The quarterly newsletter for JSBSim, an open source flight dynamics model in C++

Back of the Envelope

Volume 2, Issue 1

15 April 2005

Inside this issue:

Sneak Peek: JSBSim Commander	1
News	3
In Depth: C# and JSBSim.Net	4
Scripting in JSBSim	7
Decoupling, Generalizing, and Isolating Commands and Effectors in the JSBSim FDM	9
Simulate This! LLRV	10

Sneak Peek: JSBSim Commander

One drawback for beginners in creating a flight model for an aircraft is that the information that must be gathered can be hard to come by, and the format of the configuration file can be confusing. To address this, an application is being developed to greatly ease that process.

"JSBSim Commander" (the name for the alpha version) is currently being developed for the *Windows* platform. That news may be disconcerting for some, but there are reasons for it. The application is complex and large. It involves a lot of screens. Given the main developer's familiarity with Borland C++Builder, that tool is being used to create the GUI screens very rapidly. An attempt was made to use a cross platform tool (Glade) to build the initial GUI, but that proved to be a nightmare. Additionally, the time required to learn a new tool/paradigm just isn't there.

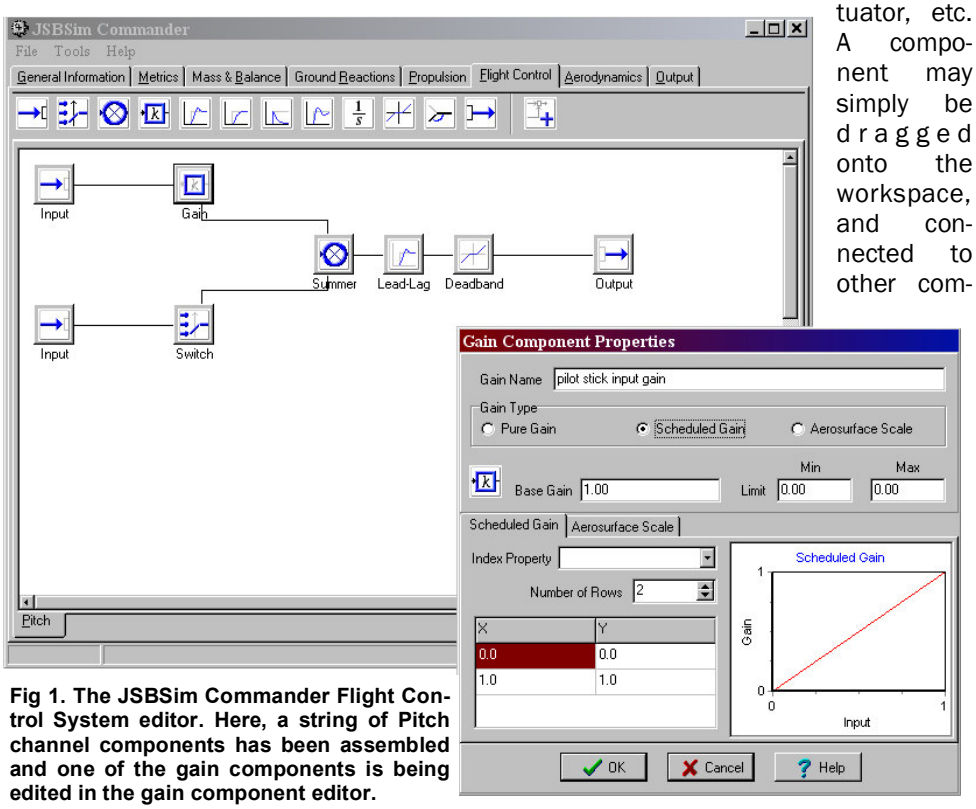
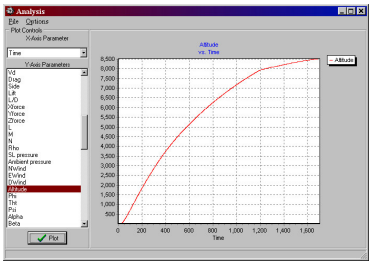


Fig 1. The JSBSim Commander Flight Control System editor. Here, a string of Pitch channel components has been assembled and one of the gain components is being edited in the gain component editor.

Here is a list of features that are planned for JSBSim Commander:

1) Reads, writes, and allows editing of JSBSim configuration files (configuration version 2.0, not yet released)
2) The flight control system editor will allow block diagrams to be built for any number of "channels". A channel represents a string of controls such as stick to elevator, pedals to rudder, landing gear lever to landing gear actuator, etc. A component may simply be dragged onto the workspace, and connected to other components. Each component can be edited depending on its type.
3) The propulsion system editor will allow the creation of new engines and thrusters, and/or the use of existing engines and thrusters. A list of existing engines and thrusters is presented to users, from which the user can "subscribe" to an engine, and then assign a thruster to it.

(Continued on page 2)



The plotting tool makes analysis of the JSBSim log files (in comma separated value format) very quick.

(Continued from page 1)

4) The aerodynamics editor is still being designed, but it will allow the creation of force and moment components that are functions of tables, equations, etc.
5) Aeromatic has been converted from a web application to a Wizard for use in creating an initial cut at an aircraft definition.
6) A plotting tool has been developed (this already works) that can be used to rapidly plot and analyze data output by JSBSim. The currently supported data format is comma separated value (csv). More formats may be supported in the future.
7) A planned capability is to allow importing high fidelity aerodynamic data from the files generated by Bill Galbraith's DATCOM+. This data would replace the default data generated by Aeromatic. [See the article in the October 2004 issue of this newsletter, or on the web site at: www.holycows.net/datcom/ for more information about DATCOM+.]

No release date has been targeted, yet.

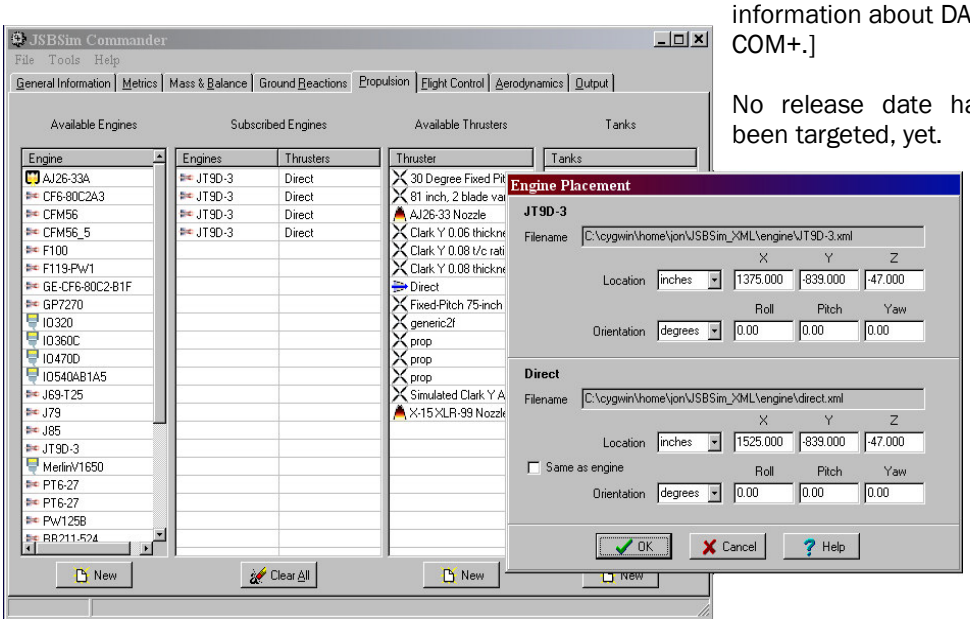
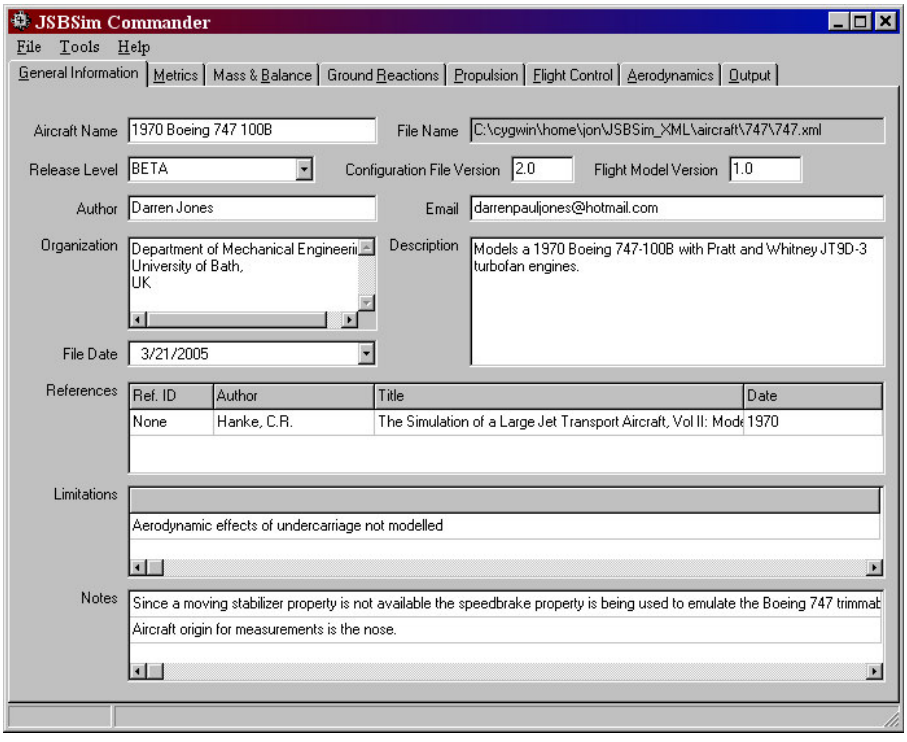


Fig 2. Above is the Engine editor. Engine/Thruster combinations can be defined here, and the location of each specific component can be edited, as well. Below, the main aircraft information editor is shown.



News Items

Rain Mountain Systems Incorporated (RMSI) is planning to use JSBSim as an analysis tool during a Phase I SBIR (Small Business Innovation Research) with the Air Force Test Pilot School. This SBIR is for the development of a "personal issue flight test recorder and display" using PDA with plug-in instrumentation cards. One of the biggest challenges in this SBIR is to develop methods for determining the aircraft state from low resolution, low-sample-rate instrumentation that might be carried aboard the test aircraft, and RMSI plans to use JSBSim to test those methods.

RMSI also hopes to use JSBSim as an interactive simulation (or as the basis of such a simulation) to provide an engineering development and customer demonstration tool for methods, approaches, and display concepts required to implement this flight test personal data assistant (FT-PDA).

Contact:
Lee Duke, Chief Engineer
Rain Mountain Systems Incorporated
duke@rainmountainsystems.com

▲

JSBSim version 0.9.7 has been released on the project web site. This is the last formal release prior to a major code reorganization, as mentioned in a previous newsletter. Version 0.9.7 includes changes to allow JSBSim aircraft to take off and land on an aircraft carrier.

www.jsbsim.org

▲

JidVolo is a flight simulator based on JSBSim, and it's mainly a work for a virtual reality course. Careful attention is given to providing realistic scenery. jidVolo currently uses Massachussets height maps since similar maps of Italy (where jidVolo is developed) are not readily available.

jidVolo runs on Linux and on Windows. You can download jidVolo here:

www.sourceforge.net/projects/jidvolo

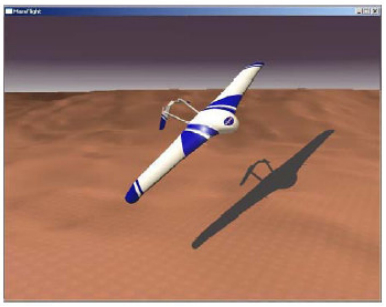
JSBSim and FlightGear are being used in early development of the *MarsFlight* software.

Red Canyon Software, Inc., a Denver-based small HUBZone business, specializing in aerospace engineering services, has been awarded a contract with NASA Glenn Research Center. The contract is for the development of NASA's "MarsFlight" Educational Software Simulator. The Phase I software is currently scheduled to be displayed at the Paris Air Show this June.

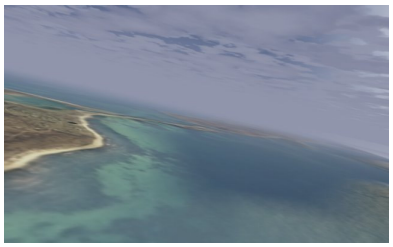
MarsFlight will be an immersive and interactive simulation of a remotely controlled plane that will fly over Mars with the goal of educating our next generation of explorers. The Mars terrain will be developed using actual data obtained from the Mars Orbiter Laser Altimeter (MOLA) which resides on the Mars Global Surveyor (MGS) satellite – currently in orbit around Mars. According to the Design and Reference document released by NASA Glenn Research Center, "Tens of thousands of people will use MarsFlight every year at many national and international exhibits that NASA participates in. In addition, it is expected that MarsFlight will be available as a free download and will be distributed to educators and schools packaged on a CDROM."

"This contract is a tremendous win for Red Canyon Software, small Colorado companies and the Colorado aerospace community. It reinforces the fact that Colorado continues to be a leader in space exploration and ranks 4th in the nation in terms of space dollars," said Barry Hamilton, CEO and Founder of Red Canyon Software. "The MarsFlight Simulator will support Red Canyon Software's mission by providing school age children a fun and exciting environment to learn about Mars, space exploration and robotic missions, encouraging our next generation of engineers and astronauts, as well as supporting space exploration. By choosing the Space Foundation as our strategic educational partner for Phase II of MarsFlight, we are optimizing the educational value of the software." Mr. Hamilton also stated, "Without the support of our current clients, Lockheed Martin, Ball Aerospace

(Continued on page 4)

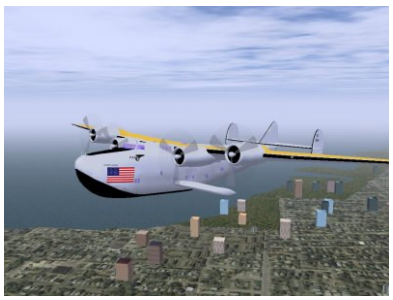


MarsFlight: NASA ARES Plane flying over Mars' Valles Marineris (The Grand Canyon of Mars).



jidVolo flight simulator, view of the ocean.

New JSBSim aircraft models in-work, modified, or completed this quarter (top-to-bottom), Boeing 314, Lockheed Constellation, Concorde (screen captures from FlightGear):



(News: Continued from page 3)
and SEAKR Engineering, we would not have had the experience and resources to win this contract".

Phase I will be completed for Beta testing no later than May 20th, 2005. Phase II will be an enhanced version of Phase I, including a dynamic topographic/ atmospheric density display and a dynamic graphic Mars Airplane systems status display. The Phase II option is scheduled for beta testing in April 2006.

www.redcanyonsoftware.com/Mars.pdf

Contact:
Jennifer L. Jones
Director of Business Development
Red Canyon Software
303.520.3906
jenniferjones@redcanyonsoftware.com

Engineers at Strategic Aeronautics, part of the Virginia SATS team, are using FlightGear and JSBSim to provide realistic simulated flight data during development of their moving map. The map provides improved situational awareness for research of automated sequencing and separation of high density IFR traffic into non towered airports. Virginia SATS is one of several teams researching advanced aviation technologies under NASA's Small Aircraft Transportation System program, which will be demonstrating findings to NASA and the public June 5-7, 2005 at the Danville, VA airport.

http://sats.larc.nasa.gov
http://www.sats2005.com (national SATS program web site)

▲

In Depth: C#, and JSBSim.NET

By Agustin Santos Mendez

"JSBSim.Net" is a port of the JSBSim Flight Dynamics Model to the .NET runtime. We have kept the framework similar in spirit to the original JSBSim while taking advantage of new features in the .NET runtime.

Why JSBSim.Net?

It is written entirely in C# and has been completely redesigned to take advantage of many .NET language features, for example *custom attributes* and other *reflection* related capabilities.

C# is a strongly-typed object-oriented language designed to give the optimum blend of simplicity, expressiveness, and performance.

C# is a strongly-typed object-oriented language designed to give the optimum blend of simplicity, expressiveness, and performance. The .NET platform is centered around a Common Language Runtime (similar to a JVM) and a set of libraries which can be exploited by a wide variety of languages which are able to work together by all compiling to an intermediate language (IL).

The last 10 years has seen a profusion of language bindings and platforms which have their own strengths and weaknesses, and their own ardent followers. The problem of maintaining so many different language bindings is non-trivial, especially with the rate of change of compiler releases. The Microsoft .NET framework offers a possible long term solution to these problems, by allowing a single product to work well with many different language bindings, and potentially across many different platforms. In that way, JSBSim.Net brings JSBSim to all .NET languages, including C#, C++ .NET (MC++), VB.NET, J#, Delphi.NET, Jscript.NET, Lua.NET, Perl.NET, Python.NET.

Though .NET has been around since 2002, it is still a relatively new techology for the development industry. By using the .Net framework as the target platform, developers can focus more on core functionality and logic, rather than dealing with the complexities of other languages.

A language is pretty much useless without libraries. C# has remarkably few core libraries, but utilizes the libraries of the .NET framework (some of which were built with C#). Briefly, the .NET libraries come with a rich set of libraries including Threading, Collection, XML, communications, regular Expression, GUI libraries (WinForms or GTK#), ... Some of these libraries are cross-platform, while others are Windows dependent.

Additionally, .NET has been a windows-

(Continued on page 5)

(Continued from page 4)

only platform until just recently with the 1.0 release of Mono, the open-source, cross-platform implementation of .NET. Without any extra work on your part, your application will run on Linux, Mac OS X, and Windows using any of the following .NET runtimes: MS.NET or Mono. We're not yet testing JSBSim.Net on Mono but it is a development goal.

When you move C++ based code to a managed environment such as .NET you can expect some performance loss. We expect than this performance loss will be minimal, but more performance tests have to be done.

The new version has significant features changes, the more significant ones are:

- Design and architecture based on JSBSim.
- 100% C# codebase built targeting the .Net Framework 1.1.
- Adherence to the best practices of .Net framework naming standards and methodologies (i.e. Use of properties instead of GetX()/SetX(). Usage of .Net framework class library wherever possible.
- Added a new logger system called Log4Net. A new Output capability has been added using this logger. Now, it is possible to specify a new output channel and determine its format using the Log4Net power (see example below).
- Add Serialization capability.
- Regression unit-testing.
- Documentation based on C# comments. Easy to export XML documentation and from there on output it via Ndoc or similar software tools.

Property System

The JSBSim property system has been modified using C# capabilities. Using C# properties (get/set), attributes, and reflection; it is easy to specify the JSBSim property node. An example is shown in Listing 1.

We can also observe the C# documentation tags (similar to Javadoc). Although, there is no "inline" functionality in C#, this properties should be treated as inline by the compiler.

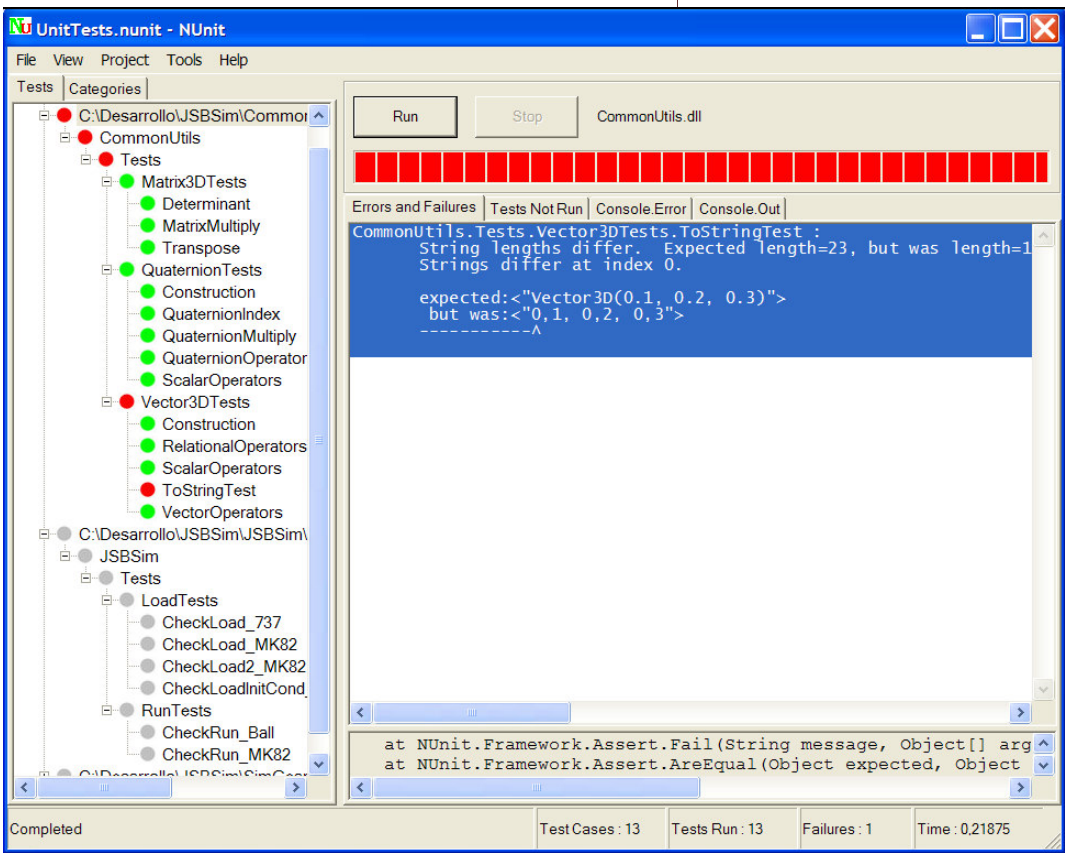


Fig. 1 Nunit in action

Unit

NUnit is a regression unit-testing framework for all .Net languages. In Listing 2 we show how we are using it.

(Continued on page 6)

```
/// <summary>
/// Total pressure above is freestream total pressure for subsonic only;
/// for supersonic it is the 1D total pressure behind a normal shock
/// </summary>
[ScriptAttribute("velocities/pt-lbs_sqft", "Total pressure.")]
public double TotalPressure { get { return pt; }}
```

Listing 1 Properties

(Continued from page 5)

In this example, we show how the "ToString test" could fail. In this case, it is due to a locale-specific format.

```
[TestFixture]
public class LoadTests
{
  private const String aircraft_737 = "737";

  [Test]
  public void CheckLoad_737()
  {
    FDMExecutive fdm = new FDMExecutive();
    fdm.LoadModel("aircraft", "engine", aircraft_737, true);

    Assert.AreEqual(aircraft_737, fdm.ModelName);
    Assert.AreEqual("MK-82", fdm.Aircraft.AircraftName);
    Assert.AreEqual(1171.0, fdm.Aircraft.WingArea);
    Assert.AreEqual(94.7, fdm.Aircraft.WingSpan);
    Assert.AreEqual(12.31, fdm.Aircraft.WingChord);
    Assert.AreEqual(348.0, fdm.Aircraft.HTailArea);
    Assert.AreEqual(48.04, fdm.Aircraft.HTailArm);
    Assert.AreEqual(297.00, fdm.Aircraft.VTailArea);
    Assert.AreEqual(new Vector3D(80.0,-30.0,70.0), fdm.Aircraft.EyepointXYZ);
  }
}
```

Listing 2 Testing

Log4Net

Use a new logger system called Log4Net. With log4net it is possible to enable logging at runtime without modifying the application binary. The log4net package is designed so that log statements can remain in shipped code without incurring a high performance cost.

At the same time, log output can be so voluminous that it quickly becomes overwhelming. One of the distinctive features of log4net is the notion of hierarchical loggers. Using these loggers it is possible to selectively control which log statements are output at arbitrary granularity.

The Log4Net configuration is based in XML, as shown in Listing 3.

With the line:

```
<appender name=...
```

we specify the output Console target of our logs. With Log4Net it is possible to redirect the output to multiple logging targets (Files, memory, remote, SMTP, ADO, UDP, EventLog, ...).

More often than not, users wish to customize not only the output destination but also the output format. This is accomplished by associating a *layout* with an appender. The layout is responsible for

(Continued on page 7)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="log4net" type="System.Configuration.IgnoreSectionHandler"/>
  </configSections>
  <log4net>
    <appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender">
      <layout type="log4net.Layout.PatternLayout">
        <param name="ConversionPattern" value="%-4r %d [%t] %-5p %c - %m%n"/>
      </layout>
    </appender>

    <!-- Set root logger level to ALL and its only appender to A1 -->
    <root>
      <level value="ALL" />
      <appender-ref ref="ConsoleAppender" />
    </root>
  </log4net>
</configuration>
```

Listing 3

(Continued from page 6)

formatting the logging request according to the user's wishes, whereas an appender takes care of sending the formatted output to its destination. We do that with the line:

<param name="ConversionPattern" value="%-4r %d [%t] %-5p %c - %m%n" />

And in Figure 2, we show Log4Net and Nunit working together

Road Map

The C# version is nearing completion, and we were able to start validating the output of our standard test programs. Current work is concentrated on polishing existing code, removing remaining bugs, and performance optimizations. Also, some tests have to be done on Mono and Portable.Net frameworks. Once complete, v1.0 will be released, and I'll begin working on some additional features:

- Integration with external build systems, such as Nant
- Scripting functionality, primarily an integration with Python.
- Flexible plugin architecture for dynamically extending core functionality at runtime.
- Integration with a charting library for .NET. We are studying two alternatives: Nplot (http://netcontrols.org/nplot/) and ZedGraph (http://zedgraph.sourceforge.net).

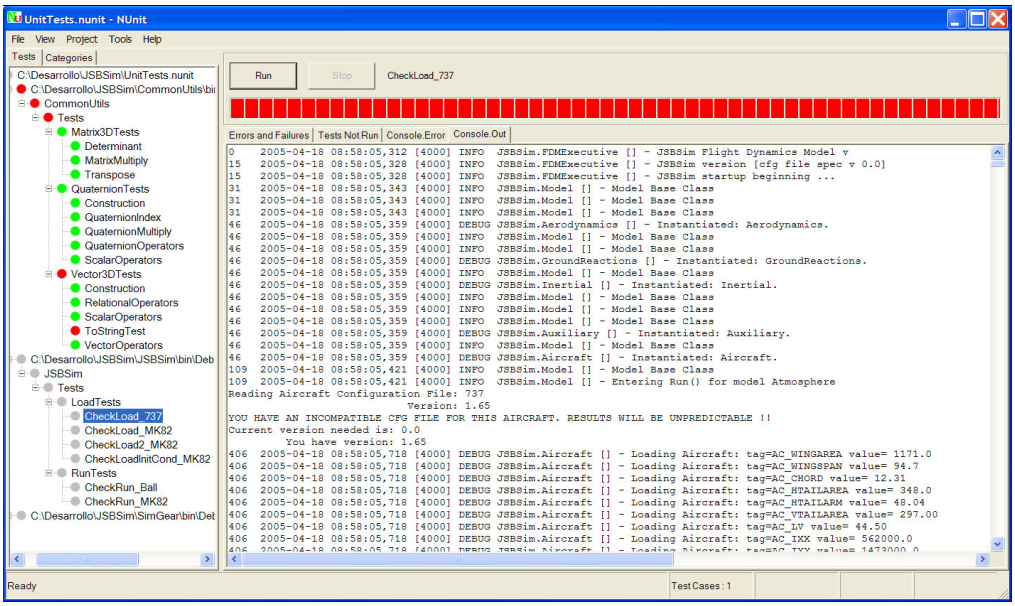


Fig. 2 Log4Net and Nunit working together.

Scripting JSBSim

The scripting feature of JSBSim is made up of language elements that allow for AND and OR conditional tests to be built up and for one or many actions to be taken when the condition[s] evaluates to true. The value of scripting is realized any time a repeatable set of conditions needs to be performed for testing or analysis, or when checking the response of the simulation code to a given test condition during debugging.

A JSBSim runscript is made up of a reference to an aircraft and a set of initial conditions, to a start and stop time, the integration time constant, and to any number of conditional tests and actions specifications. The script always starts like this (using the Cessna 172 as an example):

```
<?xml version="1.0"?>
<runscript name="C-172 takeoff run">
  <use aircraft="c172">
  <use initialize="reset00">
  ...
```

Above, we see the xml version specification, the name of the script, an optional comment, and the specification of the aircraft and initialization file to use.

(Continued on page 8)

The value of scripting is realized any time a repeatable set of conditions needs to be performed for testing or analysis, or when checking the response of the simulation code to a given test condition during debugging.

(Scripting JSBSim, Continued from page 7)

The real work is done in the *run* section:

```
<run start=0 end=100 dt=0.1">
  [directives]
</run>
```

Here, we see that the script is designed to run from 0 to 100 seconds total elapsed time with a time step of 0.1 seconds.

The directives section contains the actual conditional tests and actions to be performed. A directive specifies an action to be taken when a condition occurs. The statements that comprise the directive are wrapped in a <when></when> block. There are two required statement types within the directive "when" block: conditions and actions. A condition is specified as follows:

```
<parameter name={property}
 comparison={logical operator}
 value={value}/>
```

{logical operator} is one of the following:

le less than or equal to
lt less than
ge greater than or equal to
gt greater than
eq equal to
ne not equal to

{value} is the numeric value to make the comparison with.

An action to be taken is specified as follows:

```
<set name={property}
  type={value type}
  action={change type}
  persistent={Boolean}
  tc={time constant}/>
```

where *property* is as before (see above), and *type* is one of:

FG_DELTA an offset from the current actual value
FG_BOOL a boolean
FG_VALUE the actual value to be used

The *action* is one of:

FG_EXP An exponential change to the new value
FG_RAMP The old value ramps linearly to the new one
FG_STEP The new value immediately takes effect

The "persistent" item is set true or false, and this setting determines whether or not the trigger (set of conditional checks) would be reset once it becomes inactive (evaluates to false) and would be automatically reused. Set "persistent" to true if you would like to reuse this trigger indefinitely. The "tc" parameter is a time constant used to specify at what rate the new setting should be faded in. This is important for the ramp and exponential actions, but is meaningless for the step action. In actual practice, a "when" clause might look something like this:

```
<when>
  <parameter name="sim-time-sec"
   comparison="ge" value="0.25"/>
  <parameter name="sim-time-sec"
   comparison="le" value="0.50"/>
  <set name="fcs/aileron-cmd-norm"
  type="FG_VALUE" value="0.25"
  action="FG_STEP" persistent="false"
  tc ="0.25"/>
</when>
```

This says: "We define an event to take place when the simulation time is between 0.25 and 0.50 seconds, and when that happens, we set the aileron *command* (normalized value between –1 and +1) to 0.25 via a step, immediately changing the value – the tc item is ignored (but still needed in the current version of JSBSim, prior to rev. 2.0 of the configuration file format)."

Complicated sets of conditions and actions can be constructed and together with an autopilot can run sets of tests.

There will be some changes in scripting as we move to v2.0 of the configuration file format. Among them will be the addition of defaults for some of the settings. For instance, in the set statement, the *persistent* parameter will default to true, the value *type* will default to FG_VALUE, and the *action* will default to FG_STEP. Providing defaults will clean up the scripts considerably. There is also a probablity that the new JSBSim Commander application will feature a script editor, further simplifying the process of creating a script.

▲

Decoupling, Generalizing, and Isolating Commands and Effectors in the JSBSim FDM

The only thing that aerosurface or any other "effector" commands are useful for *within* JSBSim is to allow the aerodynamics algorithms and propulsion systems to determine the forces and moments to apply to the aircraft. There is no direct interaction, for instance, between control inputs and the atmosphere model, or with the equations of motion. Ground reactions are affected slightly by control settings – if the gear is up, the aircraft belly-lands.

Externally supplied control inputs can be handled in two ways in JSBSim: they can directly set the aerosurface *position*, gear *position*, throttle *position*, etc. (which directly affects the aerodynamic forces and moments that will be applied to the aircraft), or they can set representative "interface properties" which can in turn be used to set effector *commands*, which are used to determine effector positions – either by direct assignment, or after being processed by the flight control system.

Control inputs from the pilot perspective (or script, or other user interface) are either "normalized analog" or "normalized binary". For instance, the traditional joystick inputs, pitch command, roll and yaw commands, are analog inputs. These analog inputs range from –1 to +1. An example of a binary input is the gear deployment setting (it's one or zero, though it can transition through any value between the two). The flight control system sets up scaling of the normalized control inputs to the values expected by the aerosurface or by the control laws. For instance, the pitch control laws might expect stick force as input ranging from –25 to +25 pounds.

Some aircraft models may not have spoilers. Some may not have deployable gear. Many aerospace craft have different types and number of engines that propel them. Yet, the superset of possible aerosurfaces is mostly *hardcoded* in JSBSim. That is, there are class members and access functions that represent elevator, rudder, spoiler, flap, aileron, speed brake, etc. A more recent change to JSBSim also gives storage space and access methods to autopilot commands and settings attributes (for internal and scripted use, that is – FlightGear has its own autopilot mechanisms). All these properties, members, and access functions have bloated the FCS code, perhaps needlessly.

With the propulsion system, there can be any number of engines, and even different types. There has been a discussion about how to name the propulsion properties. The general consensus reached is that propulsion system properties *should be* named as follows (for example):

propulsion/engine[0]/throttle
propulsion/engine[0]/starter
propulsion/engine[0]/advance-ratio

The above naming convention, however, introduces a problem, because beginning with v2.0 of the configuration file format, the generic propeller model can define a *function* for the thrust and power coefficients. The function may be a table, which may list a property as an independent variable. That property name cannot be specific to an engine, e.g. *propulsion/engine[1]/advance-ratio*. So, a trick must be done: within an engine or thruster definition file, property names must be non-engine-specific. The code determines which engine to take the value of a property from. For propellers, the advance ratio must be known to determine the thrust coefficient. In the propeller configuration file, the advance ratio is referenced, *propulsion/advance-ratio*. For a multi-engine aircraft, the propulsion code figures out the correct advance ratio to use when determining the thrust coefficient for a specific engine.

A new feature has been added with the latest (v2.0, pending) release of JSBSim that could simplify the complement of Flight Control System properties. *Interface Properties* are properties that are specified in the flight_control section of the configuration file, and which serve to accept commands when running a JSBSim script, or which can also be used by the FGInterface-derived FGJSBSim class in setting control inputs. Interface Properties can be used as the "glue" between the JSBSim FCS and the external, controlling

(Continued on page 10)

Highlighted References

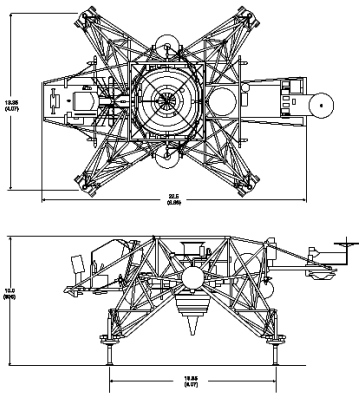
Online:

Boeing 314, Flying Clipper
www.flyingclippers.com/B314.html
www.boeing.com/history/boeing/m314.html

Lockheed Constellation
www.airliners.net/articles/read.main?id=48
www.connie.com/
www.aviation-history.com/lockheed/1049.html

Concorde
www.britishairways.com/concorde/index.html
www.aeroflight.co.uk/types/international/aerospat-bac/concorde/concorde.htm

Lunar Landing Research Vehicle
http://www.astronautix.com/craft/apoollrv.htm



Lunar Landing Research Vehicle (Image: NASA)

Visit us on the web at:
www.jsbsim.org

(Continued from page 9)

application.

Theoretically, in the case where JSBSim is being driven by an internal script (in standalone mode apart from a flight simulation application such as FlightGear), the flight control system class FGFCS would need no direct, hardcoded, attributes or access methods for aerosurfaces. An interface property could be specified in the configuration file, for instance, as an elevator command: "fcs/elevator_cmd". The flight control laws as laid out in the flight_control section of the configuration file would take that property ("fcs/elevator_cmd") as an input, process it, and the last component in the string could set another interface property that represents actual aerosurface *position*. The aerodynamic characteristics specification in the configuration file would refer to the latter interface property when determining the force and moment contribution for that aerosurface.

That all seems cut and dried, but there are more factors to consider. In a full-featured simulation such as FlightGear, the FDM may not only need to supply aircraft state values (position, orientation), but also aerosurface positions for purposes of animation. Also, in the case where JSBSim is used as the flight model for another flight simulation application, the main application may not want to use the property system – it may want to use specific function calls to set and get aerosurface positions.

At this point a decision needs to be made. Should both the property system *and* coded access methods be supported? An investigation will soon be done to determine if the property system can act alone as the method for providing control inputs to JSBSim aircraft flight models. This area of the design is still being debated.

▲

Simulate This! The Lunar Landing Research Vehicle

The Lunar Landing Training Vehicle (LLTV) is a free-flight vehicle consisting of a tubular frame on which a crew station, jet engine, lift rockets, attitude control rockets, control electronics, and associated equipment are mounted. The gimbaled jet engine, which is mounted vertically, provides main power for takeoff and supports five-sixths of the weight of the vehicle during simulation of the lunar environment. The remaining one-sixth is lifted by two 500-pound maximum thrust, throttleable lift rockets to simulate the Lunar Module descent engine. The cockpit includes a Lunar Module three-axis attitude control assembly, the throttle for the lift rockets, a horizontal velocity indicator, the altitude-rate tape indicator, and a thrust-to-weight indicator. Although the pilot of the Lunar Landing Training Vehicle was seated because of the necessity for a rocket-propelled ejection seat, the location of the flight instruments and controls relative to the pilot's hand and eyes was similar to that in the actual Lunar Landing Module.

About this newsletter ...

"Back of the Envelope" is a new communication tool written for a wider audience than core JSBSim developers, including instructors, students, and other users. The articles featured will likely tend to address questions and comments raised in the mailing lists and via email. If you would like to suggest (or even author) an article for a future issue, please email the editor at: jsb@hal-pc.org.