

JSBSim, A Flight Dynamics Model

The Purpose

The FlightGear project (www.flightgear.org) is an ongoing effort to provide a state of the art, open source, multi-platform flight simulator. One key component of a flight simulator is the Flight Dynamics Model (FDM). The FDM takes control inputs (e.g., aileron, elevator, speedbrake, throttle, etc.) and determines how a simulated aircraft responds to this input.

Originally, FlightGear had only a single FDM, called LaRCSim (Langley Research Center Simulation). It was written in C, and modeled a small propeller-driven aircraft. To model additional aircraft using LaRCSim, new code had to be written that described the flight characteristics and dynamics of the specific aircraft. Modeling a hangar full of aircraft of different types would thus be a tedious process.

The Practice

An opportunity existed to create an FDM for FlightGear that was highly configurable, maintainable, expandable, robust, and instructive. But which language should one use to forge it? The space shuttle mission and engineering simulators and the Air Force F-16 trainer simulator used Fortran as the language of choice nearly by default, having been created ten to twenty years ago (or more). The more recent Space Station Training Facility at Johnson Space Center is mostly written in Ada. I remember one discussion I had regarding the potential advantages offered by C++ over Fortran with a senior, experienced Fortran programmer. He expressed doubt regarding whether or not C++ was an appropriate language for modeling flight dynamics due to performance concerns. His prejudiced opinion was not unique - though I believe it was inaccurate. The flight dynamics model takes up a relatively small percentage of the total execution time in FlightGear. Most of the time is spent rendering scenery. With today's speedy computers and careful coding and design, C++ works well. Indeed, NASA-Langley engineers several years ago began writing the successor to LaRCSim. They call it LaSRS++, and they chose C++.

The C++ language provides features that are conducive to the needs of aircraft simulation, such as polymorphism, inheritance, and encapsulation.

The Solution

JSBSim is intended to meet these goals:

1. To allow different aircraft to be modeled without writing new code
2. To be open source and run on any platform
3. To be instructive to students of aerospace engineering (both as a tool in studying flight dynamics and as a showcase for some of the equations)
4. To be executable in both standalone mode and integrated with FlightGear (or any other flight simulator)

The Framework

Simulating the flight of an aircraft requires modeling nature (the atmosphere, winds, gravity) and machine (engines, landing gear, wings, weight, etc.). The aim is to calculate all the forces that affect a simulated aircraft and to add them up. Specific simulation models are derived from a generic base class called FGModel (the FG prefix refers to FlightGear):

FGModel (base):

- ▶ FGAtmosphere (atmosphere)
- ▶ FGFCS (flight control system)
- ▶ FGPropulsion (engines, propellers, etc.)
- ▶ FGMassBalance (weight and balance)
- ▶ FGAerodynamics (aerodynamic effects)
- ▶ FGIInertial (Coriolis acceleration, etc.)
- ▶ FGGroundReactions (landing gear, contact points)
- ▶ FGAircraft (aircraft model as a whole)
- ▶ FGTranslation (translational motion)
- ▶ FGRotation (rotational motion)
- ▶ FGPosition (integrates velocities to get position)
- ▶ FGAuxiliary (various additional calculations)
- ▶ FGOutput (provides for various output features)

Each of these classes overloads an FGModel::Run() method to do model-specific calculations, but within the overloaded Run() function the base class Run() is also called to do tasks common to all models.

To coordinate all these classes there is one manager (or Executive) class, FGFDMExec, that handles the ordered creation and scheduling of classes and runs them as needed. Instantiating the FGFDMExec class sets in motion the creation of an instance of JSBSim, as FGFDMExec encapsulates the top-level functionality needed to create and run the simulation. Minimal work needs to be done to set up JSBSim in standalone mode:

```
FGFDMExec* Exec = new FGFDMExec();
Exec->LoadScript(argv[1]);
while (Exec->Run()) {}
delete FDMExec;
```

To integrate with FlightGear is just slightly more complicated. FlightGear has a class called FGInterface, and from that are derived classes for each of the FDMs used. The FGJSBSim class (which we refer to as the "bus" because of the way it acts as a data interface device) instantiates JSBSim initially, and sets the aircraft up in a trimmed (steady and stable) state. During normal run time it also copies the control data from FlightGear to JSBSim, calls the Exec->Run() function to execute an FDM frame, and finally copies location and state information back to FlightGear from JSBSim.

Many Parts

To satisfy the first goal one big issue comes to mind: any given aircraft can have a specific number of wheels, engines, fuel tanks, and so on. The engines can be piston engines with propellers, turbo shaft engines with propellers, rocket engines with nozzles, or a combination. JSBSim must store an arbitrary number of engines, where each can be one of several types. A *thrust* acts as the vehicle through which the engine provides a force, and each engine must have a thruster paired with it. Similarly, the flight control system can be made up of any number and variety of components. Also, the aerodynamic characteristics of a simulated aircraft can be described by any number of aerodynamic coefficients, which in turn can be calculated as the product of several user-specified parameters.

The STL vector is used in JSBSim anywhere storage of generic objects is required and the number of objects to be stored is unknown at compile time. Vectors provide storage for generic items such as engines, aerodynamic coefficients, thrusters, tanks, and flight control system (FCS) components.

The concept of polymorphism is exploited in JSBSim. Two or more objects are said to be polymorphic if they have identical interfaces and different behaviors¹. The generic FGEngine class specifies the interface to all engine types, and is the base class for the derived types that model specific engine behavior (see FGEngine class heirarchy, below). The same can be said of the base class for the FCS components and the derived specific components.

The concept of aggregation is also apparent. The FGPropulsion class has a vector of objects which descend from the base class FGEngine. It also contains vectors of FGTank- and FGThrust-erived objects. The FGPropulsion class encapsulates the entire propulsion system: fuel tanks, engines, and thrusters. As far as the JSBSim Executive is concerned, the only thing it needs to do is call FGPropulsion::Run(). Similarly, the FGAerodynamics class contains (and manages) vectors of aerodynamics coefficients; the FGFCS class contains a vector of FCS components.

Each of the components described above is modeled as an object itself. The class layout for supporting objects is shown here:

```
FGForce
  FGThrust
    FGPropeller
    FGNozzle
    FGRotor

FGEngine (engines)
  FGPiston
  FGRocket
  FGTurboJet
```

¹ Martin, Robert C., *Designing Object-Oriented C++ Applications Using the Booch Method*, Prentice Hall, 1995

FGTurboShaft
FGTurboProp

FGLGear (landing gear)
FGCoefficient (aerodynamic coefficients)
FGTank
FGFCSComponent (flight control components)
FGFilter
FGSwitch
FGGradient
FGSummer
FGDeadBand
FGGain

Each of the parts of the aircraft operates itself. For example, an instance of the propeller class - knowing its own characteristics - calculates how much power it requires to keep turning it at its current spin rate. The propeller instance supplies this value to its associated engine instance that in turn calculates how much excess power is available. The engine instance returns the excess power to the propeller instance and if this excess power is non-zero, the propeller accelerates or decelerates its spin rate.

Configuring the Simulation

JSBSim assembles an aircraft as a collection of parts as specified in the configuration files. The files tell the program about the characteristics of the aircraft being modeled. The configuration file format bears a resemblance to XML (the eventual goal is for XML compliance). A segment of the configuration file for the X-15 research aircraft is shown here:

```
<AC_TANK TYPE="FUEL" NUMBER="1">
  XLOC 408.3
  YLOC 0.0
  ZLOC 0.0
  RADIUS 26.0
  CAPACITY 8236.0
  CONTENTS 8236.0
</AC_TANK>
...
<AC_ENGINE FILE="XLR99">
  ...
</AC_ENGINE>

<AC_THRUSTER FILE="xlr99_nozzle">
  ...
</AC_THRUSTER>
```

When JSBSim parses the configuration file, the tanks, the engine, and the nozzle described in the files are created on-the-fly and pushed onto a vector of tanks, a vector of engines (specific engine types are descended from the generic FGEngine type), and a vector of thrusters, respectively. Later during runtime the vectors are iterated through and each member executes its own relevant calculations. Ultimately, forces generated by each force-generating object (such as a wing, a propeller, a nozzle, gravity, etc.) are all summed and all the effects on the aircraft are integrated.

The use of a vector much simplified the piecewise construction of the aircraft. Without it, the lists of engines, wheels, fuel tanks, etc. would have been constructed perhaps using an array or linked list. In my opinion, the vector works nicely and makes the code easier to read.

An example follows illustrating the special design problems posed by the FCS subsystem because of the variety and number of other components it needs to manage.

Flight Control

JSBSim contains a set of classes that together can simulate a complex flight control system (FCS) based on information specified in the aircraft configuration file. The FCS can be as simple as a scaling factor (a gain) multiplied with the joystick input to arrive at a control surface deflection command, modeling something simple that might be found on a light aircraft. JSBSim can also simulate a more complex FCS as would be found in a modern jet fighter, containing digital filters, scheduled gains, aircraft sensor inputs, etc.

The FCS for an aircraft typically features several sequential components. The pilot command is input to the first of the components in line. The output from the last component in the sequence is sent to a control surface such as the elevator. Between the two endpoints, the output from one component is the input to the next. The FCS sums various other values into the path along the way. Values can be any one of several aircraft parameters such as angle of attack, rotational rates, airspeed, etc.

To model an FCS, we created a set of classes that represent each type of FCS component. The section of the configuration file that describes the FCS contains definitions for each component. A component definition includes the type of component, an ID number, the input (or inputs) to it, and a destination if the component is the final one in the sequence. For the X-15 example, part of the FCS definition representing the pitch channel is shown here:

```
<COMPONENT NAME="Pitch Trim Sum"
TYPE="SUMMER">
  ID 0
  INPUT FG_ELEVATOR_CMD
  INPUT FG_PITCH_TRIM_CMD
  CLIPTO -1 1
</COMPONENT>

<COMPONENT NAME="Pitch Command
Scale" TYPE="AEROSURFACE_SCALE">
  ID 1
  INPUT 0
  MIN -50
  MAX 50
</COMPONENT>
```

When JSBSim builds the FCS model (when the configuration file is parsed in the FGFCs::Load() function), the components described in the configuration file are constructed and pushed onto the component vector. The specific type of component created depends on the type as specified in the component definition:

```
if (token=="LAG_FILTER") {
  Components.push_back(new
    FGFilter(this, AC_cfg));
} else if (token == "SUMMER") {
  Components.push_back(new
    FGSummer(this, AC_cfg));
} else if (token == "SWITCH") {
  ...
} else {
  cerr << "Unknown token ["
    << token << "]" in FCS
    portion of config file"
    << endl;
  return false;
}
```

When the FGFCs::Run() function is executed each FCS component that exists in the vector of components executes its own Run() function:

```
for (i=0;i<Components.size();i++)
  Components[i]->Run();
```

In the case of the first component described in the example configuration file portion, above, the underlying object is an FGSummer type. This component adds together the specified input parameters. Any of a number of predetermined parameters can be specified in the configuration file using the identifier (FG_ELEVATOR_CMD and FG_PITCH_TRIM_CMD in this case). The actual parameter value can be retrieved by using the FGState::GetParameter() function and passing the parameter string as the argument.

Another type of FCS component is the filter:

```
<COMPONENT NAME="Elevator Filter"
TYPE="LAG_FILTER">
  ID 16
  INPUT 15
  C1 600
  OUTPUT FG_ELEVATOR_POS
</COMPONENT>
```

The input here is seen to be an actual number. This means that the input is another FCS component, and it is one that has ID 15. See the sidebar "The Tustin Substitution" to see how we model filters in JSBSim.

The Tustin Substitution

Filters are modeled in JSBSim using a mathematical tool called the Tustin substitution. Filters can be thought of as devices which process the input in some way and produce a desirable output. A filter in a control system diagram would typically be given as an equation in LaPlace space. For instance, the transfer function (showing the ratio of the output divided by the input) for a lag filter might look like this:

$$\frac{O(s)}{I(s)} = \frac{600}{s+600}$$

The Tustin substitution determines how a filter may be simulated in a situation with discrete time steps. The Tustin substitution is accomplished by replacing the "s" in the above equation with:

$$\frac{2(Z-1)}{T(Z+1)}$$

The "T" in the equation above represents the integration time constant. JSBSim typically executes at 120 Hz when running with FlightGear so T is 0.00833. When the substitution is carried out the lag filter equation becomes an equation in Z-space:

$$\frac{O(Z)}{I(Z)} = \frac{600}{\frac{2(Z-1)}{T(Z+1)} + 600}$$

If the O(z) and I(z) terms are gathered so as to eliminate the denominator the equation looks like:

$$O(Z) = I(Z)*CA + I(Z)^{-1}*CA + O(Z)^{-1}*CB$$

The I(Z) and O(Z) with exponents act as operators referring to past values. I(Z)⁻¹ refers to the previous value of the input, O(Z)⁻² refers to the second previous

output, and so on. The CA and CB are coefficients that represent parts of the longer equation. In this case they are:

$$CA = \frac{600T}{(2 + 600T)}$$

$$CB = \frac{(2 - 600T)}{(2 + 600T)}$$

Since the time step is fixed at startup the coefficient values can be calculated once at initialization and the equation is finally given as:

```
Out = In*CA+PrevIn*CA+PrevOut*CB
```

Multi-Platform

JSBSim, like FlightGear, is compilable under several operating systems including Microsoft Windows (using the CygWin environment, Borland, or Microsoft compilers), SGI IRIX, gcc under Linux, and on the Macintosh. Several issues have been uncovered while working towards this multi-platform capability:

- Path name separators under the supported operating systems can be one of “/”, “\”, or “\”. JSBSim uses conditional compilation to provide compatibility with all of these.
- Most compilers come with headers that define the various math constants, e.g. M_PI, while some do not (these math definitions are part of an older BSD standard). Conditional compilation is used by JSBSim to #define M_PI where it is not defined already.
- The following code causes an error in MSVC:

```
int myfunction(void) {
    for (int i=0; i<10; i++) {
        // do some operations
    }
    for (int i=0; i<10; i++) {
        // compiler chokes, here,
        // saying that "i" is
        // defined twice.
    }
}
```

This is easily dealt with by simply declaring the variable within the function scope instead of within the for statement.
- Some compilers support the inclusion of standard and STL headers specified without the “.h” extension, some do not. JSBSim uses conditional compilation blocks to get around this.
- Compilers differ slightly in their standard library implementation. The solution used by JSBSim is to find the commonality where possible, but use conditional compilation otherwise.
- Line endings in text files are different under DOS, Unix, and Macintosh. JSBSim uses a custom parser to read configuration files.
- Some compilers are tolerant of sloppiness with respect to namespaces. The solution that works for JSBSim is to always specify the namespace used. For instance:

```
using std::cout;
using std::string;
or,
using std;
```

Matrix Math

One of the features of C++ that helped achieve the third goal of JSBSim is operator overloading. Matrix math is used extensively in JSBSim. Matrix math can look messy, with indexing and looping sometimes necessary to perform calculations. The FGMatrix class has overloaded operators that allow matrix/vector equations to be coded so they look similar to textbook presentations of the corresponding equations.

For instance, in JSBSim once all the forces that act on a simulated aircraft are known the acceleration of the aircraft can be calculated using a variation on the easily recognizable $F=ma$ relationship. The expanded equation would be written in a textbook as:

$$\bar{a} = \begin{bmatrix} 0 & -W & V \\ W & 0 & -U \\ -V & U & 0 \end{bmatrix} \times \bar{u} + \frac{\bar{F}}{m}$$

The representation for this in JSBSim is:

```
vUVWdot = mVel*vPQR + vForces/Mass;
```

In the equation above, vUVWdot represents a column vector with three elements, as do F and \bar{u} . Notably missing in the equation as coded are the explicit iterations through the column vector for each of the three elements of the column vectors.

The inclusion of matrix operators for the usual arithmetic functions and also for matrix-specific functions such as the cross product significantly increase readability.

To Do, and Closing Thoughts

Although JSBSim has worked as a flight dynamics model for some time, there are several things that need to be done before it becomes the default FDM for FlightGear. The matrix class needs to be optimized. The landing gear model needs to be refined. The propulsion models need to be expanded.

JSBSim has met its goals. JSBSim can simulate any aircraft for which we have characteristic data. It can be compiled on several platforms. Students can gain valuable insight into flight dynamics by both studying the source code or using JSBSim as a tool to investigate flight dynamics and control. A simple plotting tool is also provided in the JSBSim distribution to facilitate analysis of data output by JSBSim. Finally, JSBSim has been integrated with FlightGear as a selectable FDM. JSBSim can also be run in a standalone mode for scripted runs or debugging purposes.

Further information on JSBSim can be found at the web site: <http://jsbsim.sf.net>.

About the author:

Jon Berndt is the coordinator and chief architect for JSBSim. His real job is with Lockheed Martin as a simulation engineer supporting NASA on the X-38 project. Jon would like to acknowledge JSBSim co-author Tony Peden for his extensive contributions, and also the FlightGear team for providing assistance.