NASA Technical Memorandum 110164

Manual for a Workstation-based Generic Flight Simulation Program (LaRCsim) Version 1.4

E. Bruce Jackson
Langley Research Center
Hampton, Virginia

April 1995

Summary

LaRCsim is a set of ANSI C routines that implement a full set of equations of motion for a rigid-body aircraft in atmospheric and low-earth orbital flight, suitable for pilot-in-the-loop simulations on a workstation-class computer. All six rigid-body degrees of freedom are modeled. The modules provided include calculations of the typical aircraft rigid body simulation variables, earth geodesy, gravity and atmosphere models, and support several data recording options. Features/limitations of the current version include English units of measure, a 1962 atmosphere model in cubic spline function lookup form, ranging from sea level to 75,000 feet, rotating oblate spheroidal earth model, with aircraft C.G. coordinates in both geocentric and geodetic axes. Angular integrations are done using quaternion angular state variables. Vehicle X-Z symmetry is assumed.

A copy of this software is available upon request to the author.

Introduction

Historically, six degree of freedom aircraft simulations have been performed on larger minicomputers or mainframe computers due to limited processing speed and data storage capability on smaller workstation and desktop computers. With the advent of more powerful reduced instruction set computer (RISC) architecture, the processing capability of a desktop computer exceeds that of a supercomputer of a decade ago.

Simultaneously with the rise in popularity of workstation and desktop computers, the acceptance of UNIX-style operating systems has grown. This popular operating system has brought with it the C programming language in which the original UNIX kernal was written. While the standard C libraries lack some of the mathematical procedures of FORTRAN, in which most digital aircraft models are written, it is still possible to make use of this powerful and portable language. Abstract data types, longer variable names, data structures, and recursion allow the simulation architect to write maintainable and self-documenting software, with full access, through standardized library routines, to operating system capabilities in a nearly machine independent fashion.

Although not fully utilized in this version of LaRCsim, the popular X-Windows facility is easily manipulated in C. This provides for graphical operator/user interface capabilities on any X capable terminal or personal computer terminal emulator (called a *window server*).

This version of LaRCsim utilizes a `curses`-based terminal interface, which will support almost all types of computer terminals. X-windows support is planned for later versions of LaRCsim. Also supported is a Silicon Graphics GL workstation interface that includes out-the-window scenery and heads-up display symbology. The pilot controls are provided through a mouse or, optionally, an analog-to-digital interface (driver code for the analog-to-digital interface is not included since the software depends upon the choice of host processor and interface hardware.)

Output options include time history information in ASCII text tab-delimited, Dryden's GetData .ASC1, or Agile-Vu "`.flt`" format; a fourth option will write the time history data into a text file suitable for execution by one of several popular controls analysis software tools. Any global or static local variable can be recorded. The recording module uses debugger symbol to access static or global variables at a user-selected frequency. Specification of variables to be recorded can be made at run-time.

Overview

What is LaRCsim?

LaRCsim is a set of C routines that implement a full set of equations of motion for a rigid-body aircraft in atmospheric and low-earth orbital flight. It is intended

to be used with additional, user-provided subroutines (either FORTRAN or C) that describe the aerodynamics, propulsion system, and other flight dynamic elements of a specific air vehicle. Once combined with the vehicle-specific routines, LaRCsim provides a desktop- and/or cockpit-based near-real-time simulation of the vehicle for engineering analysis and control law development.

The six rigid-body degrees of freedom are modeled. The modules provided include all of the kinematic relationships, most of the conventional output variables, geodesy and atmospheric models, and a data recording option. Some features/limitations of the current version are as follows:

- English units of measure.
- 1962 atmosphere model in cubic spline function lookup form, ranging from sea level to 75,000 ft. Included in the model are density, speed of sound, and sigma.
- Rotating oblate spheroidal earth model, with aircraft C.G. coordinates in both geocentric and geodetic axes.
- Vehicle X-Z symmetry is assumed.
- Quaternions are used in determining the angular orientation (although equivalent Euler angles are also calculated) to avoid the singularity at ± 90 degrees pitch angle.
- Gravitational harmonic effects due to the earth's oblateness are modeled.
- Modular design allows user to incorporate modified atmosphere, turbulence, and steady winds into the simulation.
- Rotating machinery effects are not modeled.

Origin and Purpose

LaRCsim was developed as part of an engineering flight simulation facility at NASA Langley Research Center that is used to debug aircraft flight control laws. This facility, known as Advanced Controls Evaluation Simulator (ACES), is used in the Dynamics and Control Branch (DCB) and currently consists of a dual RISC processer Silicon Graphics Onyx computer with RealityEngine-2 graphics driving an evaluation cockpit with throttles and a side stick hand controller.

The LaRCsim routines are used to provide appropriate aircraft dynamic responses to flight control commands. The flight control laws may be written in C or Fortran. The equations of motion are based upon work by McFarland in reference 1. The axis frames and sign conventions comply with the ANSI/AIAA recommended practice as outlined in reference 2; geodesy calculations use the relationships outlined in reference 3, as well as a custom geocentric to geodetic conversion developed by the author. The atmosphere model is derived from data found in references 4 and 5; other physical constants were obtained from references 6 and 7. LaRCsim itself is based upon FORTRAN routines originally developed by the author for the U. S. Naval Air Test Center (now the Naval Air Warfare Center) under a project known as CASTLE (see reference 8); these routines have ties back to the NASA Ames FORTRAN simulation routines known as BASIC, written by McFarland and others.

It is intended that LaRCsim applications be capable of running both with a cockpit and pilot in the loop as well as in terminal interactive and batch modes. This version includes both a generic display terminal and Silicon Graphics GL-based keyboard/mouse interfaces in addition to an external cockpit interface.

Changes from version 1.3

The ACES facility is still being developed, and LaRCsim continues to evolve. This release, version 1.4, differs from version 1.3 as follows:

- Six-degree of freedom trim capability has been added.

- The default settings file has been renamed, and is automatically updated at the end of a session so LaRCsim "remembers" settings from the previous session.
- Initial conditions may be specified at by a flag on the command line.
- Time step and initialization flags are now passed to model routines.

Additional information on these changes is available in the README file, provided in the software distribution. Please see this file for more information on what is required to adapt a version 1.3 simulation model to version 1.4. This report details the requirements to implement a new version 1.4 simulation model.

Input files

Default settings file. LaRCsim is fairly self-contained, and does not require any special supporting files to run. It does, however, utilize one file if it is present in the default directory: if present, a file named *.simname* (also called the default settings file) specifies what parameters are to be recorded during the simulation run, what parameters are to be used to trim the vehicle and what parameters are to be set to zero by the trim algorithm. The settings file may specify a default initial condition to which the model is initialized if no other initial condition file is specified on the command line. This file is automatically updated at the end of a LaRCsim session to record any changes in these settings. A sample settings file is shown in figure 1.

In the present version of LaRCsim, the default settings file contains four sections of information: previous simulation operation settings, a list of parameters to record, the default trim parameters, and the default initial conditions. These sections are independent and may appear in any order.

The first few lines of the default settings file demonstrates the use of a pound sign (#) as the first non-blank character to denote a comment line; comments can appear on any line (as long as the first non-blank character is a #). Blank lines are ignored.

The third line in the file is the first line that is used by LaRCsim: "`sim`" appears on a line by itself to indicated the beginning of a list of simulation options that were in force at the end of the last session. This line is followed by "`0010`" on the next line by itself; this flag line indicates which version of syntax is used (presently version 1.0) so that future version of LaRCsim will be able to recognize and use older input files. The contents of this section indicate what type of files to record at the end of the simulation session; the spacing with which to write the data files, the end time of the simulation; and the update rates for the model, screen refresh, and data recording; and how long (in seconds) the data buffer should be. In the example given in figure 1, a data file in matrix format will be written when the simulation ends. It will contain up to one hour's worth of simulation data, recorded at 20 Hz and every frame will be written to the data set. The model itself will run up to one hour, at 120 Hz, and the video screen (or terminal screen) will be updated at 30 frames per second.

In the next section, "`record`" appears on a line by itself to indicate the beginning of a list of parameters to be recorded during the simulation session. The next six lines are parameter declarations; these six parameters, if successfully located in the debugger symbol tables, will be added to 19 predefined variables and recorded during the simulation session.

The first three declaration lines are examples of how to specify scalar parameters. Note that these declarations are **local** variables to each routine. LaRCsim, by way of compiler-provided symbol tables, can locate and track the value of any local or global variable, but the variables must be **static** variables, declared as such at the top of each function. If the variables are **automatic** (i.e., not static), then the variable is defined only as long as the program is executing that function; thus, LaRCsim is unable to track automatic variables. The third declaration, of variable `forward_mu` in function `navion_gear`, is actually an automatic variable (in

```
# .navion created at 950406 22:57:12 by bjax
#============================== sim

sim
0010
    write_av       0
    write_mat      1
    write_tab      0
    write_asc1     0
    write_spacing  1
    end_time       3600.000000
    model_hz       120.000000
    term_update_hz 30.00000
    data_rate      20.000000
    buffer_time    3600.0000
end
#============================== record

record
0010
    aero        elevator
    aero        aileron
    gear        forward_mu
    *   generic_.f_gear_v[0]
    *   generic_.f_gear_v[1]
    *   generic_.f_gear_v[2]
end
#============================== trim

trim
0010
  controls: 3
#    module    parameter   min_val   max_val   pert_size
    *   generic_.euler_angles_v[1] -7.853981E-01   7.853981E-01    1.000000E-02
    aero        long_trim       -1.000000E+00   1.000000E+00    1.000000E-02
    *   cockpit_.throttle_pct   0.000000E+00    1.000000E+00    1.000000E-02
  outputs: 3
#    module    parameter   trim_criteria
    *   generic_.omega_dot_body_v[1]    5.000000E-05
    *   generic_.v_dot_body_v[0]        5.000000E-04
    *   generic_.v_dot_body_v[2]        5.000000E-04
end
#============================== init

init
0010
  continuous_states: 22
#    module    parameter   value
    *   generic_.geodetic_position_v[0] 2.374953E-04
    *   generic_.geodetic_position_v[1] 7.714288E-07
    *   generic_.geodetic_position_v[2] 1.099708E+01
    *   generic_.v_local_v[0]   1.740701E+02
    *   generic_.v_local_v[1]   1.522121E+03
    *   generic_.v_local_v[2]   -3.972784E+00
    *   generic_.euler_angles_v[0]      -1.481027E-04
    *   generic_.euler_angles_v[1]      1.127979E-01
    *   generic_.euler_angles_v[2]      2.089291E-03
    *   generic_.omega_body_v[0]        5.395570E-06
    *   generic_.omega_body_v[1]        0.000000E+00
    *   generic_.omega_body_v[2]        -2.788522E-05
    *   generic_.earth_position_angle   0.000000E+00
    *   generic_.mass   8.547270E+01
    *   generic_.i_xx   1.048000E+03
    *   generic_.i_yy   3.000000E+03
    *   generic_.i_zz   3.530000E+03
    *   generic_.i_xz   0.000000E+00
    *   generic_.d_cg_rp_body_v[0]      0.000000E+00
    *   generic_.d_cg_rp_body_v[1]      0.000000E+00
    *   generic_.d_cg_rp_body_v[2]      0.000000E+00
    aero        long_trim       -1.365538E-03
  discrete_states: 0
#    module    parameter   value
end
```

Figure 1. A sample default settings file.

the example simulation), and thus LaRCsim will complain when it reads this input file and attempts to locate `forward_mu` for the first time.

A local static variable is specified by the name of the function or subroutine in which it exists (e.g. `aero` or `navion_gear`) and the name of the variable. Case is important. `Elevator` is not the same variable as `elevator`.

The next three lines are examples of **global** variables; these are variables that have been declared outside the scope of a function. They are identified to LaRCsim as global by use of the `*` in place of a function name.

These last three lines also demonstrate the capability of LaRCsim to parse and locate elements of complex data structures; here, the elements of the landing gear force vector, `f_gear_v`, itself a part of the global data structure `generic_`, will be added to the list of variables to record. The syntax for non-scalar data elements follows that of ANSI C. Arrays are all zero-index-based, as in C (unlike FORTRAN).

The `end` word must appear on a line by itself to delimit the list of recording variables that began with `record`.

The next section of the default settings file tells LaRCsim how to attempt to trim the vehicle when requested. The format is similar to that used by the `record` section, with the addition of a count of how many controls and how many output variables are specified (on the `controls: 3` and `outputs: 3` lines). Note: in this version of LaRCsim, the number of controls *must* equal the number of outputs. LaRCsim presently supports trim strategies with up to ten controls and outputs; in practice, however, no more than six are required for a rigid fixed-wing aircraft. See the section below for a description of the trim method and suggested techniques.

Each trim `control` specification includes a module and parameter name, as before for `record` specifications, as well as minimum and maximum values and perturbation size (see the Trimming Strategies section below for more information about these values).

Each trim `output` specification includes a module and parameter name and a criteria value that specifies how close to zero the output must driven by the trim algorithm before a successful trim is achieved.

The next section of the settings file, the `init` section, specifies what parameters are considered states, and should include both continuous states and discrete states (flags, Booleans, and integers), as well as a specification for the default values of these states. The initial condition described in this settings file do not have to describe a trimmed flight condition. Each line of the `init` section includes a module and parameter name, as before, as well as the initial value for that state.

Overriding the default settings. The user may specify on the command line, with the `-i` option flag, a different settings file with an alternate initial condition (IC) description. An IC settings file should have a file name that describes the initial condition, and end with a `.ic` file type, such as `on_ground.ic`, `two_mile_final.ic`, etc. The contents of this file are identical in format to the `init` section of the default settings file; LaRCsim will substitute the optional initial conditions for those found in the default settings file.

As an example, the command line

```
navion -i on_ground.ic
```

will cause the navion simulation to start at a specified initial condition defined in an IC settings file named `on_ground.ic`.

Similarly, the default trim strategy may be replaced with a new one by identifying a file containing the new `trim` portion of the settings file using the `-i` flag. By convention, the trim settings file should end in `.trim` and contain only a `trim` specifications section.

Additional parameters may be added to the list of recorded parameters by specifying (again with the `-i` flag) a file that contains a `record` specification. Any parameters thus specified will be added to the existing list of recorded parameters.

In the present version of LaRCsim, only one settings file may be specified at run time; it is possible to combine several settings file into a single file, and specify that file name at run time to achieve the desired set of trim parameters, recorded variables, and initial conditions.

Optional search path and redirection. At startup, LaRCsim will search the directories listed in an environment variable `LARCSIMPATH`, if it is defined, to find both the default settings file (e.g. `.navion`) and any specified settings file files (e.g. `on_ground.ic`). LaRCsim will use the first occurance of these files discovered in the path of directories specified by `LARCSIMPATH`. The variable `LARCSIMPATH` should be a colon-separated list of directories, similar to standard UNIX `PATH` environment variables. If `LARCSIM` is undefined, only the default directory will be searched to find the settings file.

A settings file may contain a line beginning with '`@`'; this indicates to LaRCsim an additional file that should be parsed. For example, the default settings file for the terminal version of a simulation (e.g. `.navion_term`) could contain the single line, `@.navion`; LaRCsim would interpret this to mean the contents of `.navion` should be parsed instead of `.navion_term`. (Note: `.navion_term` should be set to read-only to prevent it from being overwritten at the end of the LaRCsim session.)

The file pointed to by the indirection flag '`@`' could itself contain an additional indirection flag; caution should be used to avoid circular references.

Output files

.simname This default settings file, if it does not already exist, is created at the end of each simulation session and will contain the default values for record parameters, trim controls, and initial conditions. If the default settings file already exists and is not write-protected it will be replaced with a new copy.

`run.flt` This file, if requested with the `-a` flag, will be generated at the end of a session and will contain a time history of each recorded parameter in Agile-Vu format.

`run.m` This file, if requested with the `-r` flag, will be generated at the end of a session and will contain the time history information in matrix notation, suitable for use as a script in one of the popular control system design and analysis products.

`run.asc1` This file, if requested by use of the `-x` command line switch, wil be generated at the end of a session and will contain the time history information in a format understood by the Dryden Flight Research Center's `GetData` and `XPlot` tools.

`run.dat` This file, if requested with the `-t` command line switch, will contain ASCII tab-delimited columns of the recorded data; the first line contains the names of the parameters included. This format may be useful for importing time history data into spreadsheet or other charting programs.

Running a LaRCsim Example

Compiling LaRCsim

Building LaRCsim from the distribution is straightforward:

1. Define an environment variable, `LARCSIM`, to point to the source directory for the main LaRCsim routines. This should probably be done in the user's `.login` file (Example: `setenv LARCSIM /aces/larcsim/v014`)
2. Change the default directory to `$LARCSIM`.
3. Enter the command "`make`." This will:
   a. create a new object library file, `libls.a`
   b. compile all of the LaRCsim source files

c. put all the generated object files in the `libls.a` archive library

The object archive library `libls.a` only needs to be rebuilt after a LaRCsim module has been modified.

Compiling and building the example simulation

Once the `libls.a` file has been built in the `$LARCSIM` directory, move to the directory containing the aircraft files (in the case of the example simulation, move to the `navion` directory).

1. Enter the command "`make`" (for Silicon Graphics-based simulations) or "`make term`" for a terminal-based simulation. This will compile all the `navion` source files and link them together to form the executable simulation program `navion` (for Silicon Graphics-based simulations, or `navion_term`, for a terminal-based simulation).
2. If desired, create a default settings file in the format described above. It should be named *.simname*, where *simname* is the name of the executable simulation program.

Running the example simulation program

Typing `navion` on the IRIX command line will run the navion example simulation program on the GL console; the `navion_term` command will run the navion example simulation on most terminals.

Command line switches. The command for running a LaRCsim model may include a number of optional flags or switches:

`-A` Run in conjunction with ACES cockpit (valid only for DCB users).

`-k` Run on the Silicon Graphics console using the mouse as a joystick (`-k` and the `-A` flags are mutually exclusive).

`-i` *filename*`.ic` Identifies an optional settings file that contains an alternate initial condition, trim strategy, or additional parameters to be recorded.

`-f <`*iteration rate*`>` Specifies an iteration rate, in iterations per second, that the simulation model is to execute. Default frame rate is 120 iterations per second.

`-o <`*output rate*`>` Specifies the rate at which the terminal or GL display screen should be updated, in frames per second. This rate must be an integral sub-multiple of the *iteration rate* (see `-f` above). For example, if the simulation model *iteration rate* is 120 iterations per second, legitimate choices for *output rate* are 120, 60, 40, 30, etc. frames per second (corresponding to $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}$, etc. of the *iteration rate*). Default screen refresh rate is 20 frames per second.

`-e <`*end time*`>` Specifies an end time for the simulation run. The simulation will terminate when this value of simulated time is reached, if the simulation is not reset prior to that time.

`-b <`*buffer length*`>` Specifies the length of the data storage buffer, in seconds. This circular buffer retains the last *buffer length* seconds of time history data. If not specified, the default *buffer length* equals the simulation *end time* given by `-e` above.

`-s <`*storage rate*`>` Specifies the rate, in records per second, at which the requested parameters will be recorded to the circular data buffer. This rate must be an integral sub-multiple of the *iteration rate* (see `-f` above). For example, if the simulation model *iteration rate* is 120 iterations per second, legitimate choices for *storage rate* are 120, 60, 40, 30, etc. records per second (corresponding to $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}$, etc. of the *iteration rate*). If not specified, the default *storage rate* will be one-eighth of the *iteration rate* of the simulation model.

`-a` <*filename*> Specifies that an Agile-Vu compatible ".flt" file is to be written at the end of the session. Default filename is `run.flt`. If this option is the last one on the command line, a filename must be specified.

`-t` <*filename*> Specifies that a tab-delimited ASCII listing of time history data be written at the session. Default filename is `run.dat`. If this option is the last one on the command line, a filename must be specified.

`-x` <*filename*> Specifies that a GetData/X-Plot compatible ".asc1" file is to be written at the end of the session. Default filename is `run.asc1`. If this option is the last one on the command line, a filename must be specified.

`-r` <*filename*> Specifies that a matrix manipulation software compatible `.m` file is to be written at the end of the session. Default filename is `run.m`. If this option is the last one on the command line, a filename must be specified.

`-d` Specifies that the run allow interactive debugging; this prevents scheduling of timer interrupts and forces the GL display into single-buffer mode. This switch is probably not of great use to the typical user.

GL console operation. The command `navion -k` will bring up the out-the-window view, on the SGI console, with a heads-up display (HUD) overlay, and allow the user to maneuver the aircraft using the mouse and keyboard. The mouse movement simulates a control stick: push forward to move the stick forward, left to roll left, etc.

When the simulation first comes up, the aircraft is placed in the specified initial condition and the display will indicate the simulation is paused (on a GL display, this is indicated by the HUD symbology showing up in a red color). At this point the simulation may be trimmed (using the 't' key) or put into operation (with the 'p' key). A trim may be requested at any time during a run by use of the 't' key; this allows the vehicle to be flown to an interesting point of the sky and retrimmed. A successful trim will cause the current flight conditions to be remembered as the new initial condition.

At any point, the 'r' key will reset the simulation to the last remembered initial condition, allowing repeated landing attempts, for example.

The simulation may be paused at any point by use of the 'p' key to toggle between pause and run modes. Data is recorded in run mode and during trim attempts.

The simulation session will last for up to 60 minutes; a longer period of time may be specified on the command line as a parameter for the `-e` option (see the previous section for information on various command line options).

Pressing the escape key causes the simulation to terminate, and any recorded data will be written to the requested output files.

Display terminal operation. The command `navion_term` will operate the same simulation, but does not use a mouse or provide GL graphics. Instead, a simple instrument panel is presented on the user's terminal screen and several keyboard keys are pressed into service for flight controls. Figure 2 shows the screen used in LaRCsim version 1.4, with flight control keys indicated. No rudder command is available in this version.

External cockpit operation. The command `navion -A` will operate the same simulation, but LaRCsim will call the external cockpit interface routine to provide control stick, rudder pedal, and throttle positions, as well as pause and reset buttons. Most keyboard commands will still operate.

Note for DCB users: in the ACES cockpit, the upper red button on the handgrip resets the simulation, and the thumb button pauses the simulation.

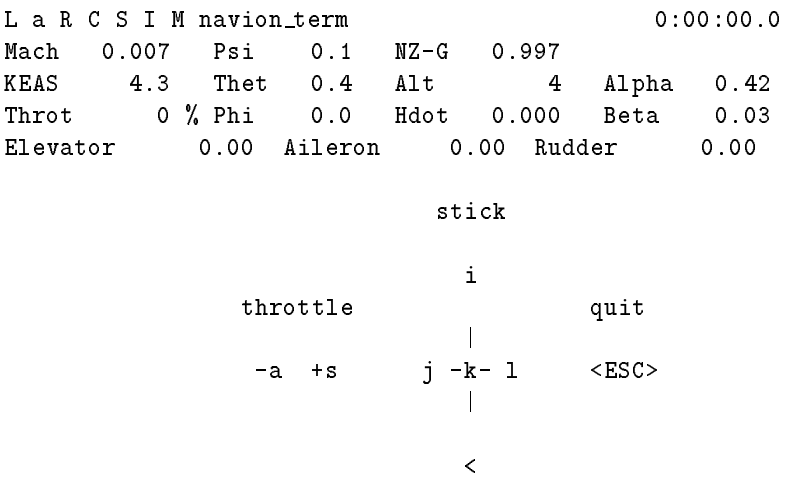

Figure 2. Terminal mode display

Trimming strategies

The trim algorithm, new to this version of LaRCsim, uses up to ten user-specified "controls" to drive a like number of "outputs" to values near zero. LaRCsim also forces pitch rate to zero prior to each trim attempt, so trimmed turns are not currently possible. Steady-heading sideslip trims, however, are possible and have been demonstrated. On-ground longitudinal trims are also supported.

The current mechanism to specify (and modify) the trim method requires editing the default settings file, or specifying a settings file containing a different set of trim controls and outputs by use of the `-i` flag on the command line. Listed below are examples of trim specifications that have been tested and used successfully in LaRCsim simulations at Langley Research Center.

In-flight longitudinal trim. In this example, pitch attitude, throttle, and a local variable in the aerodynamics module called "long_trim" are used to zero out the accelerations in pitch and body-X and -Z axes:

```
trim
0010
controls:  3
# module   parameter  min_val  max_val  pert_size
  *  generic_.euler_angles_v[1]  -0.785  0.785  1.0E-02
  aero  long_trim  -1.0000E+00  1.0000E+00  1.0000E-02
  *  cockpit_.throttle_pct  0.0000E+00  1.0000E+00  1.0000E-02
outputs:  3
#  module  parameter  trim_criteria
  *  generic_.omega_dot_body_v[1]  5.0000E-05
  *  generic_.v_dot_body_v[0]  5.0000E-04
  *  generic_.v_dot_body_v[2]  5.0000E-04
end
```

On-ground trim. With this strategy, two controls (pitch attitude and altitude) are used to obtain zero pitch and vertical acceleration, regardless of the aircraft's velocity or heading:

```
trim
0010
controls:  2
#  module  parameter  min_val  max_val  pert_size
  *  generic_.euler_angles_v[1]  -0.785  0.785  1.0E-02
```

```
  *  generic_.geodetic_position_v[2]  0 30 0.0001
outputs:  2
#  module  parameter  trim_criteria
  *  generic_.omega_dot_body_v[1]  5.0000E-05
  *  generic_.v_dot_local_v[2]  5.0000E-04
end
```

Steady-heading sideslip trim. In this strategy, three pilot control trim variables are used, along with throttle, pitch attitude, and heading angle to achieve zero accelerations in angular and local velocities:

```
# this trim is for steady-heading sideslip, where
# sideslip is given by local velocities.
trim
0010
controls:  6
#  module  parameter  min_val  max_val  pert_size
  subsystems longtrim  -3.0000E+01  3.0000E+01  3.0000E-02
  *  generic_.euler_angles_v[1]  -0.5  0.5  1.0000E-03
  *  cockpit_.throttle_pct  0.0000E+00  1.0000E+00  1.0000E-03
  subsystems lattrim  -10 10 0.01
  subsystems pedtrim  -10 10 0.01
  *  generic_.euler_angles_v[0]  -0.5  0.5 0.001
outputs:  6
#  module  parameter  trim_criteria
  *  generic_.omega_dot_body_v[0]  5.0000E-05
  *  generic_.omega_dot_body_v[1]  5.0000E-05
  *  generic_.omega_dot_body_v[2]  5.0000E-05
  *  generic_.v_dot_local_v[0]  5.0000E-04
  *  generic_.v_dot_local_v[1]  5.0000E-04
  *  generic_.v_dot_local_v[2]  5.0000E-04
end
```

Creating a New Aircraft Simulation

Mandatory routines

A new simulation model must provide, as a minimum, an aerodynamics routine with an entry point labeled `aero()`. The source code is usually kept in a file named after the specific vehicle, e.g. `navion_aero.c`. In addition, a complete vehicle model would include `engine()`, `subsystems()`, `inertias()`, and `gear()` routines, although stub routines are provided for these.

Inputs to these routines come from the `GENERIC` global variable structure, for which useful aliases are provided in the `ls_generic.h` header file (see Appendix A). The more sophisticated models will undoubtedly create an aircraft-specific set of global variables; the use of a `struct` or `COMMON` is recommended to share these global specific variables between simulation components. Interface to the simple keyboard, mouse and/or ACES cockpits is available through the `COCKPIT` data structure.

The expected outputs from `aero()` are simply the aerodynamic forces and moments about the reference point, in lbs and ft-lbs, respectively, being stored in the `F_aero_v` and `M_aero_v` vectors (scalar names `F_X_aero`, `F_Y_aero`, `F_Z_aero`, `M_l_aero`, `M_m_aero`, and `M_n_aero`).

Likewise, the outputs from any `engine()` or `gear()` routines should be stored in the `F_engine_v`, `M_engine_v`, `F_gear_v`, and `M_engine_v` vectors as appropriate. Refer to the example simulation for samples of how to do this.

If desired, the LaRCsim user may craft an `inertias()` routine to keep track of fuel burn (using an aircraft specific fuel flow parameter provided from `engine()`)

and adjust the inertia properties and center of gravity location values kept in `GENERIC`: `Mass`, `I_xx`, `I_yy`, `I_zz`, `I_xz`, and vector quantity `D_cg_rp_body_v` (the location of the center of gravity, measured from the reference point, in body axis); for most simulation studies of an engineering nature, the fuel quantity is a constant that can be, along with mass properties and C.G. location, be set at initialization (through user routine `model_init()`, or through a settings file.).

The user *must* have a `model_init()` routine, which is called before each simulation run, to set certain parameters. See the section below for a list of necessary parameters. Failure to set certain parameters will lead to an immediate divide by zero error, or unreasonable dynamic response of the simulation.

The `subsystems()` hook allows control system models, navigation system models, sensor models, autopilots, etc. to be included in the more elaborate simulations. These routines will likely use some of the parameters provided in `GENERIC` and get other inputs from and store outputs to user-defined common memory structure(s).

Mandatory parameters

The following is a list of the variables for which the user-supplied vehicle routines *must* provide reasonable values:

`Mass`	vehicle inertial properties;
`I_xx`	these must be non-zero
`I_yy`	
`I_zz`	
`I_xz`	
`D_pilot_rp_body_v`	pilot location w.r.t. reference point
`D_cg_rp_body_v`	C. of Grav. location w.r.t. reference point
`F_aero_v`	aero forces, body axes
`F_engine_v`	engine forces, body axes
`F_gear_v`	gear forces, body axes
`M_aero_v`	aero moments, body axes, about ref. pt.
`M_engine_v`	engine moments, body axes, about ref. pt.
`M_gear_v`	gear moments, body axes, about ref. pt.
`Runway_altitude`	location of threshold of runway of interest
`Runway_latitude`	
`Runway_longitude`	
`Runway_heading`	

These values may be initialized once, in the `model_init()` function, or may be calculated each frame, in a procedure called by `ls_model()`. The mass properties must by non-zero to avoid mathematical errors.

The following variables should be specified in `model_init()` to the appropriate initial conditions; they are thereafter calculated by the EOM routines:

`Geodetic_position_v`	geodetic position in radiansfeet
`Euler_angles_v`	aircraft attitude (ϕ, θ, ψ), radians
`V_local_v`	center of gravity velocities, in ft/s
`Omega_body_v`	body axis rates, in rad/s

where geodetic position is latitude, longitude, and altitude above sea level. The following variables may be set by the user routines if desired:

`V_local_airmass_v`	airmass velocity: steady wind
`V_local_gust_v`	body axis turbulence

Support for FORTRAN routines

Existing FORTRAN routines can be interfaced to LaRCsim through use of "wrapper" routines that translate between existing FORTRAN `COMMON` data structures and the `GENERIC` and other LaRCsim data structures. It is possible to write FORTRAN versions of `aero()`, `engine()`, `inertias()`, etc., but the reader is encouraged to write new models in C (or even C++) for maintainability and compatibility reasons.

The secret to writing these "wrapper" routines is to realize that, at least in IRIX, FORTRAN entry points and commons appear (from the C side) as having the same name that they do in FORTRAN, but in lowercase and with an underscore ('_') appended, and vice-versa. Thus, a FORTRAN `COMMON` structure named `SIMPAR` will appear to the C language routine as a global variable named `simpar_` (it must be declared as an external global structure in the C routine or header file). Likewise, a FORTRAN subroutine declared as `SUBROUTINE PLSURF` can be called from a C program as `plsurf_()`. Consult the documentation for each particular operating system for more information on how to develop a "wrapper" for an implementation on that system.

When the real-time loop is entered, the routines specified in `ls_model()` are called once per loop. The user is expected to replace the simple `aero()` and `engine()` routines provided in this package with more realistic aerodynamic and propulsion system models. These models should calculate, based upon the current Mach, altitude, angle of attack, etc. the appropriate forces and moments due to aerodynamics, engines, and perhaps landing gear, if appropriate. These forces and moments are to be provided in units of lbs and ft-lbs, in the X-Y-Z body axis system (positive indicates forward, right, and down, respectively) acting at the predefined reference position. If fuel consumption or weapon drops are to be simulated, an `inertias()` routine should be added, and the values of `Mass`, `I_xx`, `I_yy`, `I_zz` and `I_xz` should be updated in each loop. Center of gravity movement should be reflected in updates to the `D_cg_rp_body_v` vector as well. It is also possible to change runway location during simulation operation, if appropriate; the code to provide this capability is not included in the present LaRCsim version, however.

Function Data Interpolation

Overview. Mathematical descriptions of the aerodynamics of most flight vehicles usually include non-linear elements, such as the stall "break" characteristic exhibited by straight fixed-wing aircraft at higher angles of attack. Other aerodynamic properties exhibit even more pronounced non-linearities with respect to angles of attack, sideslip, Mach number, control surface deflection and other "independent" flight conditions. Other components of a flight vehicle model, such as propulsion systems and control law gain tables, often need to represent a very non-linear parameter in some fashion.

Many ways have been developed in previous years to represent these non-linear functions, including specialized mechanical analogues and electrical circuits. In present flight simulators these functions are represented through special-purpose software. To save memory, early software-based functions were generated using polynomials to approximate the non-linear characteristics of the actual airplane. As memory became less expensive, small tables of numbers were stored and then interpolated at run time. The present industry practice is to use large amounts of memory to store multi-dimensional tables; a return to polynomial representation may be underway to generate models that are mathematically smooth (see reference 10). The atmosphere model developed for LaRCsim uses a combination of these techniques; it represents atmospheric properties by use of a table, based upon altitude, of the coefficients of a set of cubic spline functions that

provide smoothly varying curves that agree with the original atmosphere model at the "knots".

To provide a general, C-based function generation capability, the `ls_funcgen.c` module was developed. This simple code makes use of an object paradigm to represent the function tables and a recursive C-routine to perform the interpolation along each dimension. This particular solution is, in the opinion of the author, elegant in its object-oriented design, recursiveness and the capability to handle function sets of unlimited size and dimension; it is, on the other hand, a little difficult to understand, and not as fast as an in-line, non-recursive, FORTRAN routine used for comparision.

To become really useful, a set of tools to generate the function data code for a particular simulation would be nice and may become available in a later version of LaRCsim.

Terminology. The following terms are used to describe the function generation routine:

Breakpoint data set A monotonically increasing vector of real numbers that represent the values of an independent variable for which the dependent function is known and tabulated.

Dependent variable The value of the function, or the return value from the function generation subroutine. Known values of the dependent variable for specific values of the independent variable(s) upon which it depends are provided by the user in the form of data tables; the routines described in this section provide linearly interpolated values of the dependent variable for an arbitrary set of independent variable values.

Dimension Each dimension of a data table represents an independent variable upon which the dependent variable, represented as points in the function table, are based.

Function table A multi-dimensional table of dependent variable values that correspond to a given number of breakpoint data sets. In LaRCsim, the first dimension varies most rapidly.

Independent variable An argument to the function. In terms of aerodynamic tables, the independent variables are usually one or more of the following: angle of attack, angle of sideslip, Mach number, and control surface deflection.

Index and weights value A floating point number, corresponding to a specific breakpoint set, that represents the present location of the independent variable in that breakpoint set. The integer before the decimal represents the index (0-origin based) of the breakpoint data point that is closest to, but less than, the actual independent variable value; the fractional portion of the number represents the fractional distance the independent variable is between the indexed and next-higher breakpoint value. It is defined as w, where

$$w = i + d$$

where d is the interpolation ratio given below and i is the current index of the next-lower value of the breakpoint set.

Interpolation ratio This fractional quantity, d, represents the location of the independent variable between the next lower and next-

higher values of the breakpoint set. It is defined as:

$$d = \frac{x - x_i}{x_{i+1} - x_i}$$

where x is the value of the independent variable, x_{i+1} is the next-higher value of the breakpoint set, and x_i is the next-lower value of the breakpoint set.

Normalization The process of determining the proper index and weights value w (see above) for the present independent variable value.

Implementation. If one were to describe the problem of data interpolation, one might use the following description:

> The value of a function is represented in an orthagonal N-dimensional table. Each dimension of the table corresponds to a monotonicly increasing independent breakpoint variable. The data in the table is arranged such that each entry represents the known value of the function, or dependent variable, corresponding to fixed value(s) of the breakpoint, or independent variable(s), at that index of the table. The problem is to determine the value of the dependent variable at any arbitrary value(s) of the independent variable(s). This is done by interpolating the known value of the function between the two surrounding table entries; in effect, generating a new table entry. If multidimensional, this process may be repeated for each dimension of the table, but the "known" values used for each succeeding interpolation are actually interpolated values from the previous dimension. This recursion continues until the value of the dependent variable has been interpolated for the last dimension; this quantity is the value of the function corresponding to the arbitrary values of the independent variables.

In the most general case, some breakpoint sets may be shared between function tables; and since breakpoint normalization is relatively CPU intensive, re-use of normalized breakpoints is a good idea. Similarly, often times the function table itself may be duplicted to represent similar but independent functions; a common example is a set of spoilers on an aircraft that are operated independently, where the spoilers have similar or identical aerodynamic effect (except for perhaps a minus sign) but may well be operated at different deflections.

The function generator data structures used in LaRCsim allow for re-use of breakpoints and function table data; for this reason, understanding the data structures may take a little examination and thought. Separate "objects" that represent the breakpoint sets, the function values themselves, the actual function data (which associates the function data with the corresponding breakpoint sets) and the final object, the non-linear function (which associates function data with breakpoint normalization data) are all stored as separate data structures, as described below.

In keeping with the object-oriented abstraction of the problem, breakpoint data sets and function tables are stored separately in `BREAKPOINTS` and `DATA` structures. They are associated together in an individual `FUNC_DATA` structure; the `FUNC_DATA` structure is an abstraction of a multi-dimensional curve or surface. These data structures are defined in the header file `ls_funcgen.h`.

The `NONLINEAR_FUNCTION` structure associates this function data with the interpolation information (index and weights as well as the last value returned on the previous lookup call). This structure is an abstraction of the process of interpolating a `FUNC_DATA` curve; it includes a pointer to the function data as well as state information about where the function was most recently found, which speeds

up subsequent searches since a sequential search through the breakpoint vector, starting with the last index used, is used instead of a binary search. The crawl search is believed to be better for flight simulation function generation applications than a binary search, since the traditional independent arguments change fairly slowly.

The tables are effectively unlimited in size and number of dimensions; the maximum length in any dimension is set by `MAX_LENGTH`, and the number of dimensions is set by `MAX_DIMENSION`; both are declared in the `ls_funcgen.h` header file.

Another data structure, `ARG_LIST`, is used to pass interpolation information to the lookup function. It contains the current index value and interpolation ratio for each dimension of the nonlinear function.

For an example implementation of these data objects and an actual implementation of this code, refer to the header information found in `ls_funcgen.c`.

Implementation Details

File Descriptions

The source and header files that make up the LaRCsim application are listed below, along with individual file version numbers:

In the LARCSIM directory:

```
Makefile, v 1.0                    ls_funcgen.c, v 1.6
ls_ACES.h, v 1.4                   ls_geodesy.c, v 1.5
ls_cockpit.h, v 1.3                ls_gravity.c, v 1.2
ls_constants.h, v 1.0              ls_ifgl.c, v 1.15
ls_err.h, v 1.1                    ls_ifterm.c, v 1.1
ls_funcgen.h, v 1.1                ls_init.c, v 1.4
ls_generic.h, v 1.0                ls_matrix.c, v 1.1
ls_matrix.h, v 1.1                 ls_model.c, v 1.3
ls_sim_control.h, v 1.11           ls_record.c, v 1.11
ls_sym.h, v 1.9                    ls_settings.c, v 1.6
ls_tape.h, v 1.6                   ls_step.c, v 1.5
ls_types.h, v 1.0                  ls_sym.c, v 2.7
LaRCsim.c, v 1.4.1.7               ls_sync.c, v 1.7
atmos_62.c, v 1.0                  ls_trim.c, v 1.9
default_model_routines.c, v 1.3    ls_writeasc1.c, v 1.7
ls_ACES.c, v 1.8                   ls_writeav.c, v 1.10
ls_accel.c, v 1.5                  ls_writemat.c, v 1.11
ls_aux.c, v 1.12                   ls_writetab.c, v 1.4
ls_err.c, v 1.2
```

In the example directory:

```
Makefile, v 1.0                    navion_engine.c, v 1.1
navion.h, v 1.3                    navion_gear.c, v 1.0
.navion, v 1.0                     navion_init.c, v 1.0
navion_aero.c, v 1.0
```

Each of these components of the LaRCsim simulation program are described below.

Compilation support files.

`Makefile` A simple makefile that allows the LaRCsim object library `libls.a` to be created and/or updated on most Unix platforms by the simple command `make`. To build

the example simulation, issue the `make` command in the LaRCsim directory, and then move to the navion subdirectory and issue another `make` command.

Header files.

`ls_ACES.h` This header file describes various constants and data structures used with the Dynamics and Control Branch Advanced Controls Evaluation Simulator (ACES) hardware; it is not of interest to a non-DCB user.

`ls_types.h` This file defines the two principal data types used in LaRCsim: `SCALAR` and `VECTOR_3`. The `SCALAR` data type, which is defined as a `double`, is suggested for use by any C or C++ routines added to LaRCsim. This definition allows easy modification of the level of precision of calculations, since changing the type definition of `SCALAR` in this routine to, say, `float`, would halve the precision of all LaRCsim module calculations.

Prior to version 1.3, the scalar floating-point type `DATA` was defined, but is not recommended for further use to avoid confusion with the FORTRAN compiler directive of the same name. It remains defined in this module for commonality with older routines, but may be removed in future versions.

A 3-element vector of `SCALAR` elements, `VECTOR_3`, is defined for use by routines which may benefit from using vector notation. Many of the components of the `generic_` global variable structure are defined in terms of `VECTOR_3` elements, with an alternative set of three scalar names defined for convenience.

`ls_constants.h` This header file defines useful constants, such as `PI`, equatorial radius of the earth `EQUATORIAL_RADIUS` as well as its square `RESQ`, earth geodesy parameters `FP`, `E`, and `EPS`, the inverse of nominal gravitational acceleration `INVG`, the rotation rate of the earth, `OMEGA_EARTH` (in radians per second), useful conversion factors `V_TO_KNOTS`, `DEG_TO_RAD`, and `RAD_TO_DEG`, and standard sea-level atmospheric density, `SEA_LEVEL_DENSITY`, in English units (slug/ft^3).

`ls_generic.h` This header file defines the `generic_` aircraft parameter global structure which is used to pass global parameters between aircraft subsystem models and the various equations of motion routines. The generic parameters provide the common aircraft state information (positions and velocities) as well as other parameters such as accelerations, forces and moments, vehicle geometry, mass and inertia, and atmospheric properties. A complete description of the contents of the `generic_` data structure is given in Appendix A.

`ls_sim_control.h` This header file defines the `SIM_CONTROL` global structure which is used to indicate command-line and other options set by the user. It contains the mode flag `sim_type` to indicate what mode of operation has been requested (`batch`, `terminal`, `GLmouse`, or `cockpit`), as well as information about run number, date and time stamps, and output formats requested for trajectory information.

`ls_cockpit.h` This header file defines the `COCKPIT` global structure which is used to pass pilot control position information between the cockpit (either a keyboard, mouse, or actual cockpit) and the rest of the simulation routines. Some abbreviations for locations within the `COCKPIT` structure are also provided for convenience.

`ls_err.h` This header file defines the `ERROR` global structure which is used to signal error conditions to the rest of the simulation. At present, the

only errors defined are those relating to the table lookup routines defined in `ls_funcgen.c`.

`ls_funcgen.h` This header file provides prototypes for the linear interpolation (data table lookup) routines available in this version of LaRCsim. See the section "Function Data Interpolation" above for more information.

`ls_matrix.h` This header file provides function prototypes for general real matrix manipulation routines; it is used by the `ls_trim` routines.

`ls_sym.h` This header file provides prototypes for various symbol table lookup and manipulation routines, `ls_findsym()`, `ls_put_sym_val()`, and `ls_get_sym_val()`. This particular header file is probably of not much interest to the casual LaRCsim user.

`ls_tape.h` This header file defines the time-history data recording structure, `tape_`, which is used in the `ls_record()` and `ls_write`*xxx*`()` routines, and is of not much interest to the casual LaRCsim user. However, the number of parameters that may be stored is determined by the definition of `MAX_TAPE_CHANNELS` which is contained in this header file (currently set to 1024 parameters).

Routines called in the main execution loop.

`ls_accel.c` The first of three main EOM routines. This function sums the body-axis forces and moments provided by the `aero()`, `engine()`, and `gear()` routines (these are written by the user; example `aero()` and `engine()` routines are found in the file `navion.c` included in this package) and calculates the resulting total angular and linear accelerations in geocentric coordinates. Forces and moments are taken to act at the reference point, which is fixed to the body. The center of gravity location is defined relative to the reference point by variables `D[xyz]_cg` (found in vector `D_cg_rp_body_v`). The total angular and linear accelerations are corrected to act through and about the center of gravity.

`ls_step.c` This is the second of the three main EOM routines. This function performs the integration of the vehicle accelerations and velocities to form the updated vehicle velocities and positions. The time variable, `Simtime`, is integrated as well. The integration of accelerations uses a predictive (forward) integration; the integration of velocities is a modified trapezoidal backwards integration algorithm. These integration routines have been used successfully at NASA-Ames, NASA-Langley, and NATC/NAWC Patuxent River for many years and are well proven.

`ls_aux.c` This is the third major EOM routine. This function calculates most of the auxiliary variables based upon the updated vehicle state, including conventional accelerometer readings at both the C.G. and the pilot station, new values for angles of attack, sideslip, flight path, Mach number, gravity, and numerous descriptions of velocity and position in several axes. The state variables for geocentric latitude, longitude, and radius are converted to more useful geodetic (map coordinates) latitude, longitude and altitude (M.S.L.) as well as runway relative coordinates from a prespecified runway.

The next three routines are called by the main EOM routines to perform supporting calculations.

`atmos_62.c` The 1962 Standard Atmosphere Tables for density and speed of sound, in cubic spline lookup format, along with the necessary

interpolation routines. Data is included from sea level to 240,000 ft.; however, the ambient temperature and pressure are described as parametric equations and are only valid to about 75,000 ft. in this version of LaRCsim.

`ls_geodesy.c` This function converts geocentric latitude and radius to geodetic latitude and altitude above sea level, and vice versa. It is based upon relationships provided in reference 3, which define the transformation from geodetic to geocentric; unfortunately, reference 3 doesn't include the opposite transformation, which is fairly complex. Since LaRCsim uses geocentric coordinates as *the* inertial axes set, and performs the translational integrations in the geocentric frame, it is necessary to have a means to efficiently convert back to geodetic coordinates, since these are the coordinates most often used for navigation (map latitude, longitude, and altitude). The `ls_geoc_to_geod()` routine, found in the `ls_geodesy.c` module performs this approximate conversion. Note: recently an engineering note in the AIAA Journal of Guidance, Control and Dynamics describes a closed-form solution; it is quite complex and has not yet been evaluated for this application (reference 9).

`ls_gravity.c` This routine calculates the value for local gravity, based upon geocentric latitude and radius, including effects due to oblateness of the earth (harmonics), based on equations given in reference 3.

The user-supplied aircraft model is called by the next routine.

`ls_model.c` This routine is an executive to the vehicle (user supplied) routines `engine()`, `subsystems()`, `aero()`, and `gear()`, or whatever set of routines the user decides are needed to adequately model the vehicle properties.

Any functions that are not satisfied by user-provided routines are provided by the next routine:

`default_model_routines.c` This module contains stub routines for what are normally user-provided functions, `inertias()`, `subsystems()`, `engine()`, and `gear()`. If these are not provided by the user, these stub routines satisfy the loader at link time, with no ill effects aside from fixed weight, thrust, and the lack of ability to land. The user *must* provide initial values of certain mass properties, as well as force and moment vectors, in a routine named `model_init()`. See the section above on creating a new model for more details on what parameters must be initialized by user software.

Data logging is provided by a call to the next routine:

`ls_record.c` This routine stores preselected global variables into a data structure for later playback or analysis. `ls_record()` automatically saves 19 channels of data (e.g. these outputs are hardwired) that contain state and basic input information from each run; in addition, the user can specify (through the settings file) additional parameters to record. Variables are addressed via memory locations found in the debugger symbol table of the executable; for this reason, the various modules that comprise a LaRCsim executable **must** be compiled and linked using the symbol table option (usually a -g switch). A LaRCsim simulation that can't locate a specified variable will complain at invocation, but continue to execute; those parameters that are not found will not be recorded. The data structure `TAPE` utilizes a circular buffer that, when full, begins to replace the oldest

time history data with newer data. In the version 1.4 distribution of LaRCsim, this buffer records every eighth time slice.

Finally, interfaces with the pilot and synchronization with the real world are accomplished by the following routines:

`ls_sync` This module contains routines involved with synchronizing the operation of LaRCsim to match simulated time with real-world time on some UNIX platforms. The portability of this module is in question, however. It makes use of system services `signal()`, `setitimer()`, `pause()`, and the `itimerval` data structure, which are supported on both SGI (IRIX 5.2) and Sun (SunOS 4.1.3) platforms.

`ls_ifgl.c` This module contains an IRIS GL (Graphics Library) interface for interactive runs on Silicon Graphics computers (running IRIX 5.x), as well as dummy synchronization routines (which aren't needed if run under GL, since the drawing calls effectively synchronize to real-time). This module replaces `ls_ifsun.c` for Silicon Graphics implementations.

`ls_ifterm.c` This module contains a simple interface for interactive runs on most Unix computers, using the `curses` library of terminal routines, as well as routines to synchronize simulation with real-time, using standard unix system routines `setitimer()`, `signal()`, and `pause()`. It was the intent of the author to keep the routines very generic, without relying on either BSD or System V style system calls; our ignorance of these nuances may well show through, however; this routine works well on a SunSPARCstation-1 and -2, and will work on an SGI IRIS 4D machine.

`ls_ACES.c` This module contains driver code to communicate with the Advanced Controls Evaluation Simulator (ACES) cockpit used in the Dynamics and Control Branch, and is of little interest to the non-DCB LaRCsim user.

Support routines. The following routines provide additional services for the LaRCsim application, and are not typically called during the main simulation loop:

`LaRCsim.c` This routine is provided as an example executive function to call the appropriate routines in the proper sequence both prior, during, and at the end of a simulated run. `LaRCsim.c` includes the `main()` procedure for the simulation. It also interprets any command line options provided by the user, and initializes some simulation data structures with default values. At the conclusion of the simulation, it calls the output routines `ls_writemat`, `ls_writeav`, `ls_writetab`, and `ls_writeasc1`.

`ls_err.c` This module reports errors in a semi-meaningful way. By properly loading the `ERROR` structure elements (see `ls_err.h`) and then calling `print_error()`, a LaRCsim routine can have an error message printed on stderr.

`ls_funcgen.c` The `ls_funcgen` module provides a simple linear interpolation routine for doing function generation using data tables. At present, this routine is limited to functions of six dimensions and 63 breakpoints along each dimension. It reports errors via the `ls_err()` routine. See the section above on "Function Data Interpolation" for more information on using this capability.

`ls_init.c` This routine calls the EOM functions and the user-supplied vehicle initialization routines in the proper sequence to initialize the vehicle prior to a run, or to reset at the end of a run.

`ls_matrix.c` This module contains several utilities to create, delete, print, and invert general real matrices. It is used by the trim routine.

`ls_settings.c` This module contains the code that deals with settings files. Two main routines are defined: `ls_get_settings()` and `ls_put_settings()`. A single parameter, `desired_file_name`, is accepted by `ls_get_settings()`. Calling `ls_get_settings()` with a file name specified will cause a search for a file by that name along the `LARCSIMPATH` directory path; if a null string is passed to `ls_get_settings()`, a default settings file with the name of the executable simulation program, prepended with a '.', will be hunted for along the path. If either file is found, that file will be opened, read into memory, and parsed by the `ls_parse_settings()` routine. A table of facilities is kept that provide entry points for both reading and writing each type of information (e.g. trim, init, record). `ls_parse_settings()` will call the appropriate routine as the designated keyword is found, passing a pointer to the appropriate location in the file buffer to that routine. If `ls_parse_settings()` encounters a line in which the first non-blank characters is '`@`', it will use the characters following the '`@`' sign as a file name, search for and open that file, and recursively call itself. A call to `ls_put_settings()` will create a default settings file, replacing the previous one, if it exists, and then calls each facilities' `put_settings()` routines, as kept by the facility table, in sequence, causing the current LaRCsim settings to be recorded.

`ls_sym.c` This routine performs symbol table lookups to resolve static local and global variable names into virtual memory addresses. It is used by `ls_record()` to record time history data during run time. It is not intended for use by the general LaRCsim user; and its portability is in question, as this capability is usually highly platform-dependent. It does appear to work on SGI (IRIX 5.2) and Sun (SunOS 4.1.3) operating systems, however.

`ls_trim.c` This module contains a Newton-Raphson algorithm for solving simultaneous non-linear equations. Given n "control" parameters, `ls_trim()` will perturb those parameters and observe the effect upon n other "output" variables. After measuring these partial derivatives, using a single-sided difference approach, the algorithm makes a constrained step of all n controls simultaneously to try to reduce the root-mean-square value of the sum of the n outputs. This process repeats for up to `Max_Cycles` or until all outputs are within a specified tolerance of zero.

`ls_writeav.c` This module writes time history data from the `Tape` data storage structure to a file named `run.flt` at the end of the simulation session. This data file is in a format recognizable to the Agile-Vu trajectory visualization tool developed for Silicon Graphics workstations by McDonnell-Douglas and the Naval Air Development Center. The `-a` command line switch will choose this output format; by default, no `run.flt` file is created.

`ls_writeasc1.c` This module writes time history data from the `Tape` data storage structure to a file named `run.asc1` at the end of the simulation session. This data file is in a format recognizable to the GetData and XPlot programs, written for X-windows machines by the kind folk at NASA Dryden Flight Research Center. (see reference 11 for information on this time history format.) The `-x` command line switch will choose this output format; by default, no `run.asc1` file is created.

`ls_writetab.c` This module writes time history data from the `Tape` data storage structure to a file named `run.dat` at the end of the simulation session. This data file contains a ASCII based, tab-delimited listing of each parameter at each recording point; these files can therefore become quite large for a long simulation session. The `-t` command line switch will choose this output format; by default, no `run.dat` file is created.

`ls_writemat.c` This module writes time history data from the `Tape` data storage structure to a file named `run.m` at the end of the simulation session. This data file is in a format recognizable to a typical commercial matrix manipulation application. The `-r` command line switch will choose this output format; by default, no `run.m` file is created.

The following routines, contained in a separate directory, provide an example aircraft simulation including simple aerodynamic, engine, and initialization routines.

`navion.h` This header file defines a data structure that contains the linear aero coefficients, `COEFFS`, which can be made available for run-time modification of the example aircraft's aerodynamic properties and stability characteristics.

`navion_aero.c` A simple, linear aerodynamics model of the North American Navion for a trimmed level flight at 100 knots.

`navion_engine.c` This file contains a simple `engine()` routine with an optimistic thrust calculation that allows the venerable Navion to break Mach 1 in level flight.

`navion_gear.c` This module includes a fairly simple landing gear (mass-spring-damper) model of tricycle arrangement, and is not representative of the North American Aviation Navion.

`navion_init.c` This module initializes the mass properties and sets forces and moments and velocities to zero. It also initializes elements of the pilot and cg displacement vectors (relative to the reference point).

`Makefile` This makefile is used to build either a GL-based (for Silicon Graphics machines) or terminal-based version of the navion example LaRCsim executable. Invoke with `make` to generate the GL-based executable (which will be named `navion`), or specify `make terminal` to create the `curses`-based executable, `navion_term`.

`.navion` This ASCII data file contains a list of any parameters that are to be added to the recorded parameters list, as well as the desired set of trim parameters and initial condition states and controls. This file shows an example of the format to be used, and may be opened and modified with a text editor.

Theory of Operation

Inspection of the LaRCsim code (see Appendix B), beginning with the `main()` routine found in module `LaRCsim.c`, will demonstrate how and in what order the software is called. The `main()` routine initializes the contents of the `sim_control_` data structure and certain execution variables, such as the local variables `endtime`, `speedup`, `io_dt` (the terminal refresh period), `multiloop` (the number of model loops per terminal refresh), and `model_dt` (the model iteration time step). A call is then made to `ls_get_settings()` which opens the default settings file, if it exists, allows it to override these hardwired default values.

`ls_get_settings()` parses the default settings file and makes calls to `ls_record_get_settings()`, `ls_trim_get_settings()`, and `ls_init_get_settings()`, each of which initialize their various data structures and parse the appropriate section of the default settings file. `ls_get_settings()` then returns control back to `main()`.

`main()` then makes a call a call to `ls_check_opts()` which looks at any command line arguments, allowing them to override the default settings, if appropriate. (If the `-i` flag is encountered, for example, another call is made to `ls_get_settings()`, this time passing the name of the requested optional settings file). `ls_stamp()` is then called to generate a time and date stamp for the simulation run. These are stored in the `sim_control_` data structure.

The `main()` routine then calls `ls_init()`, which sets `Simtime = 0` and then initializes the initial conditions data structure. If no initial conditions were specified in the default settings or optional settings file, the initial conditions data structure is set to contain information about the thirteen rigid body and environment states. `ls_init()` then uses the values of the initial conditions data structure to set the simulation to the specified initial condition and then calls `model_init()`, normally a user-supplied routine. The sample routine provided in this package is found in file `navion_init.c`; it initializes control positions, inertia properties, vehicle forces and moments, and vehicle positions and velocities. Routine `ls_init()` then calls `ls_step()` with a time step of 0 and the initialization flag set.

Responding to the initialization flag, `ls_step()` initializes the integrator internal states ("past values") to zero, converts the initial geodetic latitude, longitude, and altitude values into geocentric latitude, longitude and radius (from the center of the earth) values; corrects the eastward velocity component to account for earth rotation; initializes the quaternion variables based upon the present Euler angles; initializes the local-to-body transformation matrix; calculates local gravity; and calls `ls_aux()` so that the miscellaneous output variables (such as angles of attack and sideslip, various velocities, and Mach number) reflect the current initial conditions. A call is then made to `ls_model()`. This routine calls the user-supplied vehicle routines `inertias()`, `subsystems()`, `aero()`, `engine()`, and `gear()`, passing to them a value of 0 for time step and with the initialization flag non-zero, indicating a reset is requested. These user-supplied routines calculate the forces and moments for the current flight conditions, setting the appropriate values in the `generic_` data structure. A call is then made by `ls_step()` to the `ls_accel()` routine to sum the forces and moments and calculate appropriate initial accelerations at the vehicle center of gravity. `ls_aux()` is then called to calculate the appropriate accelerometer outputs. `ls_step()` then sets the local variable `dt = 0` and performs the normal state integration equations. Since `dt` is 0, the vehicle state is not updated; however, the past values of the integration filters become initialized to the appropriate initial condition values. Control flow then returns to `ls_init()`, which returns control to `main()`.

Continuing with the initialization process, `main()` calls `ls_record()` to record the initial time history data. The initial call to `ls_cockpit()` is then made, which initializes either the GL screen or the terminal display, depending on which interface routine was linked in at compile time - either the `curses` library routines to draw a simple instrument panel on the terminal, or the IRIS GL routines to draw an out-the-window and heads-up-display (HUD) presentation on a Silicon Graphics screen. A call is then made to `ls_sync()`, with `io_dt` passed as a parameter, which schedules an interval timer to signal `SIGALRM` on timer expiration.

The real-time loop portion of the program is then entered. This consists of `multiloop` number of passes to `ls_loop()`. `ls_loop()` calls the following sequence: `ls_step()`, which advances the simulation one `dt` in simulated time to a new state; `ls_aux()` which calculates the new flight conditions, based on the new state; `ls_model()`, which calculates new control positions as well as vehicle forces and moments at the reference point; and finally `ls_accel()`, which sums the forces and moments at the vehicle reference point, transfers them to the center of gravity, and then calculates the resulting accelerations. `ls_loop()` then returns control to `main()`.

`main()` then calls `ls_record()`, to record the current flight conditions, velocities,

accelerations, and other parameters specified in the settings file. `main()` then makes a call to `ls_cockpit()` which refreshes the instrument panel display and gets new values for controls from the keyboard (or mouse, if GL is used). `ls_cockpit()` returns a non-zero integer if the user has signaled a desire to end the simulation. If `ls_cockpit()` returns zero, `ls_pause()` is called to await the arrival of the `SIGALRM` signal, which is caught and rescheduled, with command passing back to `main()` (see file `ls_sync.c`). If `Simtime` has exceeded the value of `endtime` or `ls_cockpit()` returned a non-zero value, the simulation calls the `ls_unsync()` and `ls_cockpit_exit()` routines, writes out any data files, calls `ls_put_settings()` to update the default settings file, and the program exits.

Concluding Remarks

This report describes how to implement, modify, and utilize a generic flight simulation software package on a UNIX-based computer. A description of each routine and all global variables are provided. The software is written entirely in ANSI C; listings of each routine are provided as well.

The structure of the code lends itself to pilot-in-the-loop operation on a sufficiently fast computer, and can be operated from a display terminal, a keyboard and mouse on a Silicon Graphics computer, or some modification, with an actual simulator cockpit. Time histories of selected parameters may be recorded in a variety of formats.

This software is patterned after similar FORTRAN routines used at the Manned Flight Simulator facility at the U.S. Navy's Naval Air Warfare Center/Aircraft Division, Patuxent River, Maryland. Those routines were themselves rewrites of older FORTRAN simulation routines that comprised a simulation architecture called BASIC used at NASA-Ames since the early 1970s.

The potential user is cautioned that results obtained from this software should be validated using conventional design methods. It is believed that equations of motion are implemented properly, but a full validation of LaRCsim against a benchmark simulation has not yet been performed. Simulated flight near either the North or South pole should be avoided, due to a singularity in the vehicle position calculations at either pole.

A copy of the latest version of this software is available upon request:

E. Bruce Jackson
MS 489
NASA Langley Research Center
Hampton, VA 23681-0001
e.b.jackson@larc.nasa.gov
(804) 864-4060

References

1. McFarland, Richard E., *A Standard Kinematic Model for Flight Simulation at NASA-Ames*, NASA CR-2497, January 1975.
2. ANSI/AIAA R-004-1992, *Recommended Practice: Atmospheric and Space Flight Vehicle Coordinate Systems*, February 1992.
3. Stevens, Brian L. and Lewis, Frank L., *Aircraft Control and Simulation*, Wiley and Sons, 1992, ISBN 0-471-61397-5.
4. Anon., *U. S. Standard Atmosphere, 1962.*
5. Anon., *Aeronautical Vest Pocket Handbook, 17th edition*, Pratt & Whitney Aircraft Group, Dec. 1977.
6. Halliday, David, and Resnick, Robert, *Fundamentals of Physics, Revised Printing*, Wiley and Sons, 1974, ISBN 0-471-34431-1.
7. Beyer, William H., editor, *CRC Standard Mathematical Tables, 28th edition*, CRC Press, Boca Raton, FL, 1987, ISBN 0-8493-0628-0.
8. Dowdy, M. C., Jackson, E. B., and Nichols, J. H., *Controls Analysis and Simulation Test Loop Environment (CASTLE) Programmer's Guide, Version 1.3*, TM 89-11, Naval Air Test Center, Patuxent River, MD, March 1989.
9. Zhu, J. *Exact Converions of Earth-Centered, Earth-Fixed Coordinates to Geodetic Coordinates.* J. Guidance, Control and Dynamics, vol. 16, no. 2, March-April 1993, p 389.
10. Morelli, Eugene A., *Nonlinear Aerodynamic Modeling using Multivariate Orthogonal Functions*, AIAA 93-3636, presented at the AIAA Atmospheric Flight Mechanics Conference, August 1993, Monterey, CA.
11. Maine, Richard E., *Manual for GetData Version 3.1, A FORTRAN Utility Program for Time History Data*, NASA TM-88288, October 1987.

Appendix A: LaRCsim Global Variables

Appendix B: Source Code Listings