

AN OBJECT-ORIENTED DESIGN FOR TRIM

Jason R. Neuhaus*

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

Abstract

This paper presents an Object-Oriented Design for trimming dynamic models. By applying Object-Oriented Programming techniques to trim, in conjunction with an object-oriented model structure, generic trim rules can be developed and shared between different model types. This trim design can be used to trim many types of models, including automobiles, aircraft, spacecraft, and rotorcraft. Class hierarchies and interactions between the model, trim, and trim rule classes, as well as a general overview of Object-Oriented Programming, are presented.

Introduction

The objective of a trim routine in simulation is to modify the states of a vehicle until equilibrium is reached. For aircraft, this equilibrium is typically defined as the set of states for which all body axis translational and angular accelerations are zero. Trim is used to properly initialize an aircraft so that flight can begin in any flight situation and maintain an initial set of specified states after the simulation switches into run/operate mode. For example, trimming on the ground involves adjusting the height of the aircraft until the vertical acceleration reaches zero. This is done so the aircraft does not fall, or the landing gear are not so compressed to cause the aircraft to jump, when the simulation is switched into the operate mode. In another example, the aircraft throttle(s), aileron, rudder, and elevator can be adjusted so that the aircraft maintains straight and level flight with no pilot inputs after the simulation begins running.

*Aerospace/Software Engineer, Member
 Copyright © 2000 by the American Institute of Aeronautics and Astronautics, Inc. No copyright is asserted in the United States under Title 17, U.S. Code. The U.S. Government has a royalty-free license to exercise all rights under the copyright claimed herein for Governmental purposes. All other rights are reserved by the copyright owner.

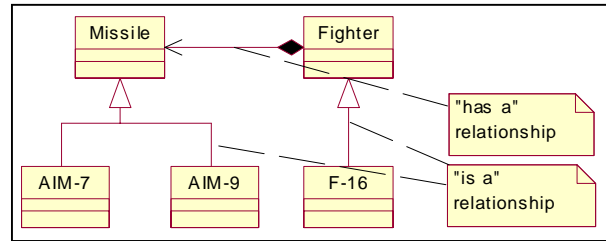


Figure 1 Simple UML Class Diagram

Object-Oriented Design

An Object-Oriented Design is made up of classes. Each class encapsulates a well-defined set of tasks and provides an interface for other classes to interact with it. Each class contains a set of methods (functions) and member data that define the behavior of the class. Through encapsulation, the class becomes a black box where only a public interface is available. This allows classes to limit the access of methods and member data to other classes. Methods and data can be defined as public, protected, or private. Public methods are accessible to any class. Protected methods only allow derived classes to access the method. Private methods and data are accessible only to that class.

In addition to defining their own methods, classes can share and overload[†] the methods of other classes through use of inheritance. Inheritance defines a relationship between classes, wherein one class shares the structure and/or behavior of another class [1]. The parent class is the class from which inheritance is taking place. The class that is inheriting from the parent class is the derived class. Inheritance is often referred to as an "is a" relationship. Figure 1 shows a simple Unified Modeling Language (UML)[2] class diagram depicting the relationships between a fighter and missile classes. The AIM-9 is a derived class, which inherits from the Missile class, the parent class. An AIM-9 "is a" missile.

[†] redefine or add additional functionality

An object is a particular instance of a class. When an object is created, it maintains all of its own data separate from all other objects of the same type[‡]. In this way, an arbitrary number of copies of a class can be created and used without data being mixed between the different instances of a particular class. Multiple fighter objects can be created, where each maintains its own private data members, such as altitude, orientation, accelerations, etc.

Any place where a parent class is used, a derived class can be substituted because of inheritance. For example, if a **Fighter** class contains **Missiles**, “has a” relationship, any class that derives from the **Missile** class can be used in place of the **Missile** class used by the **Fighter**. An AIM-9 can be used because the AIM-9 “is a” missile. The AIM-9 class contains all the functionality of the **Missile** class, plus the additional functionality specific to that particular type of missile. The **Fighter** class can only call public methods that are declared in the **Missile** class. Additional methods declared in the derived classes of the **Missile** class cannot be accessed, because the parent class has no knowledge of the data and methods declared in classes that derive from it.

Polymorphism is the overloading of a method of a parent class in a derived class. It is used to extend or replace functionality in the parent class. Methods that can be overloaded are marked by the virtual keyword in C++. The correct overloaded method to call is decided at run time. The correct method to call is not decided at compile time because the method to call depends on the exact type of the object being accessed, which, because of inheritance, may be the parent class, or any classes that derive from it. For example, assume the **Missile** class contains a public virtual method named *update* that performs some function. The AIM-7 class overloads the *update* method to add AIM-7 specific code, but the AIM-9 class does not overload *update*. If the fighter object has both an AIM-9 and an AIM-7 object, and calls *update* on both, the *update* method from the **Missile** class will be called for the AIM-9 object, but the overloaded *update* method will be called for the AIM-7 object. This occurs even though the fighter object has no knowledge what types of missile objects are being used.

[‡] Data *can* be shared between instances of a class, but is beyond the scope of this paper.

LaSRS++ Trim Classes

The following classes are used in the Langley Standard Realtime Simulation in C++ (LaSRS++)[3] trim design and are shown in Figure 2.

Vehicle Class

A vehicle’s states propagate through time in reaction to external forces and moments. The forces and moments determine the translational and angular accelerations of the vehicle.

Aircraft Class

The **Aircraft** class inherits from the **Vehicle** class. An aircraft is a vehicle that flies through an atmosphere. The dynamics of the aircraft are primarily influenced by aerodynamic, engine, and gear forces and moments.

TrimRule Class

The **TrimRule** class defines relationships between trim states, errors, and gains used to trim a vehicle object. The trim rule states and errors are based on vehicle data. Trim states are modified based on errors and gains, until the error is within a specified tolerance. Each trim rule can have multiple sets of states, errors, and gains as well as constraints. A constraint modifies a state to a predetermined value that can be a function of other states or a constant value.

AircraftTrimRule Class

The **AircraftTrimRule** class inherits from the **TrimRule** class. It adds the ability to access aircraft data, in addition to vehicle data, for use in trim states and errors. Some of the LaSRS++ aircraft trim rules are listed at the end of this paper.

Trim Class

The **Trim** class is responsible for performing all the basic trim operations on a vehicle. These operations include selecting trim rules, checking for rule conflicts, setting target values for body translational and angular accelerations, and finding a trim solution.

A trim solution is found by operating on a list of trim rules until a solution is found or problems finding a solution are detected. Some of the conditions checked to determine if a solution could not be found are if the iteration count reaches a preset maximum and no solution is found, or an error value reaches a steady state value that is not within the specified tolerance. In the event of a problem locating a solution, trim states and errors are output to aid in determining which states had trouble trimming.

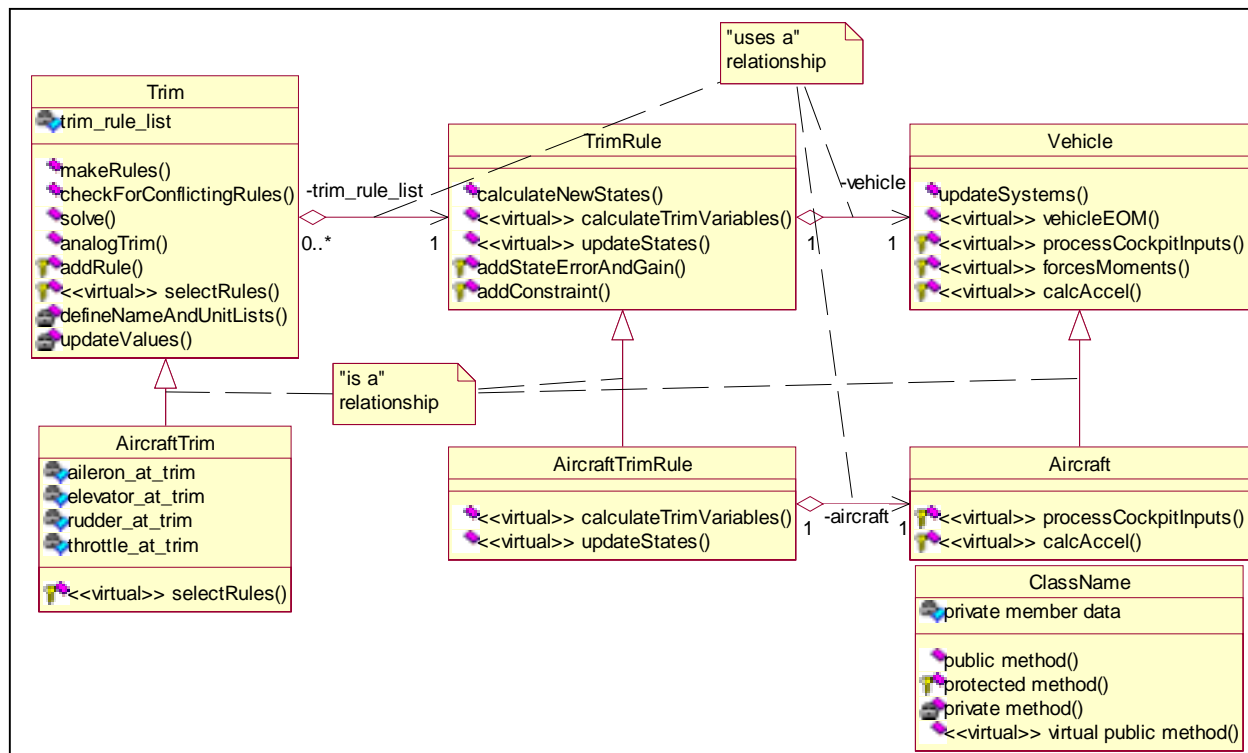


Figure 2 LaSRS++ Trim UML Class Diagram

AircraftTrim Class

The AircraftTrim class inherits from the Trim class. It selects aircraft trim rules based on the desired aircraft initialization. Four additional states exist in the AircraftTrim class for the purpose of overriding cockpit values while trimming. These four data members are *aileron_at_trim*, *elevator_at_trim*, *rudder_at_trim*, and *throttle_at_trim*. These are used as generic inputs for roll rate, pitch rate, yaw rate, and propulsive thrust respectively. Each different aircraft maps the four generic inputs to inputs specific to that aircraft. Depending on the control system, *elevator_at_trim* may drive the elevator position directly, and a separate trim variable drives the pitch stick. This is often done for α , N_z , and pitch rate command systems. Once the aircraft is trimmed, any offsets in control positions can be back driven into the hardware, displayed so the pilot can match the inputs, or cockpit inputs can be biased based on the trimmed values.

The AircraftTrim class does not contain any logic for trimming. All of the trimming logic is contained in the parent class, Trim. Through inheritance, the AircraftTrim class can use these same methods. In this way, the methods in Trim can be used on both vehicles and aircraft.

Trim Algorithm

Each simulation object uses either a trim or an aircraft trim object to trim itself, depending on the object type. When the simulation is switched into trim mode, all the simulation objects are trimmed in sequence. The following sections give the basic outline of the trimming procedure.

Rule Setup

Before the simulation objects are trimmed, each must setup initial values and chose the rules that will be used to trim the object. Typically, rules are chosen that zero out all body translational and angular accelerations. How each of the body translational and angular accelerations are trimmed to zero is dependent on the rules that are selected. It is sometimes desirable to trim an acceleration term to a known non-zero value, or not to trim it at all. Trimming to non-zero accelerations is often done to match old data or third-party test cases. An acceleration term might not be trimmed at all on an aircraft without any engines; where there is no thrust to trim out the body-x axis acceleration.

Trim Loop

After the trim rules have been setup, the solution process begins. A trim solution is found by looping until a solution is found, a maximum iteration count has been reached, or the simulation exits trim mode. A maximum iteration count is hit when one or more error signals did not converge to zero. This usually occurs when an error does not converge fast enough, converges, but oscillates around zero, or reaches a steady state value that is not within the specified tolerance. An error that converges too slowly usually indicates that the magnitude of a gain that was too small. An error that converges quickly, but oscillates around the solution, typically results when the magnitude of a gain was too large. Warnings are printed if trim believes it has encountered a steady state error. A steady state error usually indicates that one or more variables have reached a limit, for example, a control surface deflection limit.

After the simulation object has been successfully trimmed, the trim routine exits and the aircraft is prepared to operate at the flight condition specified.

Methods

Figure 3 shows a UML sequence diagram[2] listing the events that occur when trimming an aircraft. The methods called are described below.

Trim Methods

makeRules

Before trim is initiated, the user selects the trim rules to be used. This forms the *trim_rule_list* on which the trim object will operate.

selectRules

This method is called by the *makeRules* method and is responsible for determining which rules the user has selected. Once the rules have been identified, they are added to the *trim_rule_list* using the *addRule* method.

defineNameAndUnitLists

Once the list has been formed, the names and units of the states, errors, and gains of all rules on the list are gathered and stored. The names will be used to ensure that two conflicting rules are not selected, and for output purposes.

checkForConflictingRules

This method warns the user if two conflicting rules are selected. Conflicting rules may prevent the vehicle

from trimming or may result in a condition where multiple solutions exist. Two rules conflict if either the same state is modified or the same error is monitored by multiple rules. No preventive action is taken in the event of a conflict. Combinations of states and errors that do not directly conflict are not detected. For example, selecting trim rules that specify angle of attack, pitch angle, and flight path angle values would cause a conflict but not be detected.

solve

Before trimming begins, the simulation object is set up at the user-defined conditions. This is accomplished by having each trim rule initialize its internal states to the simulation object's initial states. Then all the trim rules update the states in the simulation object with their internal states, including any constraints. This ensures that constraints are set up properly in the first pass, and is accomplished through calls to the trim rule methods, *calculateTrimVariables* and *updateStates*.

updateValues

Calling this method copies all the trim state, error, and constraint values from the individual trim rules into the trim object. This is done so that the data can be recorded for diagnostic purposes.

analogTrim

This method is called by the *solve* method and performs one iteration in the trimming process. Each iteration involves updating the equations of motion, calculating new trim variable values, and pushing the new trim states back into the vehicle from the rules.

Aircraft Trim Methods

selectRules

The *AircraftTrim selectRules* method overloads the virtual *selectRules* method of the *Trim* class. This method adds logic for selecting aircraft trim rules based on the user's specifications. The aircraft trim rules selected are added to the rule list through the *addRule* method.

Trim Rule Methods

calculateTrimVariables

The *calculateTrimVariables* method updates the states, errors, and gains for each trim rule selected. The states are gathered from the current set of vehicle data. The errors are computed based on the deviation of vehicle data from target values. Gains are updated from the associated trim object.

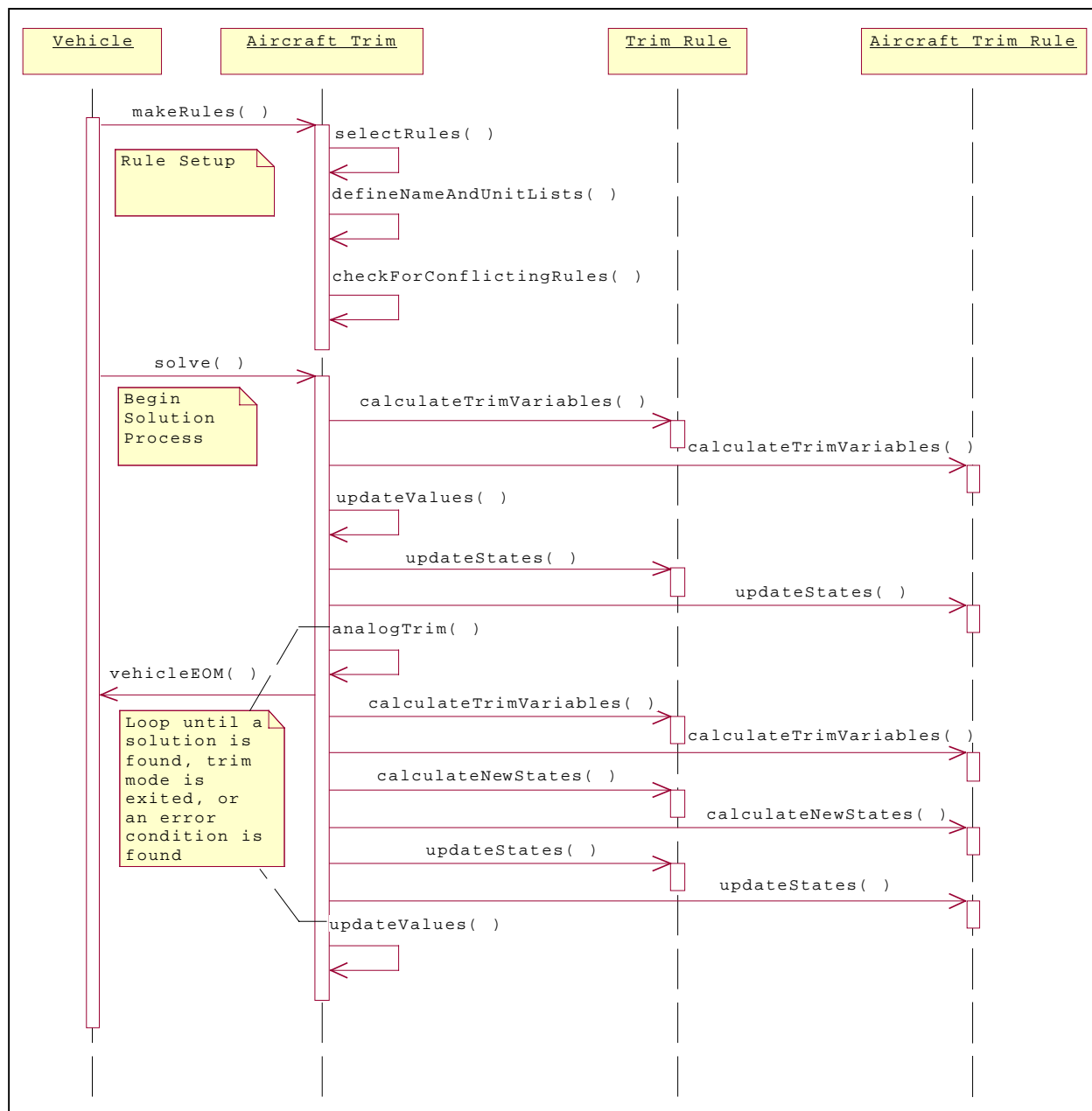


Figure 3 Trim UML Sequence Diagram

calculateNewStates

This method calculates new trim states. The following formula is normally used to calculate the new states based on the states, errors, and gains calculated in the previous iteration. Other methods of calculating the next set of trim states can be added by overloading this method.

$$state = state + error \cdot gain$$

Using this formula to calculate new trim states results in a very stable solution process, but may take longer to converge on a solution than some other trim methods.

updateStates

This method updates vehicle data that correspond to the trim states of the rule. Any constraints contained in the trim rule are recalculated and vehicle data is set accordingly.

Aircraft Trim Rule Methods

The AircraftTrimRule class does not add any additional methods to the TrimRule class. The only change is in the call to the constructor. When constructed, an aircraft trim rule object uses an aircraft object and an aircraft trim object argument, as opposed to a vehicle object and a trim object used by the TrimRule class.

calculateTrimVariables

The *calculateTrimVariables* method updates the trim states, errors, and gains for each aircraft trim rule selected. The states are gathered from current aircraft or vehicle data. The errors are computed based on the deviation of aircraft data from their corresponding target values. Gains are updated from the associated aircraft trim object.

updateStates

This method updates aircraft data based on trim states. Any constraints contained in each aircraft trim rule are recalculated and set in the aircraft.

Vehicle Methods

vehicleEOM

Several processes take place when the equations of motion of a vehicle are updated. These include processing cockpit inputs, recalculation of derived data, calculation of accelerations, and updating vehicle systems. The equations of motion are also updated each iteration while the simulation is running. During trim, a vehicle does not integrate any accelerations to determine the new set of vehicle data. The new vehicle data are determined by the trim rules.

Processing cockpit inputs updates the values of the cockpit inputs into the vehicle. In trim, however, cockpit inputs are typically overridden by internal trim variables.

Derived data are recalculated to make sure the vehicle has a consistent set of states. For example, making sure angle of attack (α), pitch (θ), and flight path angle (γ) are consistent for aircraft.

Accelerations are calculated from the vehicle data. The accelerations of interest to trim are typically body axis translational and angular accelerations.

The vehicle systems are updated so that any data in the systems are consistent with the current frame. Vehicle system updates can occur after the derived data calculations or after the acceleration calculations,

depending on the nature of the individual vehicle system. Some examples of vehicle systems include fuel systems, control systems, and propulsion systems.

Aircraft Methods

The differences between the Vehicle and Aircraft classes exist in methods called by the *VehicleEOM* method. The Aircraft class overloads the methods called by *VehicleEOM*, so that aircraft trim control variables can be properly mapped to the aircraft cockpit. For the four control variables used by aircraft trim (see AircraftTrim Class), each aircraft individually determines which trim control states override which cockpit inputs or other aircraft data. Aircraft also have additional derived data, which are calculated.

Filters and Integrators

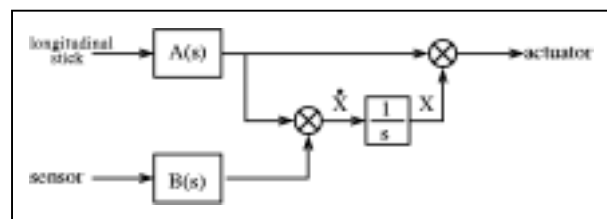


Figure 4 Pitch Loop Integrator

Filters and integrators require special attention when trimming vehicles. It is desirable, during trim, for filters to respond with their steady state outputs. Two different behaviors for integrators are used in trim. In the LaSRS++ framework, every filter and integrator knows the current mode of the simulation.

When the simulation is in trim mode, filters set their outputs to their steady state output values, $t = \infty$ ($s = 0$ in the Laplace domain). In this way, all internal derivatives of the filter are zero, such that, when the vehicle switches from trim to operate mode, there are no undesired transients.

Integrators can behave in two different manners during trim. If an integrator has a known initial value and derivative, the integrator can be held at the initial value and derivative. An example of this would be a mass integrator for burning fuel. The initial fuel amount is specified, and the initial burn rate can be directly computed from the throttle settings of the engines. If allowed to integrate during trim, the vehicle would burn fuel before it ever went into operate.

The second type of behavior is for integrators that must have derivatives equal to zero, or some other known value, but whose initial value is not known. In this

case, a trim rule must be added to monitor the integrators derivative until the target value is reached. An example of this is a pitch loop integrator is shown in Figure 4.

Earth Models

The LaSRS++ framework has three different earth models, flat, round, and ellipsoidal. The earth can also be rotated for the round and ellipsoidal models. Since non-flat earth models can be used, the assumption that trimming ailerons and elevator positions to zero will make the roll and pitch rates zero is invalid. Depending on the location of the vehicle relative to the world and the heading, small elevator, aileron, and rudder inputs may be needed to zero out pitch, roll, and yaw accelerations relative to the earth's surface, due to the curvature and rotation of the earth.

Results

This run was conducted at an altitude of 10,000 feet, target Mach number of 0.5, in a coordinated turn at a 10 degree bank angle, with a target ground track of 45.0 degrees. A rotating ellipsoidal earth model with inverse radius squared gravity was used. The aircraft trim rules used were Velocity Mach, Pitch Rate, Flight Path, Alpha Trim, Coordinated Turn, and Lateral Surfaces. The error tolerance was set to 1E-10 and took 900 iterations to find a solution. The resulting aircraft states are listed in Table 1. Plots of interesting trim states and errors vs. iteration can be found at the end of this paper.

Velocity Mach

The Velocity Mach rule adjusts Vtotal until the target Mach number (0.5) is reached. This rule makes sure the Mach number specified is correct during trim, even if another rule trims the altitude.

Alpha Trim

The Alpha Trim rule adjusts the angle of attack until the body-Z axis acceleration is zero.

Flight Path

The Flight Path rule is responsible for setting up the throttle and the pitch angle of the aircraft. The throttle is adjusted until the body-x axis acceleration is zero. The pitch angle is adjusted until the specified flight path angle (0.0) is reached.

Coordinated Turn

The Coordinated Turn rule trims the aircraft into a steady turn, v dot equal to zero. This is accomplished

by adjusting the sideslip until the body sideward acceleration, v dot, reaches zero. The yaw angle is trimmed until the desired ground track angle (45.0 degrees) is reached. The bank angle is set to the specified bank angle (10.0 degrees).

Lateral Surfaces

The Lateral Surfaces rule adjusts the *aileron_at_trim* and *rudder_at_trim* values to zero out the body roll and yaw accelerations respectively.

Pitch Rate

The Pitch Rate rule adjusts the *elevator_at_trim* until the body pitch acceleration is zero.

Table 1 Trim Results

h	Altitude	10,000 ft
M	Mach	0.5
Vt	Vtotal	538.70 ft/sec
IAS	Indicated Airspeed	276.85 knots
qbar	Dynamic Pressure	254.73 psf
α	Angle of Attack	5.0818 degrees
β	Sideslip	-0.25069 degrees
hdot	Altitude rate	7.47e-12 ft/sec
u	Body-X velocity	536.58 ft/sec
v	Body-Y velocity	-2.3570 ft/sec
w	Body-Z velocity	47.717 ft/sec
p	Body roll rate	-0.0518 deg/sec
q	Body pitch rate	0.1021 deg/sec
r	Body yaw rate	0.5875 deg/sec
ϕ	Roll	10.0 degrees
θ	Pitch	4.9617 degrees
ψ	Yaw	46.128 degrees
pdot	Body roll acceleration	6.70e-12 deg/sec ²
qdot	Body pitch acceleration	-6.65e-10 deg/sec ²
rdot	Body yaw acceleration	6.70e-11 deg/sec ²
udot	Body X acceleration	-1.39e-12 ft/sec ²
vdot	Body Y acceleration	-1.56e-11 ft/sec ²
wdot	Body Z acceleration	-2.45e-12 ft/sec ²
δ_a	<i>aileron_at_trim</i>	-0.4942 non-dim
δ_e	<i>elevator_at_trim</i>	-3.1240 non-dim
δ_r	<i>rudder_at_trim</i>	1.2146 non-dim
δ_t	<i>throttle_at_trim</i>	0.1426 non-dim

Conclusions

There are many advantages to using an object-oriented trim design. In an object-oriented framework, a set of generic trim rules can be developed that work on any type of simulation object. For aircraft, generic aircraft trim rules can be used. Any aircraft that derives from the Aircraft class, 757, F-16, etc., can all use the same

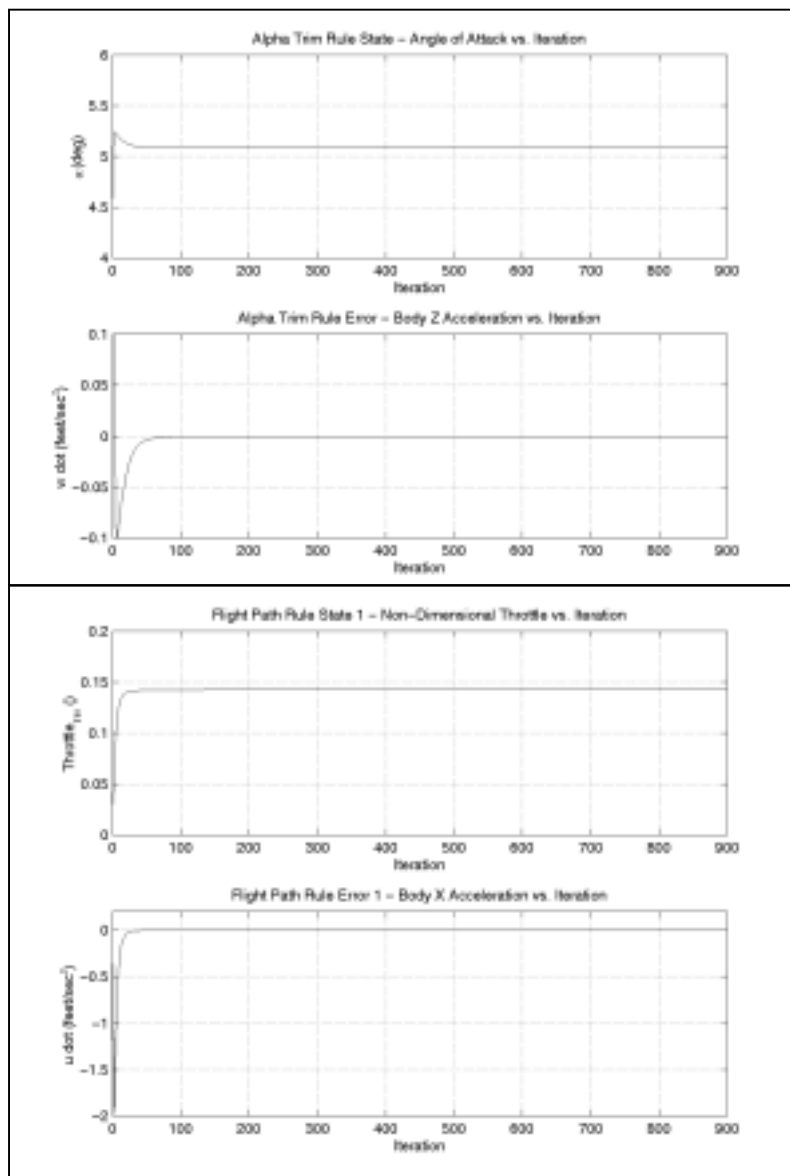
rules for trimming a given condition. Through the addition of a generic method (function) to check for rule conflicts, complex sections of code to check for specific rule conflicts are no longer needed and no longer need to be maintained when a new rule is added. The only vehicle specific code required is to map the generic control variables into the correct cockpit input on the vehicle.

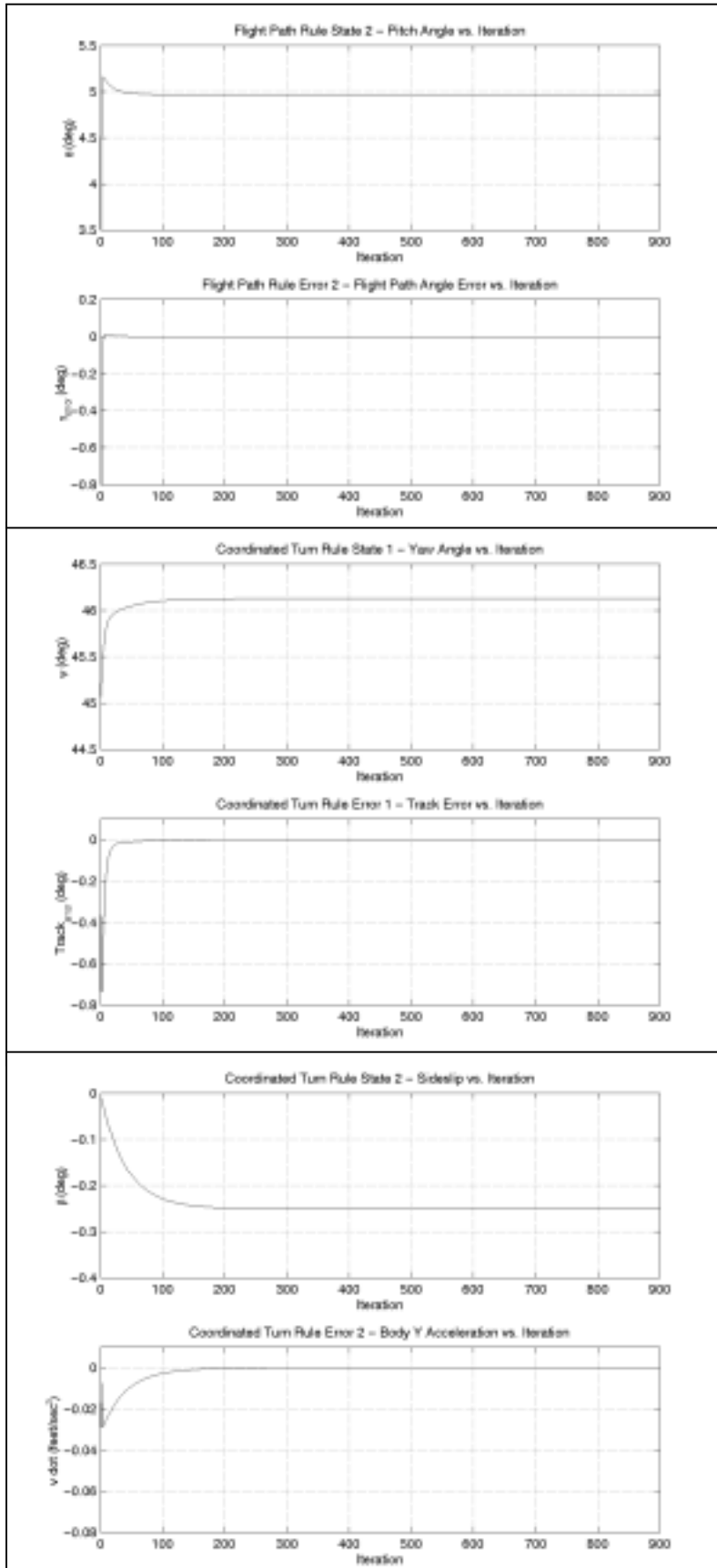
2 Muller, Pierre-Alain. *Instant UML*. Wrox Press Ltd. Chicago, Illinois, 1997.

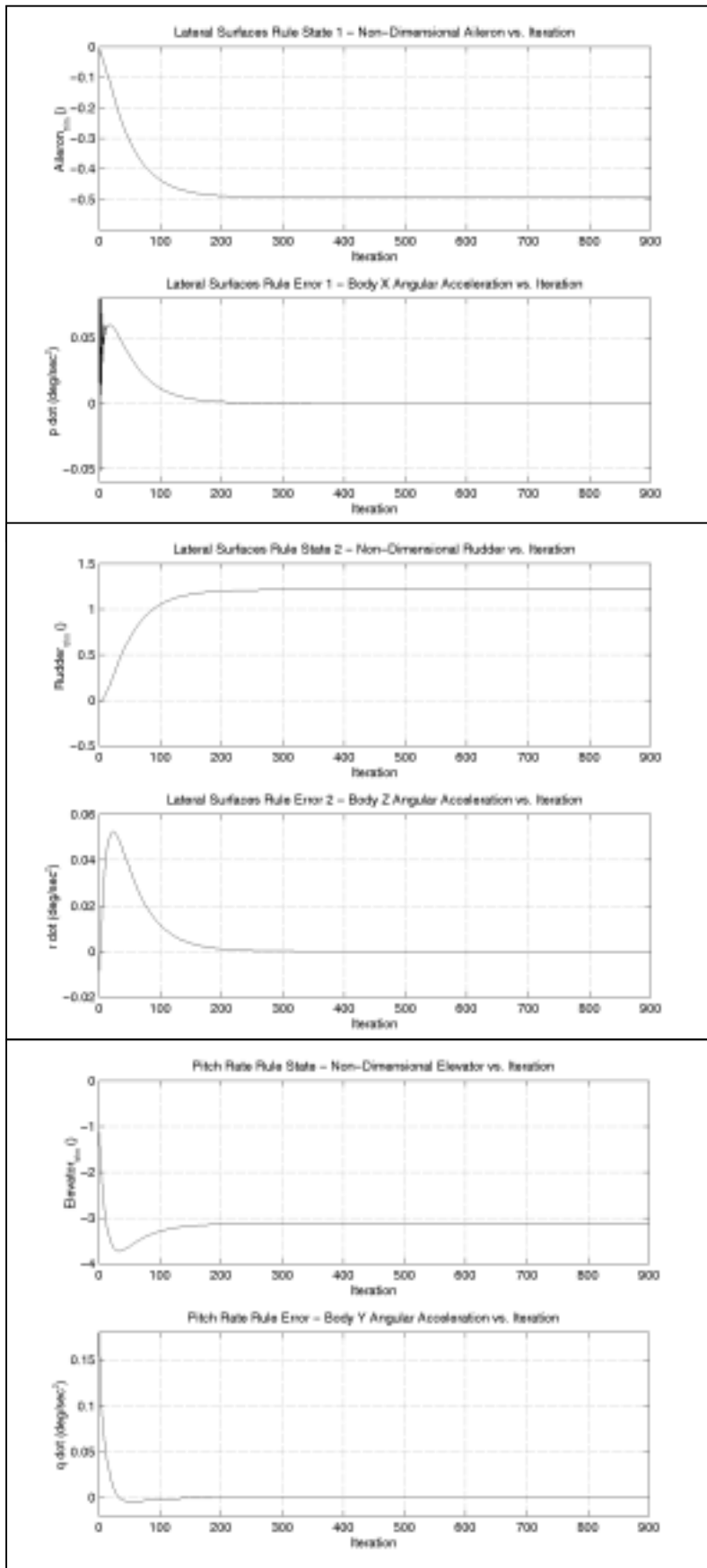
3 Leslie, R.; Geyer D.; Cunningham, K.; Madden, M.; Kenney, P.; Glaab, P. *LaSRS++: An Object-Oriented Framework for Real-Time Simulation of Aircraft*. AIAA 98-4529, Modeling and Simulation Technology Conference, Boston, MA, August 1998.

References

1 Booch, Grady. *Object-Oriented Analysis and Design With Applications*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.







Aircraft Trim Rules

All of the following rules derive from the AircraftTrimRule class. See Table 1 for symbol definitions.

Notation: $\theta, \delta t = \text{target}$ – indicates Pitch and *throttle_at_trim* are set to target values.

Velocity Rules	State	Error	Constraints
Velocity Drag	Vtotal	u dot	
Velocity Mach	Vtotal	mach	
Velocity Speed	Vtotal	speed	
Velocity Alpha	Vtotal	w dot	
Alpha Rules	State	Error	Constraints
Alpha Trim	α	w dot	
Constant Alpha			$\alpha = \text{target}$
Loaded Loop	α	Nz	$q = f(\text{w dot world relative}, \text{u air relative})$
Longitudinal Rules	State	Error	Constraints
Constant Pitch	δt	u dot	$\theta = \text{target}$
Constant Pitch/ Constant Thrust			$\theta, \delta t = \text{target}$
Constant Thrust	θ	Vtotal dot	$\delta t = \text{target}$
Constant Thrust, Pitch	θ	w dot	$\delta t = \text{target}$
Constant Thrust, Rate of Climb	θ	h dot	$\delta t = \text{target}$
Flight Path	δt	u dot	
	θ	γ	
Pitch Thrust	θ	w dot	
	δt	u dot	
Rate of Climb	δt	u dot	
	γ	h dot	
Lateral Rules	State	Error	Constraints
Bank into Wind	ψ	track	$\beta = f(u, \phi, V_{\text{total}})$
	ϕ	v dot	
Coordinated Turn	ψ	track	$\phi = \text{target}$
	β	v dot	$\phi, \theta, \psi \text{ dot} = f(g, \theta, \phi, u, w)$
Crab into Wind	ψ	track	
	β	v dot	
Loaded Roll	ψ	track	$\phi, \theta, \psi \text{ dot} = f(g, \theta, \phi, u, w)$
	β	v dot	
	ϕ	u dot	
No Lateral Motion			$\psi = \text{target}$
			$\delta a, \delta r, \phi, \phi \text{ dot}, \theta \text{ dot}, \psi \text{ dot} = 0$
One Engine Out	ϕ	track	
	β	v dot	
Steady Side Slip	ψ	track	$\beta = \text{target}$
	ϕ	v dot	
Other Rules	State	Error	Constraints
Ground Trim	h	vertical acceleration	$\delta t = \text{target}$
	θ	q dot	
	ϕ	p dot	
	ψ	track error	
Lateral Surfaces	δa	p dot	
	δr	r dot	
Pitch Rate	δe	q dot	