FGParameter classes. These inheritance relationships are shown in Figure 1.

The collection of classes that make up the JSBSim framework resembles a "family tree" with a common parent at the head of the hierarchy. This graphically illustrates the concept of inheritance and – indirectly – code reuse. An entire class framework does not necessarily have to conform to a completely hierarchical arrangement. In JSBSim, several classes represent generic objects or concepts that are each fully realized in derived classes. These base classes can be viewed as providing the abstract interface that is used to access each of the derived objects. There are also operations and attributes (a fancy name for class member variables) that are common to almost all classes, so it makes sense to implement those characteristics in a common base class. For instance, the framework base class (FGJSBBase) implements a messaging capability,

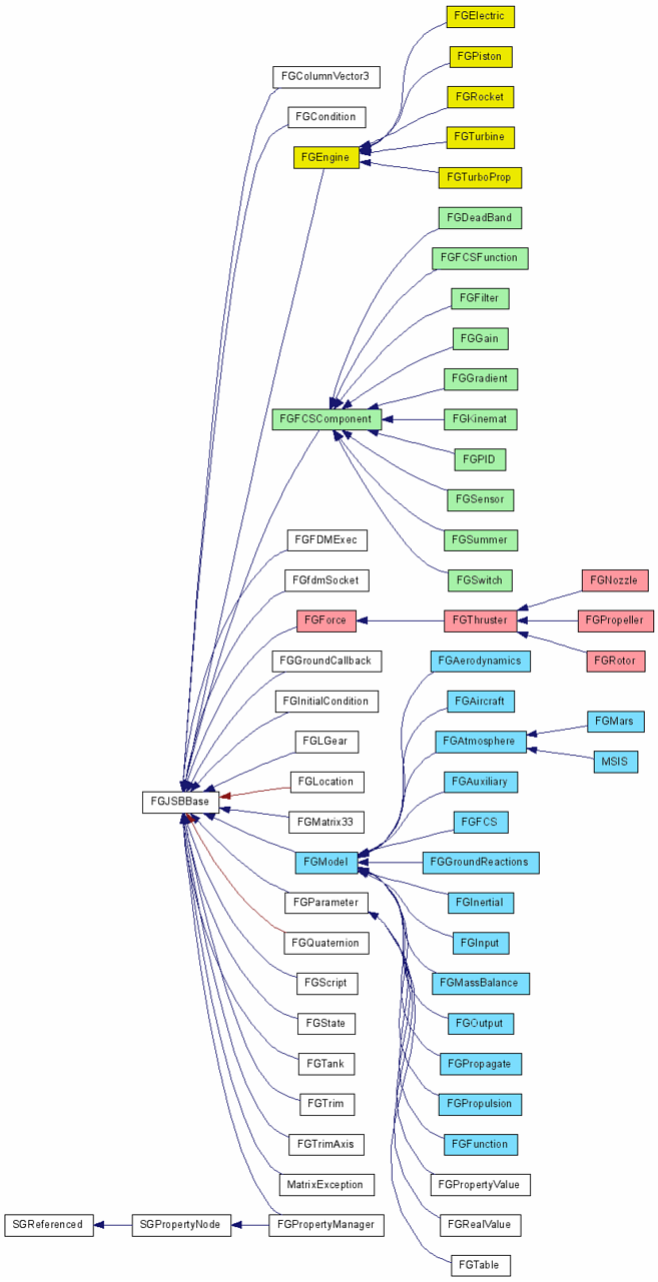


Figure 5 JSBSim class heirarchy

providing message handling methods and storage. Any descendant class can put a message on the queue to be retrieved and processed by the parent application, or by any class instance that is also derived from the base class. Unit conversion routines and constants also reside in the base class, available through inheritance to any of the many descendant classes that need them.

One of the key challenges faced in creating a generic FDM is in designing it to permit the modeling of completely arbitrary vehicle configurations. The framework needs to be able to transparently handle modeling craft ranging from a simple ball (useful for testing purposes, see Case Study 1), to a missile, an aircraft, rocket, hybrid vehicle, a rotorcraft, and so on. These craft could feature different propulsion systems, ground reaction mechanisms, aerodynamic characteristics, and control systems (if any). All four of the previously mentioned object oriented design characteristics played a part in addressing this challenge. These classes will be discussed in the sections that follow.

2.1 Directory organization

The code is located in directories that contain related groups of files. The sourec directory, *src*, is the root directory for the code. Within that directory are the executive class source and header files (FGFDMExec), the standalone JSBSim source file (JSBSim.cpp), the base class for many of the other classes (FGJSBBase), and the state class (FGState). Underneath the *src/* directory are additional subdirectories,

- initialization
- input_output
- math
- models
- simgear
- utilities

2.2 Base class

2.3 Executive class

The FGFDMExec class encapsulates the JSBSim simulation Executive. All other simulation classes are instantiated, "owned", initialized, and run through the Executive. When integrated with FlightGear (or other flight simulator) this class is typically instantiated by an interface class on the simulator side.

At the time of simulation initialization, the interface class (or, broadly: the application that is "calling" JSBSim) creates an instance of this executive class. The executive is subsequently directed to load the chosen aircraft configuration file:

```
fdmex = new FGFDMExec( … );     // (1) Instantiation
result = fdmex->LoadModel( … ); // (2) Model loading
```

When FGFDMExec is instantiated (see (1), in the above code snippet), several actions are taken within the FGFDMExec class as the constructor for FGFDMExec is called, creating an instance of FGFDMExec:

- The subsystem model classes (Atmosphere, Propulsion, Aircraft, etc.) are instantiated and initialized.

- The subsystem models are scheduled - that is a list of models is made that will be called in order when each frame is cyclically processed.

After the Executive is created, the aircraft model is loaded. When an aircraft model is loaded, the config file (the name of which is supplied in the LoadModel() function of the Executive class) is parsed and for each of the sections of the config file (propulsion, flight control, etc.) the corresponding Load() method is called (e.g. FGFCS::Load() is called to load the flight control system parameters).

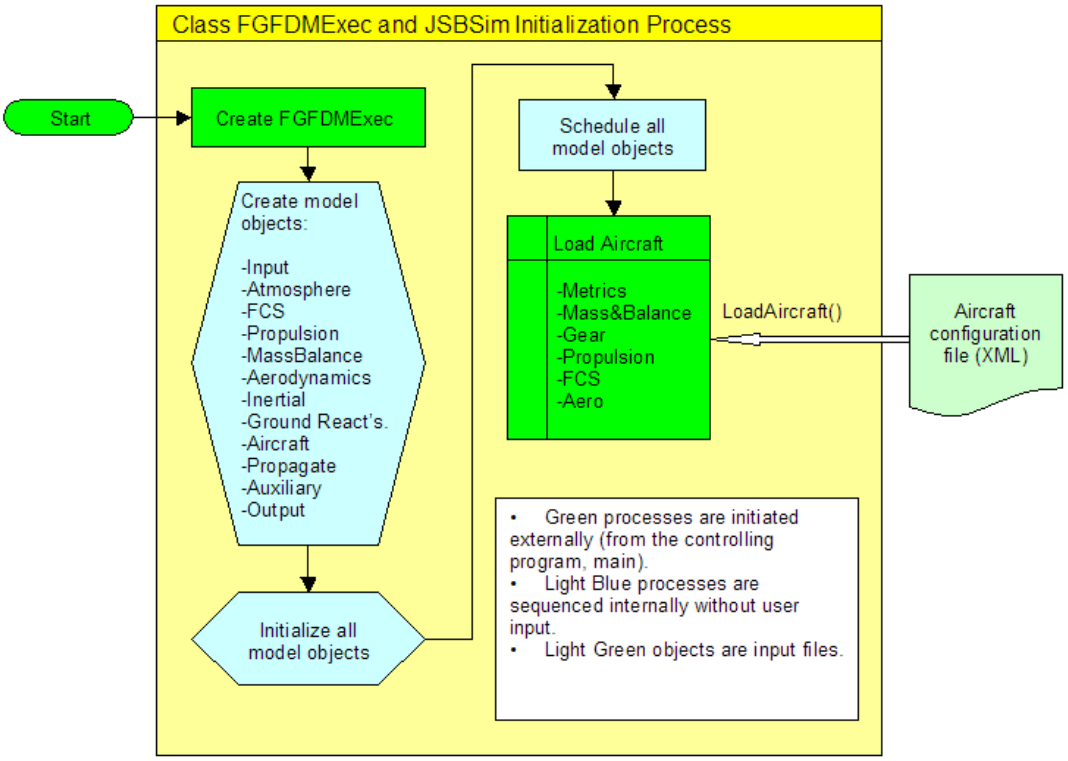


Figure 6 The JSBSim FGFDMExec class and initialization process.

Subsequent to the creation of the executive and loading of the model, initialization is performed. Initialization involves copying control inputs from the controlling application (FlightGear, or the main() application in the case of JSBSim standalone) into the appropriate JSBSim data storage locations, configuring it for the set of user-supplied initial conditions, and then copying state variables from JSBSim, back into the controlling application. The state variables are used to drive the instrument displays and to place the vehicle model in world space for visual rendering:

```
copy_to_JSBsim(); // copy control inputs to JSBSim
fdmex->RunIC(); // loop JSBSim once w/o integrating
copy_from_JSBsim(); // update the bus
```

Once initialization is complete, cyclic execution proceeds:

```
copy_to_JSBsim(); // copy control inputs to JSBSim
fdmex->Run(); // execute JSBSim
copy_from_JSBsim(); // update the bus
```

JSBSim can be used in a standalone mode by creating a compact stub program that effectively performs the same progression of steps as outlined above for the integrated version, but with two exceptions. First, the copy_to_JSBSim() and copy_from_JSBSim() functions are not used because the control inputs are handled directly by the scripting facilities and outputs are handled by the output (data logging) class. Second, the name of a script file can be supplied to the stub program. Scripting (see FGScript) provides a way to supply command inputs to the simulation:

```
FDMExec = new JSBSim::FGFDMExec();
FDMExec->LoadScript( ScriptName ); // the script loads the aircraft and
ICs
result = FDMExec->Run();
while (result) {            // cyclic execution
  result = FDMExec->Run();  // execute JSBSim
}
```

The standalone mode has been useful for verifying changes before committing updates to the source code repository. It is also useful for running sets of tests that reveal some aspects of simulated aircraft performance, such as range, time-to-climb, takeoff distance, etc.

2.4 Manager classes

2.5 Model classes

When it comes down to it, the job of a 6DoF flight dynamics model (FDM) is to answer these questions at the end of each cyclic frame:

- Where am I (latitude, longitude, altitude)?
- Which way am I oriented (pitch, yaw, roll)?

In accomplishing this, the FDM needs to know the forces and moments that are acting on the aircraft at any moment. As a prerequisite to this, the FDM must know the environmental characteristics, control inputs, etc.

JSBSim is comprised of classes that include specific model classes. Model classes are those classes that descend from the FGModel class, are executed in order during each cyclic frame, and which represent real physical objects, or which manage collections of real, physical objects. The FGPropulsion class, for instance, manages a collection of engines (FGEngine objects), the FGFCS class manages a collection of flight control components (FGFCSComponent objects), the FGGroundReactions class manages a collection of landing gear (FGLGear objects),and the FGAerodynamics class manages a list of aerodynamic forces and moments (FGFunction objects). The list of model classes is as follows (presented in order of execution):

- Input
- Atmosphere
- FCS (a collection manager class)
- Propulsion (a collection manager class)
- MassBalance
- Aerodynamics (a collection manager class)
- Inertial
- GroundReactions (a collection manager class)

- Aircraft
- Propagate
- Auxiliary
- Output

The Executive class (FGFDMExec) both stores its own private-access pointers to the instances of these classes, and manages sequential and cyclical execution of each class instance over each frame.

A description of the above mentioned Model class objects is presented in the following sections.

2.5.1 *Input*

The FGInput class implements the input capability for JSBSim, and is the first FGModel-derived object scheduled for execution in the frame.

After the model classes are instantiated and initialized, the aircraft files are loaded. As part of that process, the Load() method for FGInput is called - "'if"' there is an <input> element in the aircraft configuration file being loaded.

The FGInput::Load() method creates an instance of an FGfdmSocket class, passing the port number as an argument. This socket sets up the communication interface and begins listening so it is ready to accept a connection request from another process.

At runtime, the FGInput::Run() method is called. If data has been received through the socket, the data is processed, parsed and examined for the presence of known keywords. If those keywords are found, the appropriate action is taken.

2.5.2 *Atmosphere*

The Atmosphere model (FGAtmosphere class) models the 1976 standard atmosphere. It also models winds and turbulence, but that portion is experimental at this time.

The atmosphere class can either calculate the basic properties (pressure, density, temperature) or those parameters can be set from an external process. The atmospheric properties are calculated in the Run() method, executed each frame. If the atmospheric properties are supplied by an external source, the CalculateDerived() function is called. If the standard properties are calculated by the FGAtmosphere class itself (the normal approach), then the current altitude is first retrieved, and then the Calculate() function is called to determine the current pressure, density, and temperature. Subsequent to that, the CalculateDerived() function is called to compute additional values that depend on the standard atmospheric values. Among the additional parameters that are calculated within the CalculateDerived() function are speed of sound and density altitude.

The FGAtmosphere class also serves as a base class for other atmosphere models. There is a subdirectory, atmosphere/, under which can be found two additional atmosphere models, FGMSIS and FGMars. These are experimental, but FGMSIS has been tested and works well.

2.5.3 *FCS (Flight Control System)*

The FCS class (FGFCS) encapsulates the Flight Control System (FCS) functionality. This class owns and contains (manages) the list of FGFCSComponent-derived objects that define the control system for the loaded aircraft. FGFCS also provides function calls to read and set various

standard flight control parameters.

The constructor for FGFCS initializes various parameters and also calls bind(), which binds property names (text strings) with access methods (methods that get and set values, such as GetThrottleCmd() and SetThrottlePos() for getting and setting throttle values). The Property System is what allows property names to be specified in configuration files and accessed at run time.

When FGFCS::Load() is called, the section of the aircraft configuration file demarked by the <flight_control> element tags is parsed. The flight control definition can have any number of channels, each of which contains a sequence of FCS component definitions. When each of the component definitions is parsed, it creates an instance of the associated class object, and a pointer to that object is pushed onto a storage "array" - actually, a Standard Template Library vector container.

Three separate vector containers are used, because there are two flight control objects can be specified for any aircraft model: an autopilot, and a flight control system. The third vector may contain pointers to sensor objects. This allows sensor values to be calculated first, and then an entire set of flight control components (representing a set of control laws) may be executed before a second set of components is run.

At runtime, the list of components is executed in sequential order, with each component Run() method being called.

2.5.4 *Propulsion*

The propulsion class (FGPropulsion) manages a collection of zero or more engine objects of any type or combination of types. At initialization, the propulsion section of the configuration file is read and, if any engines are encountered, an instance of that engine is created and added to the engine array. At runtime, the propulsion class runs each engine in the list in sequence. Each engine calculates the engine parameters, fuel usage, etc. and finally returns the forces and moments that the engine exerts on the airframe. Fuel/propellants are subtracted from the tanks (FGTank).

Several different engine types are modeled in JSBSim:

- Piston engine (which supports turbocharging), FGPiston class
- Turbine engine, FGTurbine class
- Turboprop engine, FGTurboProp class
- Rocket engine, FGRocket class
- Electric engine, FGElectric class

The engine-specific parameters are defined in the engine configuration file in XML format. Each engine "owns" a thruster object that is used to turn the engine power into a force. The thruster may be a propeller, a nozzle, or a direct pass-through object.

2.5.5 *Mass & Balance*

Weight and balance modeling for JSBSim includes the calculation of moments of inertia, center of gravity, and mass quantities. The FGMassBalance class handles mass properties calculations.

At initialization, the mass_balance section of the configuration file is read and values

assigned. Point masses may be specified in the configuration file. If they are specified, they are added to an array of point mass objects. At runtime during each pass the total weight of the aircraft is calculated by summing the empty weight, the point mass weight[s] (if any), and the fuel weight. Similarly, the center of gravity and moments of inertia are calculated from the initial empty values, and the sum of the individual parts.

2.5.6 *Aerodynamics*

The aerodynamics class (FGAerodynamics), like the flight controls and ground reactions classes, is a collection manager class, storing individual force and moment definitions in any of six arrays. At initialization, the aerodynamics section of the configuration file is read, and individual force or moment values are read in. At runtime, each force or moment function definition is processed, resulting in a force or moment quantity for each of the axes defined.

2.5.7 *Inertial*

The FGInertial class is a very lightweight class that initializes the radius and reference acceleration values. At runtime, it calculates the current acceleration due to gravity, as a function of where the aircraft is at the moment (altitude).

2.5.8 *Ground Reactions*

One of the toughest parts of aircraft simulation is ground reactions. It is more difficult for JSBSim because we must model for the general case – there can be any number of bogies and they can be skids or wheels, steerable, or not, etc.

2.6 Math classes

2.7 Initialization

3. Incorporating JSBSim into Your Software

3.1 FlightGear

3.2 OpenEaagles

OpenEaagles is an open source C++ framework designed to support the rapid construction of virtual (human-in-the-loop) and constructive simulation applications. It has been used extensively to build DIS compliant distributed simulation systems. It is based upon EAAGLES (Extensible Architecture for the Analysis and Generation of Linked Simulations), a popular simulation framework developed and maintained by the U.S. Air Force to support a multitude of simulation activities.

As a framework, OpenEaagles provides a design pattern for how to construct a simulation. The goal is to provide an application developer a solid foundation so that robust, scalable, virtual, constructive, stand-alone, and distributed simulation applications can easily be built. It leverages modern object-oriented software design principles while incorporating fundamental real-time system design techniques to meet human interaction requirements.

By providing abstract representations of system components (that the object-oriented design philosophy promotes), multiple levels of fidelity can be easily intermixed and selected for optimal runtime performance. Abstract representations of systems allow a developer to tune the application to run efficiently so that human-in-the-loop interaction latency deadlines can be met. On the flip side, constructive-only simulation applications, that do not need to meet time-critical deadlines, can use models with even higher levels of fidelity.

The bulk of the development for the EAAGLES software has been performed at the Simulation and Analysis Facility (SIMAF) located at Wright Patterson AFB, Ohio. SIMAF participates in a number of distributed events each year. The vast majority of the distributed simulation software used in the facility has been “home grown” utilizing the EAAGLES framework. Applications built utilizing the framework include cockpits (F-16), ground control stations (Predator MQ-9), threat Integrated Air Defense Systems (IADS) and a futuristic battle manager.

OpenEaagles has been released as public domain code. This was done to encourage its use throughout the community. OpenEaagles closely tracks and incorporates new enhancements to EAAGLES, but does not include some functionality.

3.2.1 *Interfacing to JSBSim*

OpenEaagles is a framework that serves as a simulation design pattern as shown in Figure 1. Most of the elements that are needed to build a full-up simulation application exist within the framework, with one notable exception, the function main(). This function resides with the executable application, not the framework.

Typically main() is closely associated with the Station class which connects the simulation models to graphics and device I/O. The Station class also contains a Simulation class which

takes care of basic simulation activities such as managing a list of Players/Entities of interest. A Player can have many components and systems attached to it. Of particular interest related to JSBSim is a Players dynamics class. Each Player can have an associated dynamic model.

OpenEaagles provides a dynamic model that serves as an interface class, but unfortunately, specific flight models could not be included in the public domain release. Because JSBSim is a mature model it made sense for the OpenEaagles project to utilize it. Since JSBSim is also coded in C++, extending the included dynamics class to utilize JSBSim was fairly straightforward.

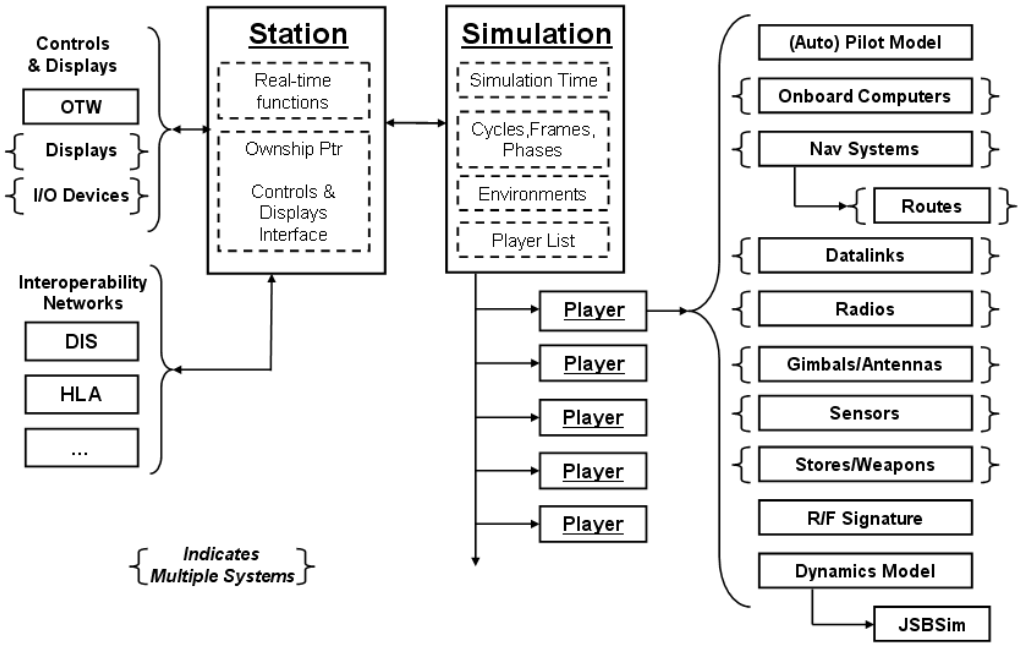


Figure 1. The OpenEaagles Simulation Framework

Since JSBSim can be configured to represent several aero propulsion systems, we have refined the OpenEaagles dynamics class to include features never before considered. For example, most of the aerodynamic models we have encountered, treat multiple engines as one. We did include the capability to model multiple engines in the dynamics class but it had never been tested. Attributes associated with engines other than jet turbines also needed to be considered. Specifically, the Propeller and TurboProp engines which requires additional controls (fuel mixture, prop pitch, etc).

3.2.2 *Calling JSBSim*

OpenEaagles is a frame-based system in which code is partitioned to support the development of real-time systems. As a frame-based system, delta time is passed as an argument to models so proper calculations involving time can be performed. Having models rely on delta time for calculation means the frequency of the entire system can change without having to change each and every model (so long as Nyquist rates are met). Additional time related information is recorded in terms of cycles (16 frames or sometimes called a major frame) and phases. Phases sequence the flow of data throughout a model. Four phases are currently defined:

- Dynamics -- update player or system dynamics including aerodynamic, propulsion, and

sensor positions (e.g., antennas, IR seekers).

- Transmit -- R/F emissions, which may contain datalink messages, are sent during this phase. The parameters for the R/F range equation, which include transmitter power, antenna pattern, gains and losses, are computed.
- Receive -- Incoming emissions are processed and filtered, and the detection reports or datalink messages are queued for processing.
- Process -- Used to process datalink messages, sensor detection reports and tracks, and to update state machines, on-board computers, shoot lists, guidance computers, autopilots or any other player or system decision logic.

JSBSim is currently being called in the dynamics phase at a rate of 50Hz.

3.2.3 *Building Applications*

Building an application with OpenEaagles consists of extending select classes of interest and writing a mainline. Typically, the mainline will call an OpenEaagles provided parser that will read an input file, and create the object hierarchy which is the simulation application. The input file describes in a simple language, the objects to create and the attributes to set.

Wherever a Player is defined, the creation of a JSBSimModel object (which was subclassed off of the DynamicsModel class) can be specified as the dynamics model of choice. The OpenEaagles inputs associated with the JSBSimModel class include the root directory for the JSBSim data files and a string that specifies the model of interest. With this information the interface class will call JSBSim, thus allowing it to process its own input files.

Thanks to C++ and the object-oriented nature of OpenEaagles and JSBSim, multiple instantiations of JSBSim (i.e. multiple Players) can easily be created and utilized within the same simulation application.

We see JSBSim being utilized as a high fidelity aero model driven by a human operator. The example fighter cockpit application, as shown in Figure 2, is a full-up simulation application where the pilot is controlling or flying one of the players in the player list. All stick and throttle inputs as well as graphic outputs are associated with the Player through the use of an “ownship” pointer that exists in the Station class. This connection is typically made by extending the Station class and customizing it for the application being built.

3.3 JSBSim and Python

4. Extending JSBSim

5. Building JSBSim

JSBSim can be built using a variety of environments, as described below. All of the environments described are downloadable, free of charge.

5.1 Building JSBSim using Microsoft Visual Studio 2008 Express

[One can download the Microsoft Visual Studio 2008 Express product at, http://www.microsoft.com/express/download/#webInstall.]

5.2 Building JSBSim under the Cygwin Environment

5.3 Building JSBSim under Cygwin with MinGW

From Wikipedia: "*MinGW (Minimalist GNU for Windows), formerly mingw32, is a native software port of the GNU Compiler Collection (GCC) to Microsoft Windows along with a set of freely distributable import libraries and header files for the Windows API. MinGW allows developers to create native Microsoft Windows applications.*" For more information on MinGW, visit the MinGW web site at www.mingw.org.

One can install the MinGW libraries when installing Cygwin. It's neither readily apparent nor intuitive to figure out, but one can compile a MinGW version of JSBSim by first configuring the makefiles using the following configure command (all on one line):

```
env CFLAGS="-mno-cygwin -O2" CPPFLAGS="-mno-cygwin –O2" \
    CXXFLAGS="-mno-cygwin -O2" LIBS="-lwsock32" \
    ./configure --target=i686-pc-mingw32
```

The first part of the command, above, sets compiler environment variables to use the "-mno-cygwin" compiler switch. A Library environment variable is also set that directs the linker to link with the mingw32 library. The library does not have any dependencies on the Cygwin "cygwin1.dll". This is one key reason that some may wish to build JSBSim using MinGW: the resulting executable is a native Windows application. The last part of the command, above, runs the configure script that builds the makefiles and sets up the build environment. Following that, one may simply run "make".

5.4 Building JSBSim under Linux

5.5 Building JSBSim on the Macintosh

5.6 Building JSBSim under IRIX

Section 3: Formulation Manual

1. Overview

2. Equations of Motion

The core purpose of a flight dynamics model is to propagate and track the path of a flying craft over the surface of the Earth (or another planet), given the forces and moments that act on the vehicle. We know the characteristics of the aircraft, and we know the characteristics of the planet (gravity, rotation rate, etc.). And it is expected that the reader is familiar with rigid body dynamics, where moving reference frames are involved. Still, putting all of the pieces together in a flight simulator can be overwhelming and tedious.

This section discusses aerospace vehicle equations of motion, as implemented in JSBSim::FGPropagate, using the quaternion, matrix, vector, and location math classes provided in JSBSim. Many of the equations listed in the following sections related to the rigid body equations of motion are referenced to Brian Stevens' and Frank Lewis' textbook, "Aircraft Control and Simulation", Second Edition (2003), and similarly in many other books.

The notation used in this reference manual is the same that Stevens and Lewis use:

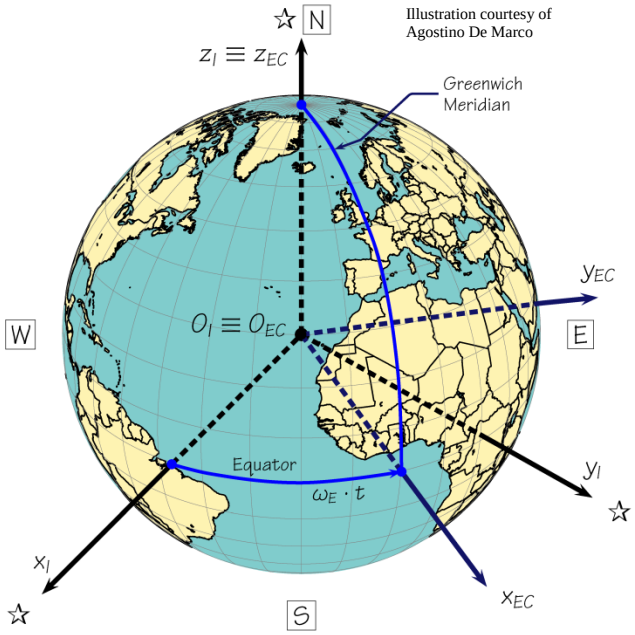
- The lower right subscript (e.g. $v_{CM/e}$) describes the object or frame relationship of the parameter. In the example given, $v_{CM/e}$, we refer to the velocity of the CM (center of mass) with respect to the ECEF frame.
- The upper right superscript (e.g. v^{b}) refers to a coordinate system. That is, it states which coordinate system the motion is expressed in.
- The left superscript specifies the frame in which a derivative is taken.

There are several ω contributions to consider in the equations that follow. The subscripts refer to four frames of interest:

v, or l — The vehicle NED (North, East, Down) frame, with the origin at the vehicle mass center, the X axis pointing North, the Y axis pointing east, and the Z axis positive downward by the right hand rule. This is very similar to the local and *nav* frames in JSBSim.

b — The body-fixed frame, with the X axis positive forwards out the nose of the aircraft, the Y axis positive out the right side of the aircraft, and the Z axis positive down.

e — The Earth Centered, Earth Fixed (ECEF) frame, with the Z axis coincident with the



spin axis and positive north, the X axis positive through 0 longitude and 0 latitude. This frame rotates with the Earth at a constant rate and does not translate.

i The Earth Centered Inertial (ECI) frame is fixed in celestial space with the Z axis positive north and coincident with the spin axis, with the X and Y axes located in the equatorial plane. At time T=0 the ECI and ECEF frames are coincident.

Examining the ω contributions:

$\omega_{b/v}$ The angular velocity vector of the body-fixed frame relative to the vehicle NED (local) frame. Note that when the vehicle is not manuevering (such as at-rest on a runway or launch pad), this angular velocity vector is zero.

$\omega_{b/e}$ The angular velocity vector of the body-fixed frame relative to the ECEF frame. Note that when the vehicle is at rest on the surface this angular velocity vector is zero.

$\omega_{b/i}$ The angular velocity vector of the body-fixed frame relative to the ECI frame.

$\omega_{v/e}$ The angular velocity vector of the vehicle NED (local) frame relative to the ECEF frame. Note that this angular velocity vector is independent of the vehicle rotational motion, and it is a function of the vehicle velocity relative to the ECEF frame – the speed at which the frame travels across the curved surface of the Earth – which of course implies it has an angular velocity of its own.

$\omega_{e/i}$ The angular velocity vector of the ECEF frame relative to the ECI frame. This is simply the angular velocity of the Earth about its spin axis.

We will see the use of the cross-product matrix in the equations that follow. The cross-product matrix (Ω) is the skew-symmetric matrix:

$$\begin{bmatrix} 0 & -R & Q \\ R & 0 & -P \\ -Q & P & 0 \end{bmatrix} \quad \text{(A)}$$

where P, Q, and R are the body rates relative to the inertial frame, expressed in the body frame.

Now, some discussion is needed of the variables used in JSBSim:

vUVW The body-fixed frame velocity vector of the vehicle relative to the ECEF frame, expressed in the body-fixed frame. With zero winds, and while the aircraft is at rest on the runway, this 3 element column vector will show zero velocity. A vehicle in a stable, equatorial, low Earth orbit will show a total *magnitude* for vUVW that is less than that required to maintain the orbit, but the total magnitude of the velocity relative to the inertial frame will show the true, inertial orbital velocity. In Stevens and Lewis notation, this is: $\mathbf{v}_{b/e}^{b}$

vPQR The body-fixed frame angular velocity vector of the vehicle relative to the ECEF frame, expressed in the body frame. While the aircraft is at rest on the runway, this 3 element column vector will show zero angular velocity. In Stevens and Lewis notation, this is: $\boldsymbol{\omega}_{b/e}^{b}$

vPQRi The body-fixed frame angular velocity vector of the vehicle relative to the ECI frame, expressed in the body frame. While the aircraft is at rest on the runway, this 3 element column vector will show a non-zero angular velocity. In Stevens and Lewis notation, this is: $\boldsymbol{\omega}_{b/i}^{b}$

vVel The body-fixed frame velocity vector of the vehicle relative to the ECEF frame, expressed in the vehicle-carried NED (local) frame. In Stevens and Lewis notation, this is: $\mathbf{v}_{b/e}^{v}$

JSBSim provides several rotation matrices for use in moving from one reference frame to another. For instance, the "Tec2b" matrix (represented in JSBSim through the use of an FGMatrix33 object) is the transformation matrix *from ECEF to body*. This matrix is represented in Stevens and Lewis as $C_{b/e}$ – or the transformation matrix relating the *body frame orientation with respect to the ECEF frame.*The transformation matrices provided in JSBSim are "owned" by two specific objects: the *location* object (an instance of an FGLocation class) and the *attitude* object (an instance of an FGQuaternion object). The location object stores the location of a vehicle in the ECEF frame. Therefore, the location object is aware of the relative orientation of the *geographic* reference frames relative to each other. The geographic frames are the inertial frame (ECI), the Earth-fixed frame (ECEF), and the local frame (NED). The attitude quaternion object stores the attitude of the vehicle body frame relative to the inertial frame. Using the transformation matrices from both frames, it is possible to determine the orientation of any of the frames relative to each other from the stored transformation matrices. The matrices stored in the location object are:

Ti2ec The ECI (inertial) to ECEF frame matrix (defines the orientation of the ECEF frame with respect to the ECI frame)

Tec2i The transpose of the Ti2ec matrix

Tec2l The ECI (inertial) to local (North, East, Down) frame matrix (defines the orientation of the local NED frame with respect to the ECI frame)

Tl2ec The transpose of the Tec2l matrix

Ti2l The inertial (ECI) to local (NED) frame transformation matrix stores the orientation of the local frame relative to the inertial frame.

Tl2i The transpose of the Ti2l matrix.

The transformation stored in the attitude quaternion is:

Ti2b The ECI (inertial) to body frame matrix (defines the orientation of the body frame with respect to the ECI frame). This is represented in Stevens and Lewis notation as Cb/i.

Tb2i The transpose of the Ti2b matrix

The transformations derived from the above transformations are:

Tec2b The ECEF to body frame matrix (defines the orientation of the body frame with respect to the ECEF frame)

Tb2ec The transpose of the Tec2b matrix

Tb2l The body to local (North, East, Down) frame matrix (defines the orientation of the local NED frame with respect to the body frame)

Tl2b The transpose of the Tb2l matrix

2.1 Translational Acceleration

The derivative of the translational velocity is found from $\Sigma\mathbf{F}=m\mathbf{a}$ in the body-fixed frame, and employing what is sometimes referred to as the Equation of Coriolis. [See "Aircraft Control and Simulation" (Stevens & Lewis, 2003) for the full set of derivations.] The equation for the derivative

of velocity is written (rearranging **ΣF**=m**a** and recalling that we are solving for acceleration in a rotating frame):

$$ {}^{b}\dot{\boldsymbol{v}}^{b}_{CM/e} = \frac{\boldsymbol{F}^{b}_{A,T}}{m} - \left(\Omega^{b}_{b/i} + \Omega^{b}_{e/i}\right)\mathbf{v}^{b}_{CM/e} + \mathrm{C}_{b/i}\mathbf{g}^{i} \qquad \text{(1.5-16d)} $$

We can look at this equation another way (with all terms below expressed in the body coordinate system, so the "*b*" superscript is omitted):

$$ {}^{b}\dot{\boldsymbol{v}}_{CM/e} = \frac{\boldsymbol{F}_{A,T}}{m} - \left(\omega_{b/i} + \omega_{e/i}\right) \times \mathbf{v}_{CM/e} + \mathbf{g} \qquad \text{(1.5-12)} $$

Splitting up the angular velocities (which are additive):

$$ {}^{b}\dot{\boldsymbol{v}}_{CM/e} = \frac{\boldsymbol{F}_{A,T}}{m} - \left(\omega_{b/v} + \omega_{v/e} + 2\omega_{e/i}\right) \times \mathbf{v}_{CM/e} + \mathbf{g} \qquad \text{(1.5-13)} $$

For our purposes in JSBSim – since we track and propagete the angular velocity of the vehicle with respect to the ECEF frame ($\omega_{b/e}$) – we modify 1.5-13 by combining the first two angular velocities on the right side of 1.5-13 to get:

$$ {}^{b}\dot{\boldsymbol{v}}_{CM/e} = \frac{\boldsymbol{F}_{A,T}}{m} - \left(\omega_{b/e} + 2\omega_{e/i}\right) \times \mathbf{v}_{CM/e} + \mathbf{g} \qquad \text{(1.5-13 modified)} $$

In JSBSim, this is effectively what we have now:

```
void FGPropagate::CalculateUVWdot(void)
{
  double mass = MassBalance->GetMass();                    // mass
  const FGColumnVector3& vForces = Aircraft->GetForces(); // forces

  const FGColumnVector3
    vGravAccel( 0.0, 0.0, Inertial->GetGAccel(VehicleRadius) );

  // Begin to compute body frame accelerations based on
  // the current body forces
  vUVWdot = vForces/mass - VState.vPQR * VState.vUVW;

  // Include Coriolis acceleration.
  // vOmega is the Earth angular rate expressed in the inertial frame,
  // so it has to be transformed to the body frame. More completely,
  // vOmega is the rate of the ECEF frame relative to the Inertial
  // frame (ECI), expressed in the Inertial frame.
  vUVWdot -= 2.0 * (Ti2b *vOmega) * VState.vUVW;

  // Include Centrifugal acceleration.
  if (!GroundReactions->GetWOW()) {
    vUVWdot -= Ti2b*(vOmega*(vOmega*(Tec2i*VState.vLocation)));
  }

  // Include Gravitation accel
  vUVWdot += Tl2b*vGravAccel;
}
```

2.2 Angular Acceleration

Angular acceleration is given in Stevens and Lewis as follows

$$ {}^{b}\dot{\omega}^{b}_{b/i} = (J^{b})^{-1}\left[\boldsymbol{M}^{\boldsymbol{b}}_{A,T} - \Omega^{\mathrm{b}}_{\mathrm{b/i}} J^{b}\omega^{b}_{b/i}\right] \qquad \text{(1.5-16e)} $$

This reads, *the derivative of the angular velocity of the body relative to the inertial frame and taken in the body-fixed frame is determined by the non-gravitational moments, the cross-product matrix, the inertia matrix, and the angular velocity of the body relative to the inertial frame, and measured in the body system*.

The angular velocity of the body (relative to the inertial frame) is physically the body rate of the aircraft *plus* the angular velocity of the Earth as it spins.

The calculations in JSBSim are as follows:

```
void FGPropagate::CalculatePQRdot(void)
{
  const FGColumnVector3& vMoments = Aircraft->GetMoments(); // moments
  const FGMatrix33& J = MassBalance->GetJ();       // inertia matrix
  const FGMatrix33& Jinv = MassBalance->GetJinv(); // inverse matrix

  // Compute the body frame rotational accelerations based on the
  // current body moments and the total inertial angular velocity
  // expressed in the body frame.
  vPQRdot = Jinv*(vMoments - vPQRi*(J*vPQRi));
}
```

J, and ***Jinv*** are the inertia and inverse inertia matrices
vOmega is the angular rotation rate vector of the Earth
vPQRi is the inertial body rate measured in the body-fixed frame

Note that in JSBSim, the use of the "*" operator has been overloaded to often act as a cross product operator when needed.

In the case of JSBSim, the multiplication of the vPQRi vector with the (J*vPQRi) term yields the expected result. This is due to the way that a vector is multiplied against another vector through the use of overloaded operators in the vector class. For readability and to be most accurate, however, it might be better to create an actual cross-product matrix and use that, which would create a closer relationship between the JSBSim code and popular reference material. However, it would introduce some inefficiency in execution.

The result of this calculation is the angular acceleration of the vehicle relative to the inertial frame (see eqn. 1.5-16e). What we eventually want is the angular velocity of the vehicle relative to the ECEF frame – that's what vPQR represents (expressed in the body frame). We will use the fact that angular velocities are additive to look at the integration of angular acceleration in understanding the JSBSim algorithm.

Starting with the integration of angular acceleration (where the t0 and t1 left hand subscripts represent the values at an initial timestep, and the subsequent timestep Δt seconds later):

$$ {}_{t1}\omega^{b}_{b/i} = {}_{t0}\omega^{b}_{b/i} + {}^{b}\dot{\omega}^{b}_{b/i} \cdot \Delta t \qquad \text{(B)} $$

Equation B, above, essentially says that the new angular velocity is calculated by summing the old angular velocity (at t0) and the incremental change – the angular acceleration of the body frame relative to the inertial frame (times the time-step duration).

In JSBSim we track and propagate the angular velocity relative to the ECEF frame. So, we need to know how to use the angular acceleration to propagate the angular rate.

Since angular velocities are additive within the same system, the angular velocity of the body

frame relative to the inertial frame can be expressed as the sum of two other angular velocities:

$$\omega_{b/i}^{b} = \omega_{b/e}^{b} + \omega_{e/i}^{b} \qquad \text{(C)}$$

The above equation is used in JSBSim to calculate the *inertial* body rates:

```
vPQRi = VState.vPQR + Ti2b*vOmega;
```

It's important to understand that vOmega in the above code is the Earth angular rate expressed in the *inertial* frame. We usually need it to be expressed in the body frame, so we must transform it:

$$\omega_{e/i}^{b} = T_{i/b}\omega_{e/i}^{i} \qquad \text{(D)}$$

Replacing the angular *velocities* in (B) with the angular velocity on the right side in (C) we have:

$${}_{t1}\omega_{b/e}^{b} + {}_{t1}\omega_{e/i}^{b} = {}_{t0}\omega_{b/e}^{b} + {}_{t0}\omega_{e/i}^{b} + {}^{b}\dot{\omega}_{b/i}^{b} \cdot \Delta t \qquad \text{(E)}$$

Since the angular velocity of the Earth relative to the inertial frame is constant, that term can be removed from each side of equation (E), leaving:

$${}_{t1}\omega_{b/e}^{b} = {}_{t0}\omega_{b/e}^{b} + {}^{b}\dot{\omega}_{b/i}^{b} \cdot \Delta t \qquad \text{(F)}$$

In other words, the incremental change in angular velocity of the body frame relative to the inertial frame (and expressed in the body frame) can be used to propagate the angular velocity of the body relative to the ECEF frame (expressed in the body frame). This is due to the fact that the Earth frame (ECEF) does not accelerate relative to the inertial (ECI) frame.

Equation F, above, is represented in JSBSim code as follows (for the Euler integrator):

```
case eRectEuler:       VState.vPQR += dt*vPQRdot;
  break;
```

2.3 Translational Velocity

The translational position derivative (velocity) in JSBSim is found from integrating the acceleration vector. The position vector derivative is given in Stevens and Lewis as:

$${}^{i}\dot{\mathbf{p}}_{CM/O}^{i} = \mathrm{C}_{i/b}\mathbf{v}_{CM/e}^{b} + \Omega_{e/i}^{i}\mathbf{p}_{CM/O}^{i} \qquad \text{(1.5-16c, Stevens and Lewis)}$$

Examining this equation, we can see that this is essentially the equation of Coriolis, which calculates the inertial frame vehicle velocity as the sum of the vehicle velocity in the ECEF frame with the velocity represented by the angular motion of the ECEF frame with respect to the inertial frame. In the program code below, vLocationDot represents the first product on the right side of equation 1.5-16 (Stevens and Lewis) – that is, vLocationDot represents the velocity of the vehicle relative to the ECEF frame (expressed in the body system), but transformed to be expressed in the inertial frame. Since vLocation (as a class) represents the vehicle position relative to the ECEF frame, vLocationDot (vehicle velocity) must also be expressed in the ECEF frame.

The term *vInertialVelocity* represents the "$\dot{\mathbf{p}}$" term on the left side of equation 1.5-16c (Stevens and Lewis). In the code below, the ECEF velocity term, *vLocationDot* is transformed to the inertial frame, and summed with the construct that is analogous to the second product (right side) of equation 1.5-16c (Stevens and Lewis).

```
void FGPropagate::CalculateLocationdot(void)
{
```

```
  // Transform the vehicle velocity relative to the ECEF frame,
  // expressed in the body frame, to be expressed in the ECEF frame.
  vLocationDot = Tb2ec * VState.vUVW;

  // Now, transform the velocity vector of the body relative to the
  // origin (Earth center) to be expressed in the inertial frame, and
  // add the vehicle velocity contribution due to the rotation of the
  // planet. The above velocity is only relative to the rotating ECEF
  // frame.
  // Reference: See Stevens and Lewis, "Aircraft Control and
  // Simulation", Second edition (2004), eqn 1.5-16c (page 50)

  vInertialVelocity = Tec2i * vLocationDot
                      + (vOmega * (Tec2i * VState.vLocation));
}
```

2.4 Angular Velocity

The angular position derivative (angular velocity) is found as described in (Stevens & Lewis, 2003). We can define a quaternion that describes the orientation of one frame relative to another through a set of Euler rotations (ψ, θ, φ – or a Z axis rotation, then a Y axis rotation, then one about the X axis). For instance, we can define a quaternion that gives us the orientation of the body frame (F_b) relative to the geographic local, vehicle-carried, NED frame (F_v):

q[0] = cos(φ/2)·cos(θ/2) ·cos(ψ/2) + sin(φ/2) ·sin(θ/2) ·sin(ψ/2)
q[1] = sin(φ/2) ·cos(θ/2) ·cos(ψ/2) - cos(φ/2) ·sin(θ/2) ·sin(ψ/2)
q[2] = cos(φ/2) ·sin(θ/2) ·cos(ψ/2) + sin(φ/2) ·cos(θ/2) ·sin(ψ/2)
q[3] = cos(φ/2) ·cos(θ/2) ·sin(ψ/2) - sin(φ/2) ·sin(θ/2) ·cos(ψ/2)

Leaving the derivation of the quaternion derivative to Stevens and Lewis (and others), and knowing the angular velocity ω of the body with respect to the reference frame (the body frame with respect to the ECEF frame, in our case) we assert that the derivative of the quaternion, $\dot{q}$, is defined as follows:

$$\dot{q} = \frac{1}{2}\begin{bmatrix} 0 & -\omega^T \\ \omega & -\Omega \end{bmatrix}\begin{bmatrix} q_0 \\ \mathbf{q} \end{bmatrix}$$

or, expanded,

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 0 & -P & -Q & -R \\ P & 0 & R & -Q \\ Q & -R & 0 & P \\ R & Q & -P & 0 \end{bmatrix}\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

Then, the quaternion can be propagated by integrating it, for example,

$q_{n+1} = q_n + \dot{q}dt$

The calculation of the quaternion derivative is performed (as are other quaternion operations) in the class FGQuaternion:

```
FGQuaternion FGQuaternion::GetQDot(const FGColumnVector3& PQR) const
{
  double norm = Magnitude();
```

```
  if (norm == 0.0) return FGQuaternion::zero();

  double rnorm = 1.0/norm;

  FGQuaternion QDot;
  QDot(1) = -0.5*(Entry(2)*PQR(eP) + Entry(3)*PQR(eQ)
            + Entry(4)*PQR(eR));
  QDot(2) =  0.5*(Entry(1)*PQR(eP) + Entry(3)*PQR(eR)
            - Entry(4)*PQR(eQ));
  QDot(3) =  0.5*(Entry(1)*PQR(eQ) + Entry(4)*PQR(eP)
            - Entry(2)*PQR(eR));
  QDot(4) =  0.5*(Entry(1)*PQR(eR) + Entry(2)*PQR(eQ)
            - Entry(3)*PQR(eP));
  return rnorm*QDot;
}
```

It should be noted that we can do this for any two frames if we know the initial orientation of one *body* frame with respect to the reference frame, and know the angular velocity of the body frame with respect to the reference frame.

In the specific case of JSBSim, the initial orientation of the vehicle is given in an ordered sequence of Euler angles. The standard aerospace sequence is used (yaw, then pitch, then roll). The Euler angles define the orientation of the vehicle relative to the local NED frame. The initialization of the vehicle state quaternion (owned by FGPropagate and contained in the VState structure) is handled from within the FGPropagate::SetInitialState() function, which passes the Euler angles to the quaternion constructor, which itself subsequently creates the quaternion and initializes the four quaternion elements using the equations for q[0], q[1], q[2], q[3] previously presented.

At runtime, the quaternion derivative is calculated using the angular velocity of the body relative to the local NED frame:

$$\omega_{v/e} = \omega_{b/e} - \omega_{b/v}$$

In the code, below, vOmegaLocal is the calculated angular velocity of the local NED frame, F_v.

```
void FGPropagate::CalculateQuatdot(void)
{
  FGColumnVector3 vOmegaLocal(
                   radInv*vVel(eEast),
                  -radInv*vVel(eNorth),
                  -radInv*vVel(eEast)*VState.vLocation.GetTanLatitude()
  );

  // Compute quaternion orientation derivative on current body rates
  vQtrndot = VState.vQtrn.GetQDot( VState.vPQR - Tl2b*vOmegaLocal);
}
```

Once the quaternion derivative is calculated it is integrated.

Section 4: Case Studies

1. Overview

2. Simple ball

We'll start with the simplest of all possible cases, modeling a ball flying through the air. This ball will have no control or propulsion systems, will exhibit only aerodynamic drag, and will have a single contact point. The definition of the ball is as follows:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
href="http://jsbsim.sourceforge.net/JSBSim.xsl"?>
<fdm_config name="BALL" version="2.0" release="BETA"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:noNamespaceSchemaLocation=
                          "http://jsbsim.sourceforge.net/JSBSim.xsd">

    <fileheader>
        <author> Joe Public </author>
        <filecreationdate> 2004-06-01 </filecreationdate>
        <version> Version 1.0 </version>
        <description> Ball vehicle test file </description>
    </fileheader>

    <metrics>
        <wingarea unit="FT2"> 1 </wingarea>
        <wingspan unit="FT"> 1 </wingspan>
        <chord unit="FT"> 1 </chord>
        <htailarea unit="FT2"> 0 </htailarea>
        <htailarm unit="FT"> 0 </htailarm>
        <vtailarea unit="FT2"> 0 </vtailarea>
        <vtailarm unit="FT"> 0 </vtailarm>
        <location name="AERORP" unit="IN">
            <x> 0 </x>
            <y> 0 </y>
            <z> 0 </z>
        </location>
    </metrics>

    <mass_balance>
        <ixx unit="SLUG*FT2"> 10 </ixx>
        <iyy unit="SLUG*FT2"> 10 </iyy>
        <izz unit="SLUG*FT2"> 10 </izz>
        <ixy unit="SLUG*FT2"> -0 </ixy>
        <ixz unit="SLUG*FT2"> -0 </ixz>
        <iyz unit="SLUG*FT2"> -0 </iyz>
        <emptywt unit="LBS"> 50 </emptywt>
        <location name="CG" unit="IN">
            <x> 0 </x>
            <y> 0 </y>
            <z> 0 </z>
        </location>
```

```
    </mass_balance>

    <ground_reactions>
        <contact type="BOGEY" name="NOSE_CONTACT">
            <location unit="IN">
                <x> 0 </x>
                <y> 0 </y>
                <z> 0 </z>
            </location>
            <static_friction> 0 </static_friction>
            <dynamic_friction> 0 </dynamic_friction>
            <rolling_friction> 0 </rolling_friction>
            <spring_coeff unit="LBS/FT"> 10000 </spring_coeff>
            <damping_coeff unit="LBS/FT/SEC"> 200000 </damping_coeff>
            <max_steer unit="DEG"> 0.0 </max_steer>
            <brake_group> NONE </brake_group>
            <retractable>0</retractable>
        </contact>
    </ground_reactions>

    <propulsion/>

    <flight_control name="FGFCS"/>

    <aerodynamics>
        <axis name="DRAG">
            <function name="aero/coefficient/CD">
                <description>Drag</description>
                <product>
                    <property>aero/qbar-psf</property>
                    <property>metrics/Sw-sqft</property>
                    <value>0.0001</value>
                </product>
            </function>
        </axis>
    </aerodynamics>

</fdm_config>
```

This file represents a ball with a reference area of 1 square foot, a lateral reference length ("wingspan") of 1 foot, and a longitudinal reference length ("chord") of 1 foot.

The ball description is not the only file that we need, however. We also have to tell the simulation where to begin – what the initial conditions of the simulation run will be. We can define initial values for various parameters in an initialization file. An example of such a file is as follows:

```
<?xml version="1.0"?>
<initialize name="Cannonball example">
  <!—
    This file sets up a cannonball example.
  -->
  <ubody unit="FT/SEC">  500.0  </ubody>
  <latitude unit="DEG">    0.0  </latitude>
  <longitude unit="DEG"> -90.0  </longitude>
  <theta unit="DEG">      60.0  </theta>
  <psi unit="DEG">        40.0  </psi>
  <altitude unit="FT">     1.0  </altitude>
```

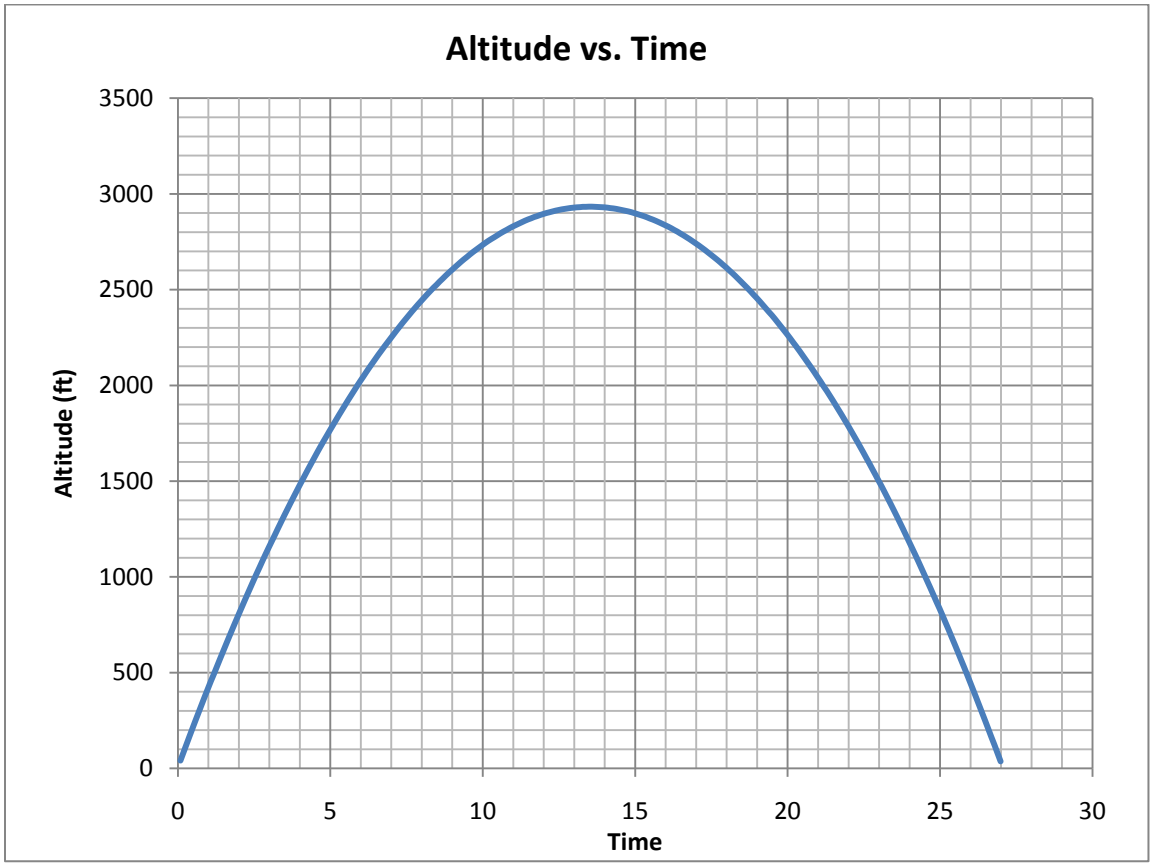
```
</initialize>
```

The file above sets the ball up with an initial velocity of 500 feet per second along its own X axis, and an angle of 60 degrees above the horizon, at a heading of 40 degrees east of north. The initial location is on the equator at 90 degrees west longitude.

A simulation run can be made using this command line (assuming that the jsbsim executable is in your path):

```
jsbsim --aircraft=ball --initfile=cannonball_init
```

The ".xml" extension does not need to be given, nor should the full path to the files. The chart above shows the altitude versus time curve that results from this run.



3. Ball with parachute

We can go one step further over the previous example and add a parachute. This is made possible using the external reactions capability. The ball example is modified by adding the parachute as follows (see the external_reactions element):

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="http://jsbsim.sourceforge.net/JSBSim.xsl"?>
<fdm_config name="BALL" version="2.0" release="BETA"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:noNamespaceSchemaLocation=
                            "http://jsbsim.sourceforge.net/JSBSim.xsd">

    <fileheader>
        <author> Jon Berndt </author>
        <filecreationdate> 2004-06-01 </filecreationdate>
        <version> Version 1.0 </version>
        <description> Test file </description>
    </fileheader>

    <metrics>
        <wingarea unit="FT2"> 1 </wingarea>
        <wingspan unit="FT"> 1 </wingspan>
        <chord unit="FT"> 1 </chord>
        <htailarea unit="FT2"> 0 </htailarea>
        <htailarm unit="FT"> 0 </htailarm>
        <vtailarea unit="FT2"> 0 </vtailarea>
        <vtailarm unit="FT"> 0 </vtailarm>
        <location name="AERORP" unit="IN">
            <x> 0 </x>
            <y> 0 </y>
            <z> 0 </z>
        </location>
    </metrics>

    <mass_balance>
        <ixx unit="SLUG*FT2"> 10 </ixx>
        <iyy unit="SLUG*FT2"> 10 </iyy>
        <izz unit="SLUG*FT2"> 10 </izz>
        <emptywt unit="LBS"> 50 </emptywt>
        <location name="CG" unit="IN">
            <x> 0 </x>
            <y> 0 </y>
            <z> 0 </z>
        </location>
    </mass_balance>

    <ground_reactions>
```

```
        <contact type="BOGEY" name="CONTACT">
            <location unit="IN">
                <x> 0 </x>
                <y> 0 </y>
                <z> 0 </z>
            </location>
            <static_friction> 0 </static_friction>
            <dynamic_friction> 0 </dynamic_friction>
            <rolling_friction> 0 </rolling_friction>
            <spring_coeff unit="LBS/FT"> 10000 </spring_coeff>
            <damping_coeff unit="LBS/FT/SEC"> 200000 </damping_coeff>
            <max_steer unit="DEG"> 0.0 </max_steer>
            <brake_group> NONE </brake_group>
            <retractable>0</retractable>
        </contact>
    </ground_reactions>

    <external_reactions>

            <!—
            "Declare" the reefing term that will be set
            in the script.
            -->

      <property> fcs/parachute_reef_pos_norm </property>

      <force name="parachute" frame="WIND">

        <function>
          <product>
            <property> aero/qbar-psf </property>
            <property> fcs/parachute_reef_pos_norm </property>
            <value> 1.0 </value>  <!-- Full drag coefficient -->
            <value> 20.0 </value> <!-- Full parachute area -->
          </product>
        </function>

            <!—
            The location below is in structural frame (x positive
            aft), so this location describes a point 1 foot aft
            of the origin. In this case, the origin is the center.
            -->

        <location unit="FT">
          <x>1</x>
          <y>0</y>
          <z>0</z>
        </location>

            <!—
            The direction describes a unit vector. In this case, since
            the selected frame is the WIND frame, the "-1" x component
            describes a direction exactly opposite of the direction
            into the wind vector. That is, the direction specified
            below is the direction that the drag force acts in.
            -->
```

```
        <direction>
          <x>-1</x>
          <y>0</y>
          <z>0</z>
        </direction>

      </force>
    </external_reactions>

    <propulsion/>

    <flight_control name="FGFCS"/>

    <aerodynamics>
        <axis name="DRAG">
            <function name="aero/coefficient/CD">
                <description>Drag</description>
                <product>
                    <property>aero/qbar-psf</property>
                    <property>metrics/Sw-sqft</property>
                    <value>0.0001</value>
                </product>
            </function>
        </axis>
    </aerodynamics>

    <output name="JSBBallOut.csv" type="CSV" rate="10">
        <property> gear/unit[0]/WOW </property>
        <rates> ON </rates>
        <velocities> ON </velocities>
        <forces> ON </forces>
        <moments> ON </moments>
        <position> ON </position>
    </output>
</fdm_config>
```

The above configuration describes a ball that has a parachute that pops out attached to the "back" end. The force magnitude is defined by the function. The run script used is shown following. Note the deployment of the parachute in reefing stages. The initial conditions are different from those in the previous case study:

```
<?xml version="1.0"?>
<initialize name="reset00">
  <!--
    This file sets up the ball initial conditions.
  -->
  <ubody unit="FT/SEC">  500.0  </ubody>
  <vbody unit="FT/SEC">    0.0  </vbody>
  <wbody unit="FT/SEC">    0.0  </wbody>
  <latitude unit="DEG">    0.0  </latitude>
  <longitude unit="DEG">  90.0  </longitude>
  <phi unit="DEG">         0.0  </phi>
  <theta unit="DEG">      40.0  </theta>
  <psi unit="DEG">        40.0  </psi>
  <altitude unit="FT"> 10000.0  </altitude>
  <winddir unit="DEG">     0.0  </winddir>
```

```
  <vwind unit="FT/SEC">    0.0  </vwind>
</initialize>
```

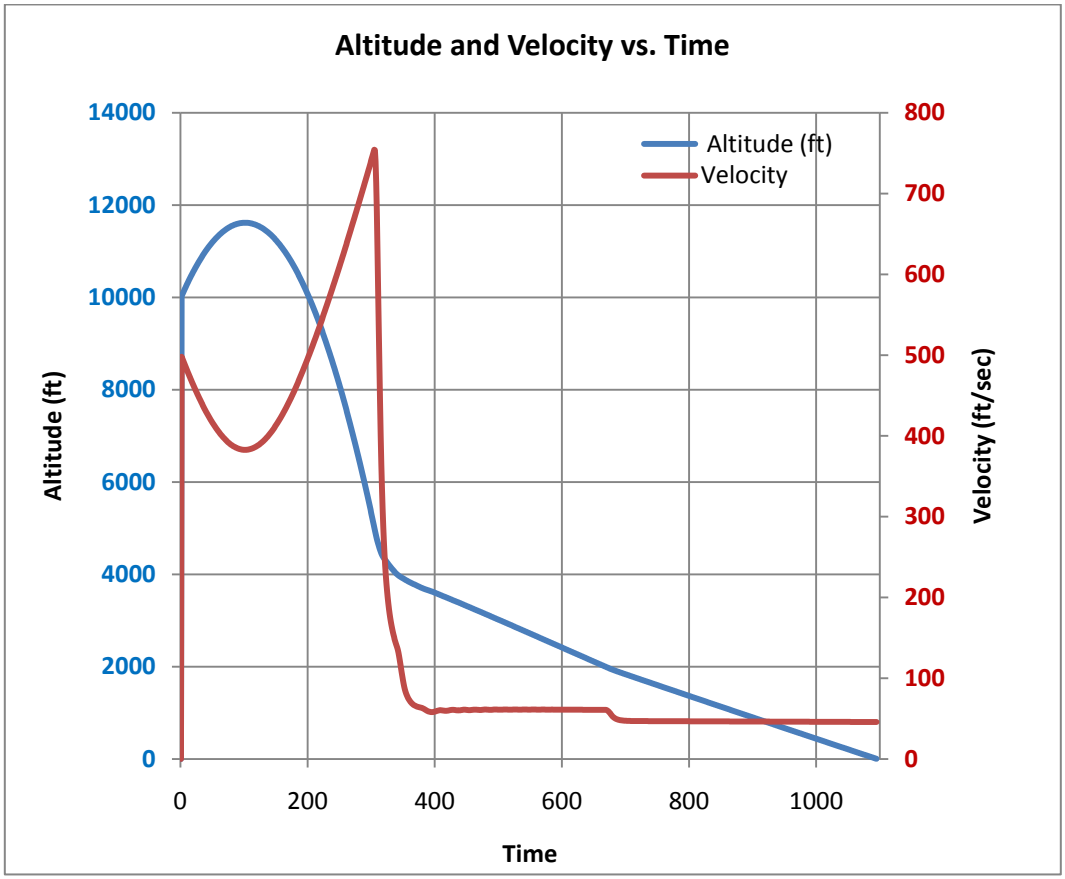
In words, the script below will,

- Deploy the parachute, reefed at 20%, when the ball descends below 5000 feet,
- Disreef the parachute to 60% open when the ball descends below 4000 feet,
- Disreef the parachute to 100% (fully open) when the ball descends below 2000 feet.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"
    href="http://jsbsim.sf.net/JSBSimScript.xsl"?>

<runscript xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xsi:noNamespaceSchemaLocation=
   "http://jsbsim.sf.net/JSBSimScript.xsd" name="ball test">

  <description>Simplest of all tests</description>

  <use aircraft="ball" initialize="reset01"/>
  <run start="0.0" end="600" dt="0.008333">

    <event name="Reef 1">
      <description> Test of parachute reefing </description>
      <condition>
        <test>
          position/h-agl-ft lt 5000
        </test>
      </condition>
      <set name="fcs/parachute_reef_pos_norm" value="0.2"
                                    action="FG_RAMP" tc="1.0"/>
      <notify>
        <property>sim-time-sec</property>
      </notify>
    </event>

    <event name="Reef 2">
      <description> Test of parachute reefing </description>
      <condition>
        <test>
          position/h-agl-ft lt 4000
        </test>
      </condition>
      <set name="fcs/parachute_reef_pos_norm" value="0.6"
                                    action="FG_RAMP" tc="1.0"/>
      <notify>
        <property>sim-time-sec</property>
      </notify>
    </event>

    <event name="Reef Final">
      <description> Test of parachute reefing </description>
      <condition>
        <test>
          position/h-agl-ft lt 2000
        </test>
      </condition>
```

```
      <set name="fcs/parachute_reef_pos_norm" value="1.0"
                                      action="FG_RAMP" tc="1.0"/>
      <notify>
        <property>sim-time-sec</property>
      </notify>
    </event>

    <event name="Terminate">
      <description> End condition </description>
      <condition>
        <test>
          gear/unit[0]/WOW eq 1
        </test>
      </condition>
      <set name="simulation/terminate" value="1.0"/>
      <notify>
        <property>sim-time-sec</property>
      </notify>
    </event>

  </run>
</runscript>
```

The plot of velocity and altitude versus time is shown:



4. Piston Aircraft with Autopilot and Scripting

For a more involved example, we'll look at modeling a wing leveler autopilot and the flight control roll model for a small general aviation aircraft. Following that, we'll also examine a heading hold autopilot. We'll refer to this aircraft model as the c172x. The entire XML file that describes the c172x will not be shown here. See the JSBSim distribution for that.

4.1 An Automatic Wing Leveler

We will assume that we have an aircraft model already at our disposal: the c172x, and that the mass properties, aerodynamics, propulsion, etc. are already determined. We begin this case study by asking the question, how can we give the aircraft flight model the capability to hold a wings-level attitude during a scripted flight? We know that if we want to hold a roll attitude of zero (wings-level), we'll have to generate a roll command that acts to remove the non-zero roll angle and return to wings-level flight. Such an autopilot will have to control the ailerons.

The autopilot control components can either go in the <autopilot> element in the aircraft configuration file, or they can be put in a separate file, which permits easier reuse. This time, we'll put the component descriptions in the aircraft configuration file, in the <autopilot> element. The relevant part of the configuration file is as follows:

```
<autopilot name="C172 A/P">

  <channel name="Roll">
    … component definitions …
  </channel>

</autopilot>

<flight_control name="C172 FCS">

  <channel name="roll">
    … component definitions …
  </channel>

</flight_control>
```

Let's make one more assumption at this point: let's assume that the normal roll flight control channel is already constructed, and that the inputs to the roll channel are:

- Roll trim
- Roll command (from the pilot joystick)
- Roll command (from the autopilot)

The "normal roll flight control" channel is the set of components that takes the above inputs (which range from -1 to +1 individually and as a sum total), processes the signal as needed, and

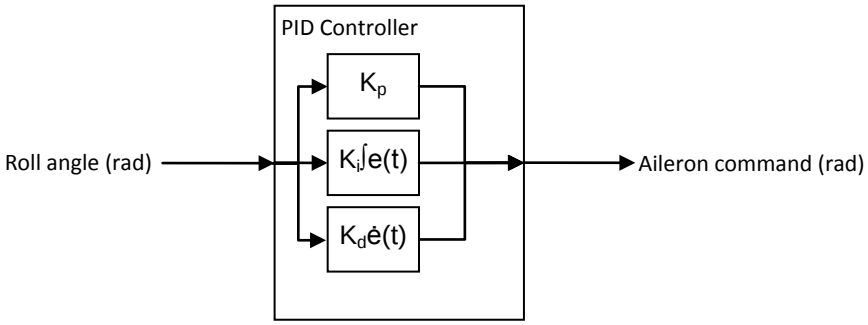
outputs an aileron command.

We will use a PID controller to process the roll angle error and output an aileron command that will keep the wings level. What is a PID controller? From Wikipedia:

A proportional–integral–derivative controller (PID controller) is a generic control loop feedback mechanism widely used in industrial control systems. A PID controller attempts to correct the error between a measured process variable and a desired setpoint by calculating and then outputting a corrective action that can adjust the process accordingly.

The PID controller calculation (algorithm) involves three separate parameters; the Proportional, the Integral and Derivative values. The Proportional value determines the reaction to the current error, the Integral determines the reaction based on the sum of recent errors and the Derivative determines the reaction to the rate at which the error has been changing. The weighted sum of these three actions is used to adjust the process via a control element such as the position of a control valve or the power supply of a heating element.

A diagram of our wing leveler control law is shown below:



In JSBSim-ML, this looks like (with example gains thrown in):

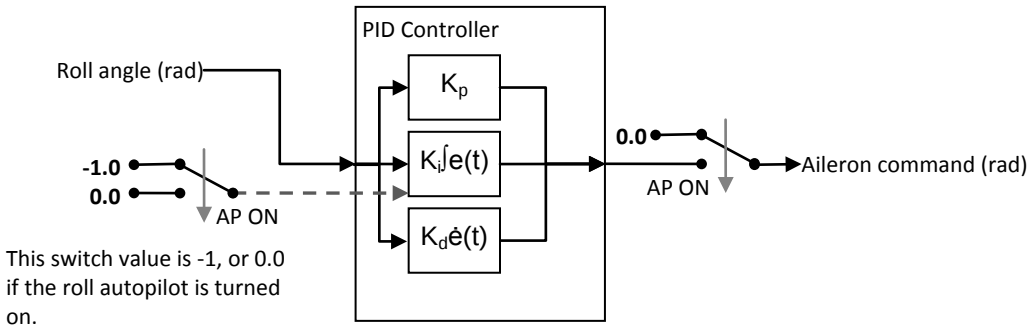
```
<pid name="fcs/roll-ap-error-pid">
  <input> attitude/phi-rad </input>
  <kp> 50.0 </kp>
  <ki>  5.0 </ki>
  <kd> 17.0 </kd>
  <output> ap/aileron_cmd </output>
</pid>
```

It doesn't get much simpler than this, does it? However, there is more that needs to be done. The autopilot of course is not going to be on at all times. And any time the PID controller is getting a non-zero input and the integrator gain is non-zero, the integrator output will be building up. So, it's important to make sure that the integrator is properly used. We can modify the above controller to use a switch that controls the integrator state, and another switch to actually *use* the output.

This is a good place to explain the wind-up protection in the integrators that JSBSim provides in the flight control components. If a trigger property is specified via the <trigger> element in the PID control specification, the operation of the integrator is determined by the value of the property. If the value of the property is zero (actually, if the absolute value of the property is less that 0.000001) then the integrator operates normally. If the value of the trigger property is

anything else, then the integrator contribution at that time is zero – that is, the integrator output remains at its current value. Additionally, if the value of the trigger property is less than zero, the integrator output is reset to zero.

The full roll autopilot specification then looks schematically like this:



In JSBSim-ML the specification looks like this:

```
  <switch name="fcs/wing-leveler-ap-on-off">
    <default value="-1"/>
    <test value="0">
      ap/attitude_hold == 1
    </test>
  </switch>

  <pid name="fcs/roll-ap-error-pid">
    <input>attitude/phi-rad</input>
    <kp> 50.0 </kp>
    <ki>  5.0 </ki>
    <kd> 17.0 </kd>
    <trigger> fcs/wing-leveler-ap-on-off </trigger>
  </pid>

  <switch name="fcs/roll-ap-autoswitch">
    <default value="0.0"/>
    <test value="-fcs/roll-ap-error-pid">
      ap/attitude_hold == 1
    </test>
  </switch>
```

The next question you might ask is, “but how do I determine the gains to use in the PID controller?”. There’s an art and science to that, as well. One approach is the Ziegler-Nichols method. Tuning the PID controller using the Ziegler-Nichols method will require an error to be instantly introduced to the controller. Additionally, the Integral and Derivative gains (k_i and k_d, respectively) should initially be set to zero. The best way to begin tuning the PID gains is to create a script that sets the aircraft up flying trimmed at a given altitude, allowing it to build up a roll error, and then turning on the wing leveler autopilot. The following script accomplishes that:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"
  href="http://jsbsim.sf.net/JSBSimScript.xsl"?>
<runscript xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
```

```
  xsi:noNamespaceSchemaLocation=http://jsbsim.sf.net/JSBSimScript.xsd
  name="At-altitude autopilot test setup.">

  <use aircraft="c172x" initialize="reset01"/>
  <run start="0.0" end="30" dt="0.00833333">

    <!-- Start the engine -->
    <event name="Engine start">
      <condition>sim-time-sec  ge  0.25</condition>
      <set name="fcs/throttle-cmd-norm" value="0.65"/>
      <set name="fcs/mixture-cmd-norm" value="0.87"/>
      <set name="propulsion/magneto_cmd" value="3"/>
      <set name="propulsion/starter_cmd" value="1"/>
      <set name="ap/heading_hold" value="0"/>
      <notify>
        <property>velocities/vc-kts</property>
        <property>position/h-agl-ft</property>
      </notify>
    </event>

    <!—Trim the aircraft -->
    <event name="Trim">
      <condition>sim-time-sec  ge  0.50</condition>
      <set name="simulation/do_simple_trim" value="0"/>
      <notify>
        <property>velocities/vc-kts</property>
        <property>position/h-agl-ft</property>
      </notify>
    </event>

    <!-- Turn on the roll autopilot after a roll “error”
         has developed (5 seconds should be good) -->
    <event name="Set roll autopilot">
      <condition>sim-time-sec  ge  5.0</condition>
      <set name="ap/attitude_hold" value="1"/>
      <notify>
        <property>velocities/vc-kts</property>
        <property>position/h-agl-ft</property>
      </notify>
    </event>

  </run>
</runscript>
```

Running JSBSim with this script, then plotting the roll angle versus time will show how the PID controller is working to reduce the roll angle error. The Ziegler-Nichols method can be employed as follows:

1) Make repeated runs, increasing the proportional gain K_p until the control action is seen to cause a sustained oscillatory control output. That gain is referred to as the ultimate gain, K_u, or the *critical* gain. Note also the peak-to-peak period of the oscillation. That is referred to as the *critical* oscillation period, T_c.
2) Knowing K_c and T_c, the PID gains can be set depending on whether a P, PI, or PID controller is needed. The gains are set according to the following table:

Control Type	Kp	Ki	Kd
P	0.5	0	0
PI	0.45	1.2K	0
PID	0.6	2	Kp

Let's walk through this. A series of ten runs will be made with the integrator and derivative gains set to zero. The proportional gain will be set to 5 initially and increased by five for each subsequent run. This process is carried out with a shell script that runs JSBSim repeatedly, with gain values specified on the command line. First, the PID controller is modified to have its gains specified as properties. This allows the gains to be set somewhere else (such as from a script):

```
<autopilot name="C-172X Autopilot">

<!-- INTERFACE PROPERTIES -->

  …

<!-- INITIAL GAIN VALUES -->

  <property value="50.0"> ap/roll-pid-kp </property>
  <property value="5.0"> ap/roll-pid-ki </property>
  <property value="17.0"> ap/roll-pid-kd </property>
  …
<channel name="Roll wing leveler">

  …

  <pid name="fcs/roll-ap-error-pid">
    <input>attitude/phi-rad</input>
    <kp> ap/roll-pid-kp </kp>
    <ki> ap/roll-pid-ki </ki>
    <kd> ap/roll-pid-kd </kd>
    <trigger> fcs/wing-leveler-ap-on-off </trigger>
  </pid>
```

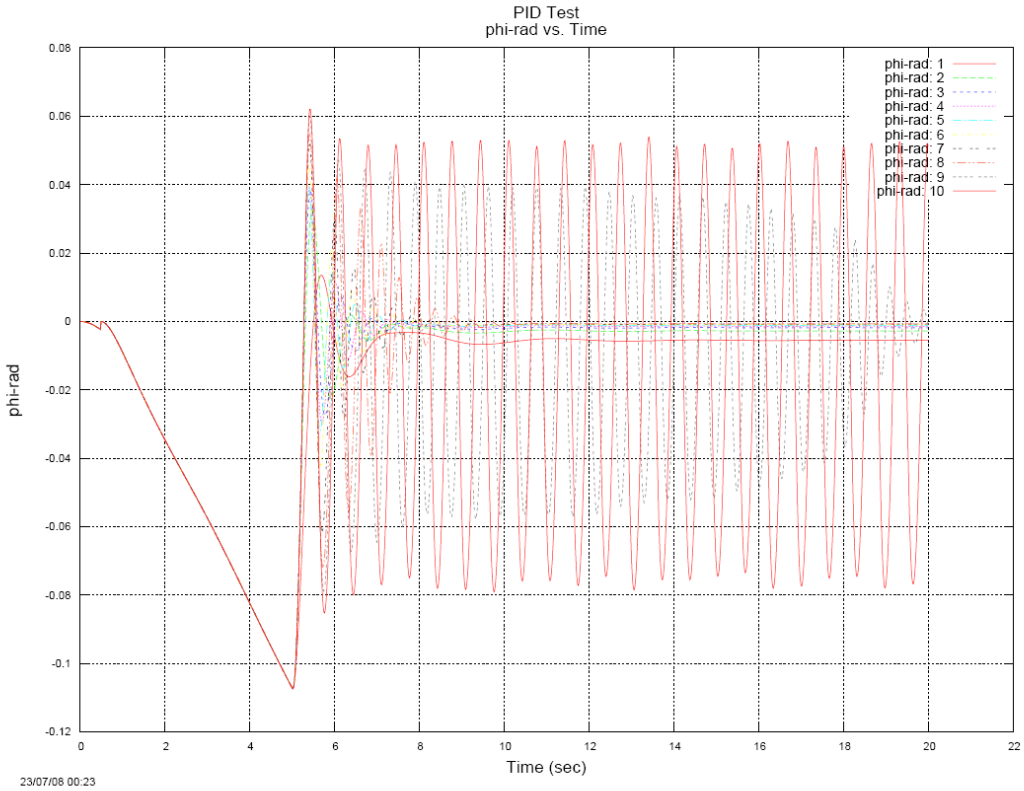
The shell script that makes the multiple runs is,

```
src/jsbsim --script=scripts/c1722 \
           --property=ap/roll-pid-kp=5 \
           --property=ap/roll-pid-ki=0 \
           --property=ap/roll-pid-kd=0 \
           --logdirectivefile=output.xml \
           --outputlogfile=pid_test_0.csv
…
src/jsbsim --script=scripts/c1722 \
           --property=ap/roll-pid-kp=50 \
           --property=ap/roll-pid-ki=0 \
           --property=ap/roll-pid-kd=0 \
           --logdirectivefile=output.xml \
           --outputlogfile=pid_test_9.csv
```

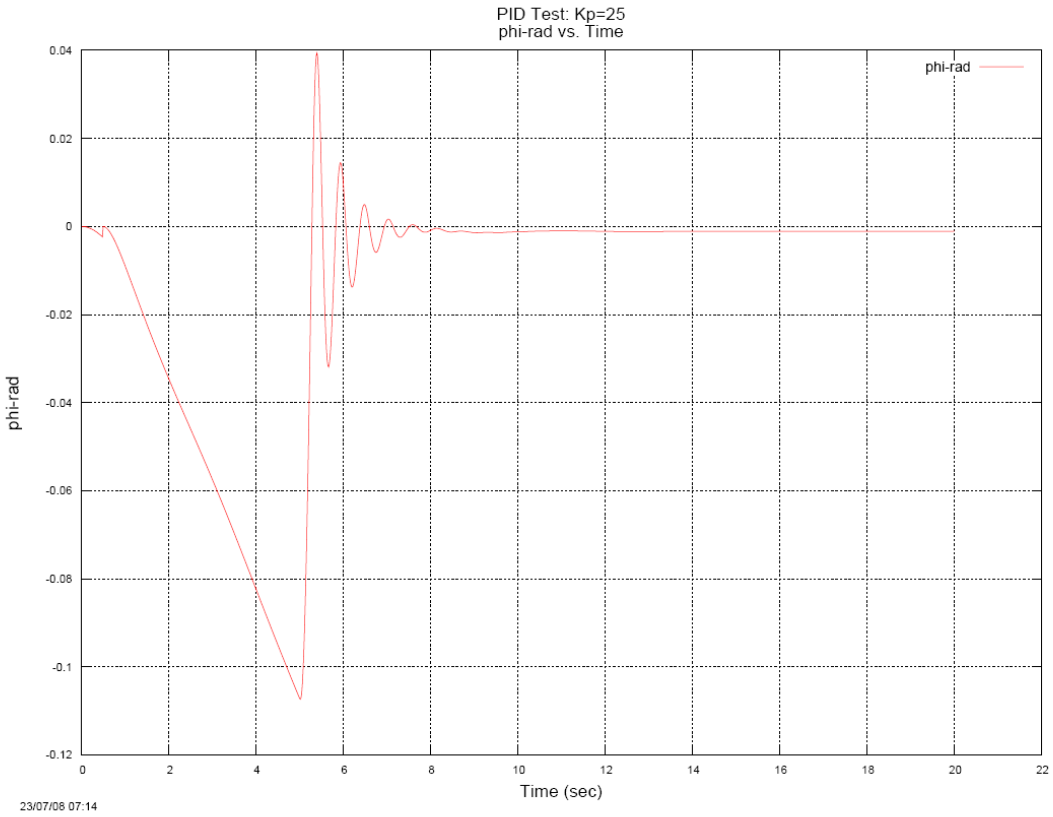
This shell script also shows that JSBSim is being told to take the output specification from the

file output.xml. That file contains a property element, directing the program to output only roll angle in radians (attitude/phi-rad). The logged output data itself is sent to the named file, pid_test_#.csv, where "#" represents a number from 0 to 9.

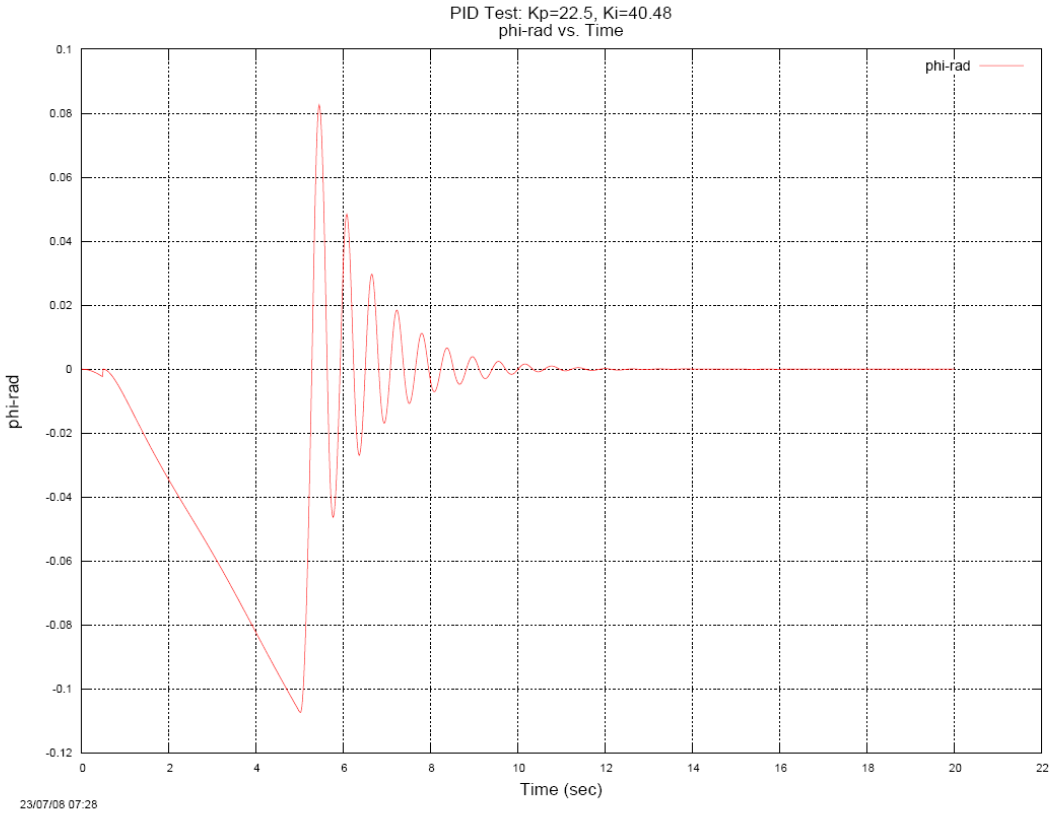
When all the data is plotted together, the response to the different value of K_p is seen. The plot begins the same for all cases: the aircraft begins flying at altitude, and begins a slow roll to the left (left wingtip down). The controller is turned on at five seconds.



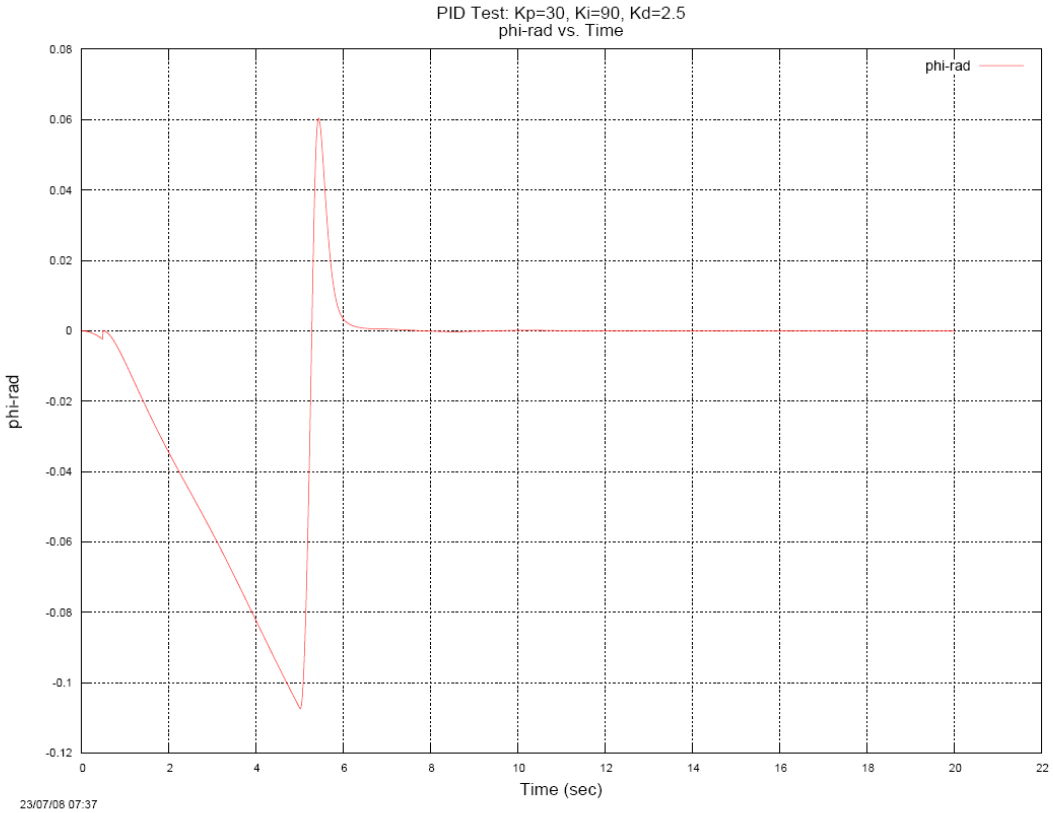
From the plot above, one can determine that the system response began to oscillate when the gain K_p was set to 50 – this is K_c – the critical gain. The peak-to-peak period of the oscillation is 0.667 seconds – this is T_c. Given the gain chart shown in step 2 of the Ziegler-Nichols method description, if a proportional only control is used, the gain K_p should be set to 0.5 K_c, or, 25. If we make the same run using that arrangement, the roll angle behavior looks like that shown in the plot:



The plot shows the roll angle being corrected, but with significant overshoot and a steady state error. So, let's try to use PI control and eliminate the bias. Again referring to the Ziegler-Nichols chart from step 2, we get a value for K_p of 22.5 and so the value for K_i is calculated from that to be 1.2 K_p/T_c = 40.48. When we run the simulation with this setup, we get this roll angle behavior:



The roll angle action is more oscillatory, but the steady state error is about zero. Let's try adding derivative control, to make a full PID controller, and see if that helps. Looking at the third line in the Ziegler-Nichols chart, we calculate the gains as follows: $K_p = 0.6\ K_c = 30$, $K_i = 2\ K_p/T_c = 90$, $K_d = K_p\ T_c/8 = 2.5$. Making a run with these values results in this roll angle behavior:

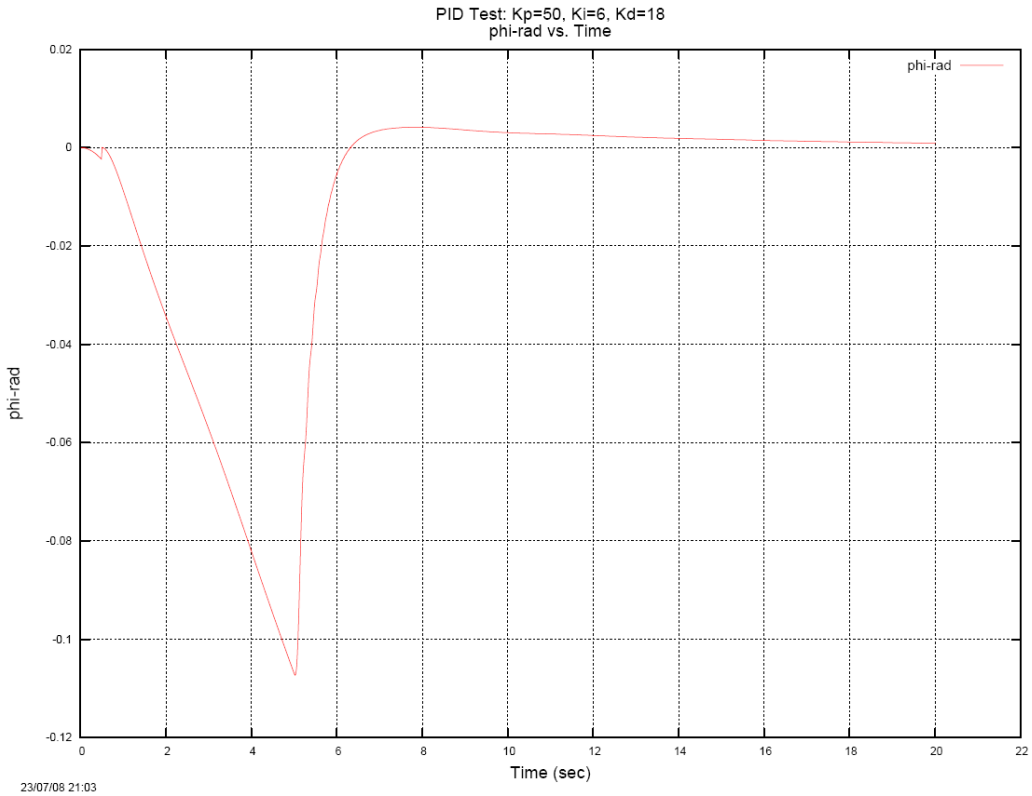


There is significant overshoot here, too, but it's less than before, and the oscillation is gone. The roll angle snaps to the wings-level position very rapidly – perhaps *too* rapidly! We might like to try and reduce the overshoot. But, which gain or gains do we adjust? Here's a table that shows the response to *increasing* any of the gains on the controller output:

Effects of increasing parameters

Parameter	Rise Time	Overshoot	Settling Time	S.S. Error
Kp	Decrease	Increase	Small Change	Decrease
Ki	Decrease	Increase	Increase	Eliminate
Kd	Small Decrease	Decrease	Decrease	None

From the chart above, it looks like we might want to increase the derivative gain, which would decrease the overshoot and the settling time. After some additional tweaking, we arrive at gains, K_p=50, K_i=6, K_d=18. The plot of the response (roll angle) to the controller is shown below:



Further testing for this PID controller might include introducing turbulence or wind gusts into the simulation, to see the response of the wing leveler to upsets.

This wing leveler controller has only a single loop. The position error is processed and an aileron command is output, resulting in a roll rate. However, in this situation, the roll rate is not limited – the roll position error is eliminated as fast as possible. A better solution might involve limiting the roll rate. The roll position error could be processed to result in a controlled and limited roll rate, and that roll rate could subsequently be compared to the actual roll rate to calculate an aileron command that would drive the roll rate error to zero. That is a good exercise for the reader!

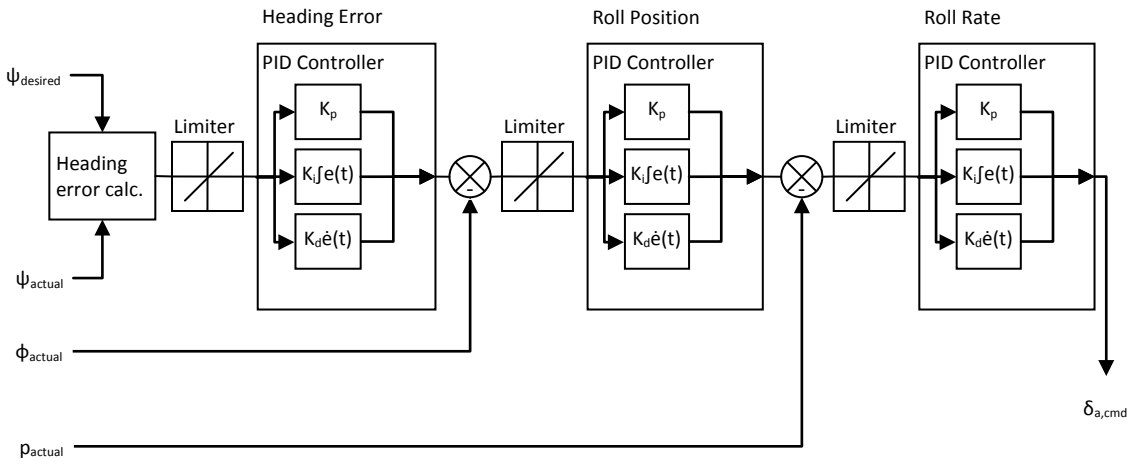
4.2 A Heading Hold Autopilot

For the heading hold autopilot, we'd like to specify some requirements. The autopilot must cause the aircraft to,

- Acquire the specified compass heading if it has not done so, or if the desired heading changes
- Maintain a specified heading
- Maintain altitude (accomplished with an altitude hold autopilot)
- When acquiring the heading, the aircraft must make a coordinated turn
- The aircraft must not bank in excess of a specified angle

The input to the autopilot will be the heading error, that is, the difference between the desired heading and the current heading. Differently than the previous wing leveler example, however,

the error will not be fed into a PID controller and directly result in a control surface command, but will instead be turned into a roll *position* command. The roll position command will be compared with the current bank angle and the difference (the error) will then be fed into another PID controller, and the output from that controller will be a roll rate command – which will be accomplished using the ailerons. In words, if there is a heading error, let's roll towards the desired heading, but let's keep our roll rate below a specified limit and keep the bank at an angle no greater than 30 degrees. Schematically, our heading hold autopilot will look like this:



5. Modeling a Waypoint Navigation System

The flight control model in JSBSim is completely configurable and includes many components that can be strung together to model specific control laws. Various automatic flight capabilities have already been modeled for some aircraft to help with testing the aircraft outside of FlightGear or other simulator. The C-172x model in the JSBSim distribution has an attitude hold, altitude hold, and heading hold capability built up.

A specific capability is needed for this case study: a waypoint navigation system (WNS) that would allow a user to enter latitude and longitude coordinates (in radians) that the aircraft would then fly to. An algorithm is developed based on the Haversine forumulas ,

Haversine Formulas

$$R = \text{Earth radius} \quad (5.1)$$

$$\Delta lat = lat_2 - lat_1 \quad (5.2)$$

$$\Delta long = long_2 - long_1 \quad (5.3)$$

$$a = \sin^2(\Delta lat/2) + \cos(lat_1)\cdot\cos(lat_2)\cdot\sin^2(\Delta long/2) \quad (5.4)$$

$$c = 2\cdot\text{atan2}(\sqrt{a}, \sqrt{(1-a)}) \quad (5.5)$$

$$distance = R\cdot c \quad (5.6)$$

$$\Phi = \text{atan2}(\sin(\Delta long)\cdot\cos(lat_2), \cos(lat_1)\cdot\sin(lat_2) - \sin(lat_1)\cdot\cos(lat_2)\cdot\cos(\Delta long)) \quad (5.7)$$

There does exist a file in the C310 directory in the default JSBSim CVS release that includes a modified version of the autopilot functions developed for testing the C172x aircraft model. The C310ap.xml file contains a heading hold, wing leveler, and altitude-hold functionality. Since the WNS calculates a heading that should be flown to, it can supply the existing heading hold autopilot with the correct heading – that is, it can feed a heading hold sub-mode. This is our assumed starting point.

The first task we will take on will be to add a set of calculations in the C310ap.xml file that will calculate the heading to a specified waypoint. First, some preliminary calculations will need to be made: delta latitude, and delta longitude. These are defined in JSBSim-ML as seen in Listing 5.1, below:

Listing 5.1

```
<fcs_function name="fcs/delta-lat-rad">
  <!-- Delta latitude in radians -->
  <function>
    <difference>
      <property>ap/wp_latitude_rad</property>
      <property>position/lat-gc-rad</property>
    </difference>
  </function>
</fcs_function>
```

```
<fcs_function name="fcs/delta-lon-rad">
  <!-- Delta longitude in radians -->
  <function>
    <difference>
      <property>ap/wp_longitude_rad</property>
      <property>position/long-gc-rad</property>
    </difference>
  </function>
</fcs_function>
```

The JSBSim *properties* for geocentric latitude and longitude are:

- position/lat-gc-rad
- position/long-gc-rad

The properties for desired latitude and longitude (the waypoint) are:

- ap/wp_latitude_rad
- ap/wp_longitude_rad

The function definitions above result in the creation of new properties at runtime, representing the difference between the desired position and the actual position. The automatically generated properties that represent the output of the calculations above are:

- fcs/delta-lon-rad
- fcs/delta-lat-rad

The above two property values represent equations 5.2 and 5.3. The actual heading from the current position to the desired location (specified by the waypoint) is given by the solution to equation 5.7. In JSBSim-ML, the definition looks like that given in Listing 5.2,

Listing 5.2

```
<fcs_function name="fcs/heading-to-waypoint-rad">
  <function>
    <atan2> <!-- atan2 (deltaY, deltaX )-->
      <product>
        <sin><property>fcs/delta-lon-rad</property></sin>
        <cos><property>ap/wp_latitude_rad</property></cos>
      </product>
      <difference>
        <product>
          <cos><property>position/lat-gc-rad</property></cos>
          <sin><property>ap/wp_latitude_rad</property></sin>
        </product>
        <product>
          <sin><property>position/lat-gc-rad</property></sin>
          <cos><property>ap/wp_latitude_rad</property></cos>
          <cos><property>fcs/delta-lon-rad</property></cos>
        </product>
      </difference>
    </atan2>
  </function>
</fcs_function>
```

The next task is to calculate the distance to the waypoint from the current position. There are additional preliminary calculations that need to be made, for equations 5.4 and 5.5. Equation 5.4

is described in JSBSim-ML format in Listing 5.3,

Listing 5.3

```
<fcs_function name="fcs/wp-distance-a">
  <function>
    <sum>
      <pow>
        <sin>
          <quotient>
            <property>fcs/delta-lat-rad</property>
            <value>2.0</value>
          </quotient>
        </sin>
        <value>2</value>
      </pow>
      <product>
        <cos>
          <property>position/lat-gc-rad</property>
        </cos>
        <cos>
          <property>ap/wp_latitude_rad</property>
        </cos>
        <pow>
          <sin>
            <quotient>
              <property>fcs/delta-lon-rad</property>
              <value>2.0</value>
            </quotient>
          </sin>
          <value>2.0</value>
        </pow>
      </product>
    </sum>
  </function>
</fcs_function>
```

Equations 5.5 and 5.6 are combined in the next fcs_function definition, described in Listing 5.4. The distance is calculated in feet, because the radius is given in feet (as noted).

Listing 5.4

```
<fcs_function name="fcs/wp-distance">
  <function>
    <product>
      <value>2.0</value>
      <atan2>
        <pow>
          <property>fcs/wp-distance-a</property>
          <value>0.5</value>
        </pow>
        <pow>
          <difference>
            <value>1.0</value>
            <property>fcs/wp-distance-a</property>
          </difference>
          <value>0.5</value>
```

```
        </pow>
      </atan2>
      <!-- 21144000 is the Earth's radius in feet -->
      <value>21144000</value>
    </product>
  </function>
</fcs_function>
```

Finally, there are three remaining short component definitions that need to be added to complete the WNS (see Listing 5.5). The summer component adds 2*pi to the heading command computed by the function component presented in Listing 2, for possible use in the next step. The switch component that follows the function component includes a conditional test of the heading-to-waypoint-rad command. If the heading-to-waypoint-rad value is already positive, then the switch takes as its output value the value specified by the heading-to-waypoint-rad property. Otherwise, the switch takes on the default value specified in the switch, heading-to-waypoint-positive, calculated previously in the summer component. Thus, the output from the switch is a positive heading command in radians. The last of these three components takes as input the switch output calculated just previously, and multiplies it by 57.3, thus turning it into a heading command in degrees.

Listing 5.5

```
<summer name="fcs/heading-to-waypoint-positive">
  <input> fcs/heading-to-waypoint-rad </input>
  <bias> 6.283 </bias>
</summer>

<switch name="fcs/wp-heading-corrector">
  <default value="fcs/heading-to-waypoint-positive"/>
  <test value="fcs/heading-to-waypoint-rad">
    fcs/heading-to-waypoint-rad gt 0.0
  </test>
</switch>

<pure_gain name="fcs/wp-heading-deg">
  <input> fcs/wp-heading-corrector </input>
  <gain> 57.3 </gain>
</pure_gain>
```

The heading command can be fed into the previously developed heading autopilot. So, the heading autopilot can now actually take either a direct heading command, or the command calculated from the WNS.

At this point we'd like to consider testing. The test involves the creation of a script to set waypoints, and a way to display the results to verify that the performance is as expected.

The test script is written in XML format. The basic format of a JSBSim script is:

```
<use aircraft="name" initialize="file"/>
<run start="0" end="time" dt="0.0833">

  <event>
    … condition[s] …
    … action[s] …
```

```
  </event>

  … more events …

</run>
```

In this test, we need to do several things:

- Start the engines.
- Set the autopilot mode to “heading hold”.
- Set the initial waypoint latitude and longitude.
- Set the initial target altitude.
- Activate the altitude autopilot on cue.
- Raise the landing gear on cue.
- Head to the first waypoint on cue.
- Head to subsequent waypoints on cue.
- Terminate on cue.

The script directs the actions as expected, but there is one feature of scripting that needs to be considered carefully: order of execution. Inputs to the script are calculated once per frame, so each test will see the same input value. For instance, prior to evaluating the tests for each event, inputs will have been calculated (such as current latitude, or the distance to a waypoint). Each property that is set in an event will take effect immediately (such as for active_waypoint). Since events are stored in program memory sequentially as they are found in the script file—and since they are likewise executed sequentially—it is possible for carelessly crafted sequential events to have their conditions evaluate to true in a single frame, in effect experiencing a cascading sequential execution. In this script, that could happen when a waypoint is reached and an event is triggered due to proximity to the target waypoint. The action taken in this test script when a waypoint is reached is to set the active waypoint to the index of the next waypoint. But, when the next waypoint test is evaluated, the distance value previously calculated may be less than the threshold distance specified for the next waypoint. With the active waypoint set to the next waypoint, the test will immediately be triggered. Subsequent events may also be triggered immediately. The solution is to list the affected events in reverse order in the script file. What happens in this case is that events which evaluate to true are followed by events that have already been triggered and executed.

The script specifies that the C310.xml file should be used. We also need to provide an initialization file that describes where to start the flight, and the initial orientation, etc. Since I am familiar with the area, Ellington Field near Houston, Texas was selected as the starting point for the scripted test flight.

Using the Google Earth application (which can be downloaded freely at http://earth.google.com) I inspected the region and picked out a few locations through which I wanted the aircraft to fly. The initial point, however, is the important one for the initialization file, representing Ellington field. The additional points are on the approach to one of the runways at Houston Hobby Airport, past that same runway, at the southwest end of Galveston Island, and at the northeast end of Galveston Island.

The script that performs the test is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"
href="http://jsbsim.sf.net/JSBSimScript.xsl"?>
<runscript xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:noNamespaceSchemaLocation=http://jsbsim.sf.net/JSBSimScript.xsd
  name="C310-01A takeoff run">

  <description>For testing autopilot capability</description>

  <use aircraft="c310" initialize="Ellington"/>
  <run start="0.0" end="3600" dt="0.00833333">

    <event name="Start engine">
      <description>
        Start engine and set initial heading and waypoints,
        turn on heading-hold mode.
      </description>
      <condition>sim-time-sec  ge  0.25</condition>
      <set name="fcs/mixture-cmd-norm[0]" value="0.87"/>
      <set name="fcs/mixture-cmd-norm[1]" value="0.87"/>
      <set name="fcs/advance-cmd-norm[0]" value="1.0"/>
      <set name="fcs/advance-cmd-norm[1]" value="1.0"/>
      <set name="propulsion/magneto_cmd" value="3"/>
      <set name="fcs/throttle-cmd-norm[0]" value="1.0"/>
      <set name="fcs/throttle-cmd-norm[1]" value="1.0"/>
      <set name="propulsion/starter_cmd" value="1"/>
      <set name="ap/altitude_setpoint" action="FG_EXP" value="1000" tc="10"/>
      <set name="ap/attitude_hold" value="0"/>
      <set name="ap/wp_latitude_rad" value="0.517238"/>
      <set name="ap/wp_longitude_rad" value="-1.662727"/>
      <set name="ap/heading_setpoint" value="355"/>
      <set name="ap/heading-setpoint-select" value="0"/>
      <set name="ap/heading_hold" value="1"/>
      <set name="ap/active-waypoint" value="0"/>
      <notify/>
    </event>

    <event name="Set altitude for 1,000 ft.">
      <condition>velocities/vc-fps ge 135.0</condition>
      <set name="ap/altitude_hold" value="1"/>
      <notify/>
    </event>

    <event name="Raise landing gear">
      <condition>position/h-agl-ft  ge  20</condition>
      <set name="gear/gear-cmd-norm" value="0"/>
      <notify/>
    </event>

    <event name="Head to first waypoint">
      <description>
        Set heading hold to selected waypoint (setpoint) instead of
        previously specified heading when altitude surpasses 800 feet.
      </description>
      <condition>position/h-agl-ft  ge  800</condition>
      <set name="ap/heading-setpoint-select" value="1"/>
      <set name="ap/active-waypoint" value="1"/>
```

```
      <notify>
        <property>fcs/wp-distance</property>
      </notify>
    </event>

    <event name="Terminate">
      <description>
        When the aircraft arrives back at Ellington
        Field (fifth waypoint) then terminate the simulation.
      </description>
      <condition>
        fcs/wp-distance lt 100
        ap/active-waypoint eq 5
      </condition>
      <set name="simulation/terminate" value="1"/>
      <notify/>
    </event>

    <event name="Set last waypoint">
      <description>
        When the distance to the fourth waypoint (northeast end of
        Galveston Island) is less than 100 feet, then set the last
        waypoint (back to Ellington Field).
      </description>
      <condition>
        fcs/wp-distance lt 100
        ap/active-waypoint eq 4
      </condition>
      <set name="ap/wp_latitude_rad" value="0.516512"/>
      <set name="ap/wp_longitude_rad" value="-1.660922"/>
      <set name="ap/active-waypoint" value="5"/>
      <notify>
        <property>fcs/wp-distance</property>
      </notify>
    </event>

    <event name="Set fourth waypoint">
      <description>
        When the distance to the third waypoint (southwest end of
        Galveston Island) is less than 800 feet, then set the fourth
        waypoint (northeast end of Galveston Island).
      </description>
      <condition>
        fcs/wp-distance lt 800
        ap/active-waypoint eq 3
      </condition>
      <set name="ap/wp_latitude_rad" value="0.511661"/>
      <set name="ap/wp_longitude_rad" value="-1.653510"/>
      <set name="ap/active-waypoint" value="4"/>
      <notify>
        <property>fcs/wp-distance</property>
      </notify>
    </event>

    <event name="Set third waypoint">
      <description>
        When the distance to the second waypoint (Hobby Airport) is
```

```
        less than 300 feet, then set the third waypoint.
      </description>
      <condition>
        fcs/wp-distance lt 300
        ap/active-waypoint eq 2
      </condition>
      <set name="ap/wp_latitude_rad" value="0.507481"/>
      <set name="ap/wp_longitude_rad" value="-1.660062"/>
      <set name="ap/active-waypoint" value="3"/>
      <notify>
        <property>fcs/wp-distance</property>
      </notify>
    </event>

    <event name="Set second waypoint">
      <description>
        When the distance to the first waypoint (Hobby
        Airport threshold) is less than 700 feet, then
        set the second waypoint.
      </description>
      <condition>
        fcs/wp-distance lt 700
        ap/active-waypoint eq 1
      </condition>
      <set name="ap/wp_latitude_rad" value="0.517533"/>
      <set name="ap/wp_longitude_rad" value="-1.663076"/>
      <set name="ap/active-waypoint" value="2"/>
      <notify>
        <property>fcs/wp-distance</property>
      </notify>
    </event>

  </run>
</runscript>
```

[Article in-work …]

6. Rocket with GNC and Scripting

For a long time now, rocket-propelled flight models in JSBSim have been possible (the X-15 was the first vehicle modeled for JSBSim), but in actually making a vertical, guided, rocket flight model, some new capabilities were added to support modeling of launch vehicles. This section will cover some of the techniques used in executing launch vehicle flight in JSBSim.

Some of the major differences in modeling vertical takeoff launch vehicle flight (compared to modeling aircraft) are:

- Takeoff is (obviously) vertical; ground reactions are tougher because they are stiffer.
- Guidance in first stage (often corresponding roughly to the atmospheric flight phase) is typically open loop, with second stage flight following a more complicated guidance scheme such as “Powered Explicit Guidance” (PEG).
- Control involves gimbaled effectors (thrusters, or nozzles).
- Control is normally completely automatic.
- Aerodynamic modeling is different.
- Sensor modeling can be more complicated, since sensors such as accelerometers cannot necessarily be placed at the center of mass (which may be inside a propellant tank!)

At this time, it is easy to model single stage rocket flight using JSBSim, but multi-stage flight is more cumbersome.

6.1 Addressing the ground reactions

For a vertically launched rocket, a way was needed to keep the vehicle firmly attached to one spot, as if it was mounted on a launch platform. It was not easy to make the standard ground reactions work well, particularly after the engines had been started, but before thrust had built up to the point where the vehicle lifted off the platform. So, a new flag was created (a property) which, if set, causes a ground reaction force equal and opposite to the aerodynamic, gravitational, and propulsive forces, and so the vehicle stays put. The property name for the hold-down flag is *forces/hold-down*.

6.2 Simple rocket guidance

The path a rocket takes in vertical flight is not normally manually flown, because the total angle of attack must be kept low to keep structural loads small, and to minimize aerodynamic drag. One key measurement tracked is q*alpha – that is, dynamic pressure multiplied by the angle of attack.

There are several phases of flight for an ascending launch vehicle:

- The vertical rise phase
- The kick angle or tilt phase

- The gravity turn phase

During the vertical rise period, the vehicle ascends straight up to clear the launch platform and then rolls to align to the appropriate heading. Sometimes, as it was in the case of the Apollo Saturn V, a slight yaw is introduced away from the launch tower just after liftoff. In the kick angle phase, the vehicle is commanded to an initial pitch angle, after which it begins the zero-lift (zero-alpha) gravity turn phase as it flies through the lower, most dense portion of the atmosphere at an increasingly higher velocity. Control during these phases is normally exercised by gimballing the main rocket engines, and/or through the use of auxiliary *reaction control jets* (RCS).

In JSBSim, we encounter a few new situations that must be addressed when modeling vertical rocket flight. First of all, in the vertical rise period, the question can be asked: how do we command the vehicle to roll to the appropriate heading, since the vehicle is pointed vertically? The frames of interest here are the *local frame*, which is a frame that is always underneath the aircraft, with the X axis pointing north, the Y axis pointing East, and the Z axis pointing downward. The origin of the local frame is at sea level. The vehicle *body frame* has the X axis along the vehicle centerline pointing forward out the nose, the Y axis pointing out the "right", and the Z axis pointing "down" out the bottom of the vehicle. For a rocket, the Y and Z directions seem somewhat arbitrary. Placing the rocket initially on a launch pad at Kennedy Space Center, Kourou, or Baikonur (for example), involves setting the initial latitude, longitude, and altitude, and orienting the vehicle on the launch pad. Since the order of rotations used in JSBSim to describe the orientation of a vehicle (that is, the orientation of the body frame for the vehicle) relative to the local frame is yaw, then pitch, then roll, it can be seen that the vehicle orientation can be described as having a pitch angle of +90 degrees relative to the local frame. If the vehicle has a particular roll orientation on the pad, the vehicle can be rolled after it is initially pitched. The initial orientation can be specified in the initialization file as having a particular *psi*, *theta*, and *phi* (Euler angles).

During the vertical rise period, the vehicle is pointed nearly straight upwards, which can cause problems in any guidance we might use to keep the vehicle pointed straight up, and also to roll it to a particular heading. One way to get around this is to simply devise a controller that both keeps the pitch and yaw rates at zero (which keeps the vehicle pointed up), and commands a roll rate to achieve the desired heading once the pitchover begins. The roll rate needs to be carefully controlled to ramp up, then back down to zero, prior to beginning to pitch over.

Control of the vehicle is performed (as mentioned before) by gimballing engines for pitch and yaw control. Roll control can be implemented by differentially gimballing multiple engines, or for a single engine vehicle by using RCS thrusters or turbine exhaust through a gimbaled nozzle. For the vertical rise period, it is not too hard to set up a control law for the pitch and yaw axes that take the pitch (or yaw) rate as input and generates a pitch (or yaw) engine gimbal command to counter that and bring the respective rate back to zero.

6.3 "Moding" and Timing

A question can be addressed at this time regarding how to command the various portions of the flight – that is, when should the vertical rise, or the tilt, or the gravity turn phases be initiated? This is related in a way to the subject of how to organize the guidance, navigation, and control (referred to as GNC) aspects of a simulated launch vehicle in JSBSim. We could call this the

simulated “flight software” (FSW), which is really what is being represented. The question to address can be summed up: is the timing of the mode changes during the flight controlled from the ground (which might be modeled in a script), or onboard the launch vehicle? If it originates on the launch vehicle, we might assert that the timing is part of the flight software. The various modes are then, in fact, initiated from aboard the vehicle itself. It’s the safest approach; think of what would happen in the case of a momentary communications loss.

So, for first stage flight we need a vertical rise phase, followed by a tilt phase, followed by a gravity turn phase – all controlled from the launch vehicle itself. For JSBSim that means we need the sequencing function to be present in the *flight control* or *system* section of the vehicle configuration file.

The *system* section is a relatively recent addition in the latest version of JSBSim. Any number of systems can be defined, and they use the same syntax as the flight control section. The system section is useful when a guidance, or navigation, or electrical system is desired to be modeled in JSBSim using the suite of system components.

6.4 Where to Start?

As with all JSBSim aircraft models, vehicle characteristics will need to be known:

- Mass properties
- Geometry
- Control
- Propulsion
- Aerodynamics
- Ground contacts

Geometry is almost always quite easy. Mass properties are little bit harder, but one can make fairly good guesses at this. [Note: for some historic rockets such as the Saturn V or Saturn IB, there *may* be reference material available on the web and/or in technical papers.] Fuel burn-off might be a complicating factor because the CG may shift, there may be fuel slosh (for liquid propellant rockets), and moments of inertia will change. *Control* (Guidance, Navigation, and Control, actually) is the most involved part of modeling a rocket. Propulsion for rockets can be modeled rather simply in JSBSim. Aerodynamics is second to control as far as level of difficulty. Ground contacts are straightforward.

6.5 Aerodynamics

Let’s look at aerodynamics first. We can determine the aerodynamic characteristics of a simple axisymmetric body using DATCOM+ (see www.holycows.net), or – if you have access to it – Missile DATCOM. There are other software titles out there as well. We will use the lift and drag curves for the V-2 missile (as found in Sutton’s *Rocket Propulsion Elements* book) to model our example rocket. We will also assume that the aerodynamics will be completely modeled by proper placement of the aerodynamic center of pressure, coupled with the lift and drag curves, and that the side force is the same as the lift force.

6.6 Mass Properties

Since we know the mass characteristics of the Little Joe, we will scale the moments and

products of inertia of that vehicle once we specify the weight of our example rocket.

[Article in-work …]

Appendices

Native properties

accelerations/Nz
accelerations/a-pilot-x-ft_sec2
accelerations/a-pilot-y-ft_sec2
accelerations/a-pilot-z-ft_sec2
accelerations/n-pilot-x-norm
accelerations/n-pilot-y-norm
accelerations/n-pilot-z-norm
accelerations/pdot-rad_sec2
accelerations/qdot-rad_sec2
accelerations/rdot-rad_sec2
accelerations/udot-ft_sec2
accelerations/vdot-ft_sec2
accelerations/wdot-ft_sec2
aero/alpha-deg
aero/alpha-max-rad
aero/alpha-min-rad
aero/alpha-rad
aero/alpha-wing-rad
aero/alphadot-deg_sec
aero/alphadot-rad_sec
aero/beta-deg
aero/beta-rad
aero/betadot-deg_sec
aero/betadot-rad_sec
aero/bi2vel
aero/ci2vel
aero/cl-squared
aero/h_b-cg-ft
aero/h_b-mac-ft
aero/mag-beta-deg
aero/mag-beta-rad
aero/qbar-area
aero/qbar-psf
aero/qbarUV-psf
aero/qbarUW-psf
aero/stall-hyst-norm
atmosphere/P-psf
atmosphere/P-sl-psf
atmosphere/T-R
atmosphere/T-sl-R
atmosphere/T-sl-dev-F
atmosphere/a-fps
atmosphere/a-ratio
atmosphere/a-sl-fps
atmosphere/crosswind-fps
atmosphere/delta
atmosphere/delta-T
atmosphere/density-altitude
atmosphere/gust-down-fps
atmosphere/gust-east-fps
atmosphere/gust-north-fps
atmosphere/headwind-fps
atmosphere/p-turb-rad_sec
atmosphere/psiw-rad
atmosphere/q-turb-rad_sec
atmosphere/r-turb-rad_sec
atmosphere/rho-sl-slugs_ft3
atmosphere/rho-slugs_ft3
atmosphere/sigma
atmosphere/theta
atmosphere/turb-gain
atmosphere/turb-rate
atmosphere/turb-rhythmicity
atmosphere/wind-from-cw
attitude/heading-true-rad
attitude/phi-rad
attitude/pitch-rad
attitude/psi-rad
attitude/roll-rad
attitude/theta-rad
fcs/aileron-cmd-norm
fcs/center-brake-cmd-norm
fcs/elevator-cmd-norm
fcs/elevator-pos-deg

fcs/elevator-pos-norm
fcs/elevator-pos-rad
fcs/flap-cmd-norm
fcs/flap-pos-deg
fcs/flap-pos-norm
fcs/flap-pos-rad
fcs/left-aileron-pos-deg
fcs/left-aileron-pos-norm
fcs/left-aileron-pos-rad
fcs/left-brake-cmd-norm
fcs/mag-elevator-pos-rad
fcs/mag-left-aileron-pos-rad
fcs/mag-right-aileron-pos-rad
fcs/mag-rudder-pos-rad
fcs/mag-speedbrake-pos-rad
fcs/mag-spoiler-pos-rad
fcs/pitch-trim-cmd-norm
fcs/right-aileron-pos-deg
fcs/right-aileron-pos-norm
fcs/right-aileron-pos-rad
fcs/right-brake-cmd-norm
fcs/roll-trim-cmd-norm
fcs/rudder-cmd-norm
fcs/rudder-pos-deg
fcs/rudder-pos-norm
fcs/rudder-pos-rad
fcs/speedbrake-cmd-norm
fcs/speedbrake-pos-deg
fcs/speedbrake-pos-norm
fcs/speedbrake-pos-rad
fcs/spoiler-cmd-norm
fcs/spoiler-pos-deg
fcs/spoiler-pos-norm
fcs/spoiler-pos-rad
fcs/steer-cmd-norm
fcs/yaw-trim-cmd-norm
flight-path/gamma-rad
flight-path/psi-gt-rad
forces/fbx-aero-lbs
forces/fbx-gear-lbs
forces/fbx-prop-lbs
forces/fbx-total-lbs
forces/fby-aero-lbs
forces/fby-gear-lbs
forces/fby-prop-lbs
forces/fby-total-lbs
forces/fbz-aero-lbs
forces/fbz-gear-lbs
forces/fbz-prop-lbs
forces/fbz-total-lbs
forces/fwx-aero-lbs
forces/fwy-aero-lbs
forces/fwz-aero-lbs
forces/hold-down
forces/lod-norm
gear/gear-cmd-norm
gear/gear-pos-norm
gear/num-units
ic/alpha-deg
ic/alpha-rad
ic/beta-deg
ic/beta-rad
ic/gamma-deg
ic/gamma-rad
ic/h-agl-ft
ic/h-sl-ft
ic/lat-gc-deg
ic/lat-gc-rad
ic/long-gc-deg
ic/long-gc-rad
ic/mach
ic/p-rad_sec
ic/phi-deg
ic/phi-rad
ic/psi-true-deg
ic/psi-true-rad
ic/q-rad_sec
ic/r-rad_sec
ic/roc-fpm
ic/roc-fps
ic/sea-level-radius-ft
ic/terrain-altitude-ft
ic/theta-deg
ic/theta-rad
ic/u-fps
ic/v-fps

ic/vc-kts
ic/vd-fps
ic/ve-fps
ic/ve-kts
ic/vg-fps
ic/vg-kts
ic/vn-fps
ic/vt-fps
ic/vt-kts
ic/vw-bx-fps
ic/vw-by-fps
ic/vw-bz-fps
ic/vw-dir-deg
ic/vw-down-fps
ic/vw-east-fps
ic/vw-mag-fps
ic/vw-north-fps
ic/w-fps
inertia/cg-x-in
inertia/cg-y-in
inertia/cg-z-in
inertia/empty-weight-lbs
inertia/mass-slugs
inertia/weight-lbs
metrics/Sh-sqft
metrics/Sv-sqft
metrics/Sw-sqft
metrics/aero-rp-x-in
metrics/aero-rp-y-in
metrics/aero-rp-z-in
metrics/bw-ft
metrics/cbarw-ft
metrics/eyepoint-x-in
metrics/eyepoint-y-in
metrics/eyepoint-z-in
metrics/iw-deg
metrics/iw-rad
metrics/lh-ft
metrics/lh-norm
metrics/lv-ft
metrics/lv-norm
metrics/runway-radius
metrics/vbarh-norm
metrics/vbarv-norm
metrics/visualrefpoint-x-in
metrics/visualrefpoint-y-in
metrics/visualrefpoint-z-in
moments/l-aero-lbsft
moments/l-gear-lbsft
moments/l-prop-lbsft
moments/l-total-lbsft
moments/m-aero-lbsft
moments/m-gear-lbsft
moments/m-prop-lbsft
moments/m-total-lbsft
moments/n-aero-lbsft
moments/n-gear-lbsft
moments/n-prop-lbsft
moments/n-total-lbsft
output-norm
position/distance-from-start-lat-mt
position/distance-from-start-lon-mt
position/distance-from-start-mag-mt
position/epa-rad
position/geod-alt-ft
position/h-agl-ft
position/h-sl-ft
position/h-sl-meters
position/lat-gc-deg
position/lat-gc-rad
position/lat-geod-deg
position/lat-geod-rad
position/long-gc-deg
position/long-gc-rad
position/radius-to-vehicle-ft
position/terrain-elevation-asl-ft
propulsion/active_engine
propulsion/cutoff_cmd
propulsion/fuel_dump
propulsion/magneto_cmd
propulsion/pt-lbs_sqft
propulsion/refuel
propulsion/set-running
propulsion/starter_cmd
propulsion/starter_cmd
propulsion/tat-c

propulsion/tat-r
propulsion/total-fuel-lbs
sim-time-sec
simulation/cycle_duration
simulation/do_simple_trim
simulation/do_trim_analysis
simulation/frame_start_time
simulation/integrator/position/rotational
simulation/integrator/position/translational
simulation/integrator/rate/rotational
simulation/integrator/rate/translational
simulation/terminate
simulation/write-state-file
systems/stall-warn-norm
velocities/eci-velocity-mag-fps
velocities/h-dot-fps
velocities/mach
velocities/machU
velocities/p-aero-rad_sec
velocities/p-rad_sec
velocities/phidot-rad_sec
velocities/psidot-rad_sec
velocities/q-aero-rad_sec
velocities/q-rad_sec
velocities/r-aero-rad_sec
velocities/r-rad_sec
velocities/thetadot-rad_sec
velocities/u-aero-fps
velocities/u-fps
velocities/v-aero-fps
velocities/v-down-fps
velocities/v-east-fps
velocities/v-fps
velocities/v-north-fps
velocities/vc-fps
velocities/vc-kts
velocities/ve-fps
velocities/ve-kts
velocities/vg-fps
velocities/vt-fps
velocities/w-aero-fps
velocities/w-fps

Glossary

FCS	Flight Control System
JSBSim-ML	JSBSim Markup Language
NED	North East Down
VRP	Vehicle Reference Point
XML	eXtensible Markup Language

INDEX

G

H

I

K

L

M

N

O

P

R

S

T

U

W