Automatic Flight in JSBSim

Jon Berndt[1]

Navigation – Where am I?
Guidance – How do I get where I want to go today?
Control – Go there.

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 fancy visuals) and fly in a stable manner to various targets, be they altitude and heading, or latitude and longitude, etc. This would be a useful feature for many reasons, among them being regression testing of JSBSim, aircraft flight model performance testing, and control system development.

This goal has been achieved in the most recent releases of JSBSim. Some of the recent features that have made this capability possible include the addition of switch and function components, sensors, new autopilot-related properties, and various other small fixes to the flight control components of JSBSim.

Overview

In implementing automatic flight in JSBSim, several files are involved (all are XML format):

- 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).

Once the autopilot is designed, it must be implemented in the JSBSim file format, referred to as JSBSim-ML (for JSBSim Markup Language). The autopilot for JSBSim uses the same components that are used in defining the flight control system (FCS). JSBSim features these FCS components:

-Filter (lag, lead-lag, washout, etc.)
-Integrator
-Switch
-Summer
-Gain (including scheduled gain)
-Deadband
-Limiter
-Kinematic
-Function
-Sensor

Mechanization of Components

Each FCS component is discussed below. Note that in many cases "properties" are referred to in the text and in the definition of the component. Properties will be explained later on, 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.

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:

[1] Jon Berndt is the JSBSim Chief Architect and Development Coordinator

```
<typename name="name">
  input <property>
  c1 <value>
  [c2 <value>]
  [c3 <value>]
  [c4 <value>]
  [c5 <value>]
  [c6 <value>]
  [<clipto>
     <min> number </min>
     <max> number </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 </c1>
  [<clipto>
     <min> number </min>
     <max> number </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 </c1>
  <c2> number </c2>
  <c3> number </c3>
  <c4> number </c4>
  [<clipto>
     <min> number </min>
     <max> number </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 </c1>
  [<clipto>
     <min> number </min>
     <max> number </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 </c1>
  <c2> number </c2>
  <c3> number </c3>
  <c4> number </c4>
  <c5> number </c5>
  <c6> number </c6>
  [<clipto>
     <min> number </min>
     <max> number </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 </c1>
  [<trigger> property </trigger>]
  [<clipto>
     <min> number </min>
     <max> number </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.

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 </min>
     <max> number </max>
```

```
   </clipto>]
  [<output> property </output>]
</switch>
```

Here's an example:

```
<switch name="Roll A/P 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.

Summer Component

The Summer component sums two or more inputs. These can be pilot control inputs or state variables, and 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 </min>
     <max> number </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="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.

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 </min>
     <max> number </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.

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 </min>
     <max> number </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 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="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.

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 name | value} </min>
    <max> {[-]property name | value} </max>
  </clipto>]
  [<gain> {property name | 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="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, we see the utility of the overall GAIN value in effecting a degrees-to-radians conversion.

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="Windup Trigger">
  <input> [-]property </input>
  <width> number </width>
  [<clipto>
    <min> {[-]property name | value} </min>
    <max> {[-]property name | 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.

Hysteresis Component

[not yet modeled]

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.

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)

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.

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="Gear Control">
  <input> [-]property </input>
  <traverse>
    <setting>
      <position> number </position>
      <time> number </time>
    </setting>
    …
  </traverse>
  [<clipto>
    <min> {[-]property name | value} </min>
```

```
    <max> {[-]property name | value} </max>
  </clipto>]
  [<gain> {property name | value} </gain>]
  [<output> {property} </output>]
</kinematic>
```

The detent 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="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 should be an OUTPUT specified.

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="Windup Trigger">
  [<input> [-]property </input>]
  <function>
    …
  </function>
  [<clipto>
    <min> {[-]property name | value} </min>
    <max> {[-]property name | 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>
```

Sensor Component

The sensor component is a recent addition to JSBSim. Previously, when a flight control system used “sensed” values such as qbar or orientation, the values were effectively taken from “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.

So, the sensor component was created that can model imperfections and increase 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 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 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”}> number </noise>
  <quantization name="name">
    <bits> number </bits>
    <min> number </min>
    <max> number </max>
  </quantization>
  <drift_rate> number </drift_rate>
  <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>
```

Properties

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. They are denoted by a categorized hierarchy that looks a little bit like a directory/file structure.

Many properties are static 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="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="Pitch Command Scale">
            <input>fcs/pitch-trim-sum</input>
            <range>
                <min>-50</min>
                <max>50</max>
            </range>
        </aerosurface_scale>

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

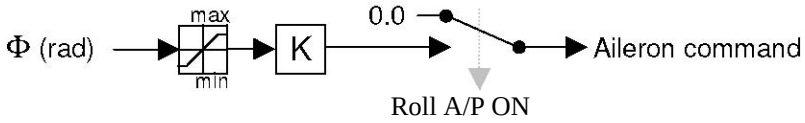
A close examination of the above series of components reveals a hint as to how the property names are given to components that are defined at runtime. The first component above ("Pitch Trim Sum") takes input from two places, the known static properties, *fcs/elevator-cmd-norm* and *fcs/pitch-trim-cmd-norm*. 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*. The convention for automatic naming of properties for the flight control system (FCS) and autopilot is that all non-alphanumeric characters in a component name are replaced by hyphens ("-"). This includes spaces and slashes. Additionally, all letters are made lower-case. Continuing with the above case shows that the last component, "Pitch Gain 1", takes as input the output from the preceding component, "Pitch Command Scale", which is given the property name (being in the FCS model), 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. That's the next step.

Autopilot Basics

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.

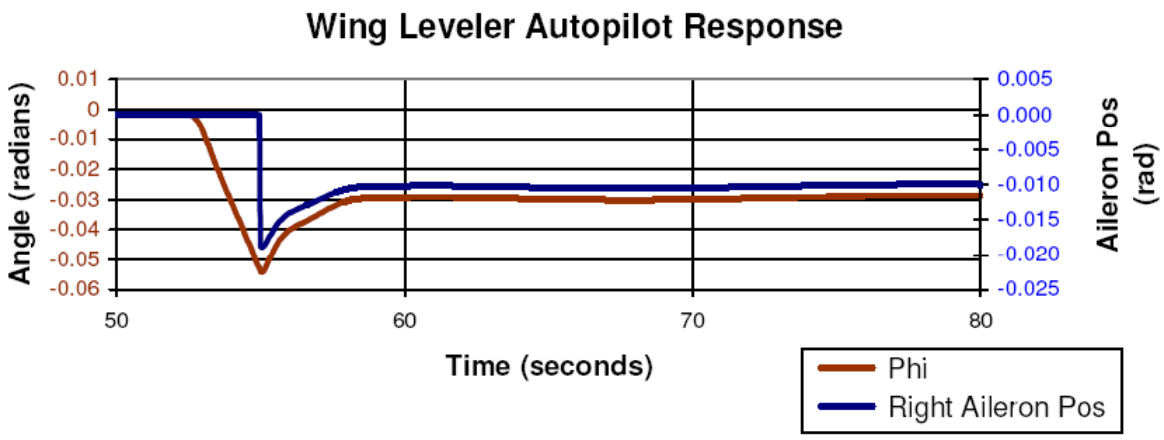
One possibility is to simply command the ailerons based on the roll angle. This is called proportional control, because the output is simply the input multiplied by a value – the output is proportional to the input. The autopilot aileron command is sent to the main flight control system and summed in 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 below:



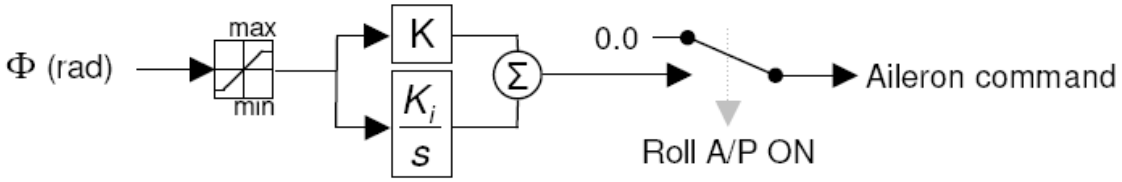
Also, the min, max, and K values must be chosen properly. As an example, we set up JSBSim to run with this autopilot and see what the response is. Here is the autopilot we use, in JSBSim format:

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

  <pure_gain name="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="Roll AP Autoswitch">
    <default value="0.0"/>
    <test logic="AND" value="fcs/roll-ap-wing-leveler">
      ap/attitude_hold == 1
    </test>
  </switch>

  <pure_gain name="Roll AP Aileron Command Normalizer">
     <input>fcs/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 *Roll A/P 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 *Roll A/P 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:

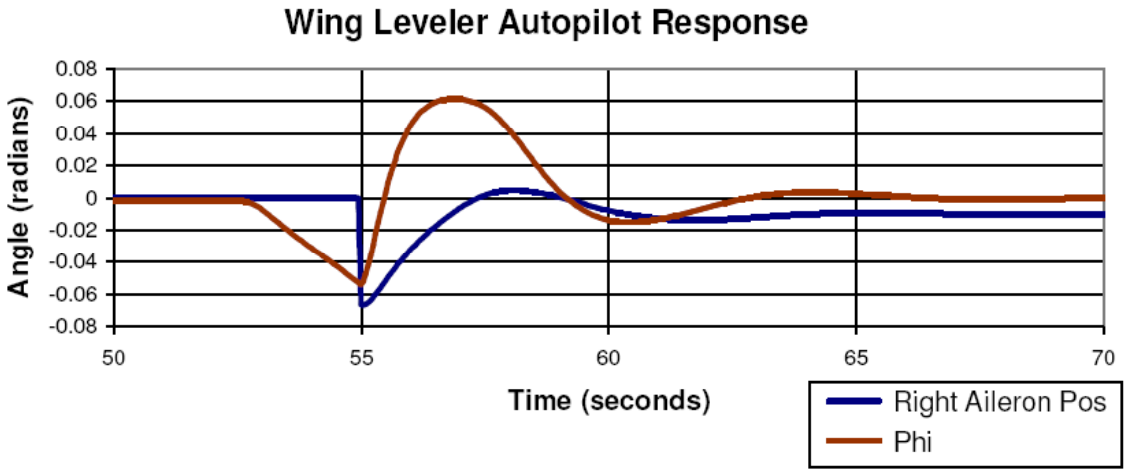


This block diagram of a wing leveler using PI control is implemented in JSBSim-ML as follows:

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

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

  <pure_gain name="Roll AP Wing Leveler">
    <input> fcs/limited-phi </input>
    <gain>2.0</gain>
  </pure_gain>

  <integrator name="Roll AP Error Integrator">
    <input> fcs/limited-phi </input>
    <c1> 0.125 </c1>
  </integrator>

  <summer name="Roll AP Error summer">
     <input> fcs/roll-ap-wing-leveler</input>
     <input> fcs/roll-ap-error-integrator</input>
     <clipto>
       <min>-1.0</min>
       <max> 1.0</max>
     </clipto>
```

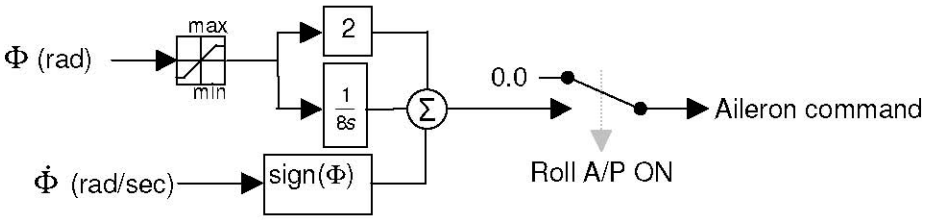
```
  </summer>

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

  <pure_gain name="Roll AP Aileron Command Normalizer">
     <input>fcs/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:



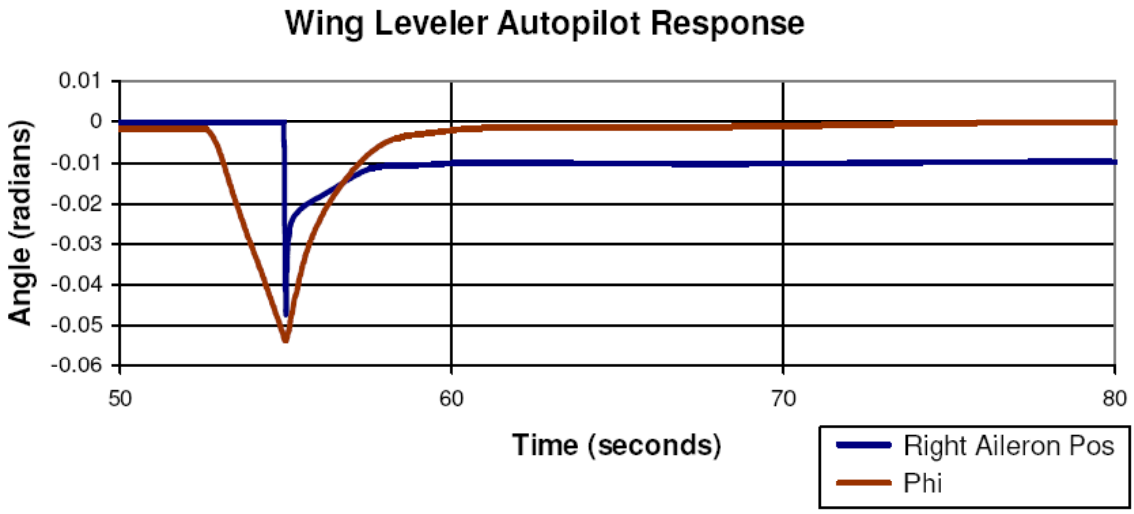
The roll angle is seen in the above graph to settle out, now, to wings level. There is, however, a good degree of overshoot initially. This is where the tweaking comes into play. One can vary the relative contributions of the proportional and integral parts of the control command to achieve different 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="Limited Phi">
    <input> attitude/phi-rad </input>
    <clipto>
      <min>-0.255</min>
      <max>0.255</max>
    </clipto>
  </pure_gain>

  <pure_gain name="Roll AP Wing Leveler">
    <input> fcs/limited-phi </input>
    <gain>2.0</gain>
  </pure_gain>

  <integrator name="Roll AP Error Integrator">
    <input> fcs/limited-phi </input>
    <c1> 0.125 </c1>
  </integrator>

  <summer name="Roll AP Error summer">
     <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>

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

  <pure_gain name="Roll AP Aileron Command Normalizer">
     <input>fcs/roll-ap-autoswitch</input>
     <gain>-1</gain>
  </pure_gain>
</channel>
```


Wing Leveler Autopilot Response
0.01
0
-0.01
-0.02
-0.03
-0.04
-0.05
-0.06
Angle (radians)
50
60
70
80
Time (seconds)
Right Aileron Pos
Phi