ACSL

MODEL

ACSL REFERENCE MANUAL
VERSION 11

MGA SOFTWARE

Advanced Continuous Simulation Language (ACSL)

Reference Manual

Version 11

MGA Software
Concord MA 01742 USA

ACSL Reference Manual, Edition 11.1

ISBN 0-925649-14-7

Foreword

This manual describes the Advanced Continuous Simulation Language, ACSL (pronounced "axle"). The language is designed for modelling and evaluating the performance of continuous systems described by time-dependent, nonlinear differential equations.

Typical applications

Applications of ACSL in new areas are being developed constantly. Typical areas in which ACSL is currently applied include control system design, missile and aircraft simulation, chemical process dynamics, power plant dynamics, plant and animal growth, toxicology models, vehicle handling, microprocessor controllers, and robotics.

Prerequisites

This manual assumes that you are an engineer or scientist with some experience in computers and programming. You thus know the basics of calculus and of FORTRAN and are familiar enough with the computer operating system to manage and edit files.

Related documents

This manual documents the machine-independent elements of ACSL. Documents describing installation and execution on a particular computer are published separately and shipped with each ACSL system. An *ACSL Beginner's Guide* is also available.

How to use this manual

We recommend that you read through the first three chapters and then turn to the example programs in Appendix A. The ACSL statements (Chapter 4) and runtime commands (Chapter 5) are in alphabetical order for reference.

The linear analysis and frequency response features are described under the ANALYZE command in Chapter 5. System symbols are listed in Appendix C.

Before running a simulation, be sure to study Chapter 7 on debugging.

Table of Contents

1. Introduction

1.1 Simulation language

Simulation of physical systems is a standard analysis tool used in the evaluation of hardware design prior to actual construction. The continuous system simulation language described herein, and referred to as ACSL (Advanced Continuous Simulation Language), has been developed expressly for the purpose of modelling systems described by time dependent, nonlinear differential equations and/or transfer functions.*

Applications

Typical application areas for continuous simulation are control system design, chemical process representation, missile and aircraft simulation, power plant dynamics, biomedical systems, vehicle handling, microprocessor controllers, fluid flow, and heat transfer analysis. Users can derive model code from block diagrams, mathematical equations, conventional Fortran statements, etc.

Graphical front end

A graphical front end for ACSL is available in a software package called Graphic Modeller. Block diagrams are developed on screen from pre-defined PowerBlocks representing ACSL statements, operators, functions, and/or user-defined blocks in an unlimited hierarchical structure. ACSL code is generated directly from the graphical model.

Language highlights

Highlights of the ACSL language are free form input, function generation of up to three variables, and independent error control on each integrator. Flexibility is provided for plotting the behavior of the models under a number of external forcing functions. Many simulation oriented operators such as variable time delay, dead zone, backlash, and quantization are included and made readily accessible. Global single or double precision calculation can be selected.

The ACSL program is intended to provide a simple method of representing these mathematical models on a digital computer. Working from an equation description of the problem or a block diagram, the user writes ACSL statements to describe the system under investigation. Runtime commands exercise the model; these statements can be submitted for solution in batch mode or entered interactively. Interactively you can run the simulation, look at the results, and change constants to experiment with the model; for example, trying different spring constants or actuator limits.

An important feature of ACSL is its sorting of the continuous model equations, in contrast to general purpose programming languages such as Fortran where program execution depends critically on statement order. ACSL program structure and the translator sorting procedure are described in Chapter 3.

---

* The basic structure follows the specification established by the Technical Committee on Continuous System Simulation Languages and under the auspices of Simulation Councils, Inc. (SCi) in *Simulation* **9** (Dec. 1967) pp 281-303.

1.2 Job processing

Inputs to ACSL are in two parts. The *program* defines the system being modelled; the *runtime commands* exercise this model (*i.e.*, they change parameters, execute runs, specify plots, etc.).

The program is read by the ACSL translator, which translates it to a Fortran compile file. The Fortran code is then compiled, linked with the ACSL runtime library, and executed, finishing at an ACSL runtime prompt:

```
ACSL>
```

At this point, ACSL is waiting for runtime commands, which can be entered interactively, by reading a command file, or in a batch process. The commands are read, decoded, and executed in sequence. See Chapter 5 for a description of the commands. A minimum of a START command is necessary to exercise the model.

1.3 Language features

The language consists of a set of arithmetic operators, standard functions, a set of special ACSL statements, and a MACRO capability. The MACRO capability allows extension of the special ACSL statements, either at the system level for each installation, or for each individual user.

Operators and functions

The arithmetic, relational, and logical operators, described in Chapter 2, work as they do in Fortran. The functions, listed in Chapter 4, consist of special ACSL operators such as QNTZR (quantization), UNIF (uniform random number), etc. In addition, all the functions of the Fortran library are available; *e.g.*, SQRT (square root), MOD (modulus), etc.

Integration

Integration is a special ACSL operator that is called by either INTEG (scalar) or INTVC (vector). This is written in the form:

```
r = INTEG(x, rzero)
```

which implies:

$$R = RZERO + \int_0^T X\,dt$$

This integration operator is the heart of the simulation system. In building any model, it is necessary to change differential operators into integration operators; this is conventionally accomplished by expressing the highest derivative of a variable in terms of lower derivatives and other variables. As an example, consider the spring dashpot system excited by a given function of time F(t). In general form, this is:

$$\ddot{x} + 2\zeta\omega\dot{x} + \omega^2 x = F(t)$$

where ω is the natural frequency and ζ the damping ratio. Expressing this equation in terms of the highest derivative, $\ddot{x}$, gives:

$$\ddot{x} = F(t) - 2\zeta\omega\dot{x} - \omega^2 x$$

INTEG operator

This derivative can then be integrated once for $\dot{x}$ and again for x. Since $\dot{x}$ appears on the right hand side, it must be given a name (XD). The two integrations can be written in the program as:

```
xd = INTEG(F(T) - 2*ze*w*xd - w**2*x, xdic)
x  = INTEG(xd, xic)
```

This process transforms the original set of differential equations to a set of first order equations which can be solved directly by integrating.

Redundant state variables

Be careful in the transformation process to avoid the introduction of redundant state variables. In the above sequence, the reference to $\dot{x}$ (XD) could have been avoided by calculating $\ddot{x}$ (XDD) directly and embedding an INTEG operator in the expression; *i.e.*,

```
xdd = F(T) - 2*ze*w*INTEG(xdd, xdic) - w**2*x
```

Now X is the second integral of XDD and can be calculated as follows:

```
x = INTEG(INTEG(xdd, xdic), xic)
```

In these two equations, there are three INTEG operators (each one a state variable), but two are integrations of XDD with the same initial condition. One of these is redundant and can be eliminated by explicitly naming the first derivative as shown previously.

Limit cycle example

As an introduction to the process of coding a model, we will outline a simple example using the arithmetic operators and the SQRT and INTEG functions. The following equations define a limit cycle:

$$\dot{x} = y + \frac{x(1 - x^2 - y^2)}{\sqrt{x^2 + y^2}} \qquad ; x(0) = 0.5$$

$$\dot{y} = -x + \frac{y(1 - x^2 - y^2)}{\sqrt{x^2 + y^2}} \qquad ; y(0) = 0.0$$

The equations are coded as follows:

```
r2 = x**2 + y**2
x  = INTEG(y + x*(1.0 - r2)/SQRT(r2), 0.5)
y  = INTEG(-x + y*(1.0 - r2)/SQRT(r2), 0.0)
```

Program code

While this series of statements completely describes the equations to be solved, it does not represent the complete problem statement. Figure 1-1 lists the complete running program. Each statement is numbered for reference. A DERIVATIVE statement (1) is needed to define the beginning of the program. The communication interval (data logging rate) is specified by the CINTERVAL operator (3). A termination condition must be specified in a TERMT statement (7). Finally, the program definition is completed by an END statement (8) to match the DERIVATIVE statement. This END statement tells the translator that no more ACSL statements are expected, although Fortran subroutines or functions can be added at the end of an ACSL program.

A further refinement is in assigning symbolic names for the initial conditions, X(0) and Y(0), and for the stop time TF. These are preset in the CONSTANT statement (2). It is better programming practice to use a symbol preset to the value desired than to embed literal constants (0.5, for example) in the program. Changes then occur in one place in the program, and values can also be changed at runtime.

Runtime commands

The second part of Figure 1-1 is an example of runtime statements. OUTPUT (9) defines the variables that are to be listed on the screen at each communication interval

```
(1) DERIVATIVE ! limit cycle
(2)       CONSTANT       xz = 0.5  , yz = 0.0  ,tf = 4.0
(3)       CINTERVAL      cint = 0.2
(4)       r2       = x**2 + y**2
(5)       x        = INTEG( y + x*(1.0 - r2)/SQRT(r2), xz)
(6)       y        = INTEG(-x + y*(1.0 - r2)/SQRT(r2), yz)
(7)       TERMT(t .ge. tf, 'Time Limit')
(8) END


(9) OUTPUT t,x,y,sq
(10) START
(11) ! Change initial conditions and run again
(12) SET xz=0.7
(13) START
(14) QUIT
```

Figure 1-1. Limit cycle example of model code and runtime commands

as the run progresses. The model is then executed by the command START (10). Line (11) is a comment, (12) changes the initial condition, and (13) runs the model again. Line (14) quits from the runtime session.

Section 1 of Appendix A lists the actual program, gives more detailed explanations, and shows examples of results and plots.

1.4 Coding procedure

The ACSL coding line contains eighty columns. ACSL statements may be placed anywhere on the line, but we suggest that developing standards to make programs easier to read. Suggested standards are as follows:

Section delineators (`PROGRAM`, `INITIAL`, `END`, etc.)	start in column 1
PROCEDURAL brackets (`PROCEDURAL/END`)	start in column 6
Unlabelled statements	start in column 7
Labelled statements	start label in column 6
Equal sign in assignment statements	in column 15 or beyond

Separator More than one statement can be placed on one line by separating the statements with a semicolon (;).

Continuation Any statement can be continued on to the next line by adding an ampersand (&) at the end of the line, to the right of any nonblank information but before column 80. Any trailing blanks are squeezed out of the line containing the ampersand and the following line is added directly. Starting with column 1, it is as though the characters were strung on the end after the last nonblank character of the preceding line. Leading blanks of the continuation line are not suppressed unless a leading ampersand is present on the continuation line. Comments can be added after the trailing ampersand since these comments are stripped off first. An empty line needs two ampersands since a single one can't be distinguished.

Comments Everything after an exclamation point (!) is considered a comment and is ignored by the translator (in the program) and the runtime executive (in runtime commands).

Blank lines Blank lines can be placed anywhere in the program and have no effect.

Table 1-1. System variable defaults

Statement	Default name	Default value	Definition
ALGORITHM	IALG	5	Integration algorithm
CINTERVAL	CINT	0.1	Communication interval
ERRTAG	none	.FALSE.	Error flag
MAXTERVAL	MAXT	1.0E+10	Maximum calculation interval
MINTERVAL	MINT	1.0E-10	Minimum calculation interval
VARIABLE	T	0.0	Independent variable

Note: It is probably a good idea to treat these names as reserved and not use them for any purpose except that stated.

1.5 Reserved names

The only reserved names in the language are those of the form Znnnnn and ZZaaaa, where *n* is any digit and *a* is any alphanumeric character. All generated variables and system subroutine names are in this form.

System variable defaults

It is a good idea, in addition, to keep in mind the system variable default names listed in Table 1-1. These default names can be changed to any name, but if names are not specified, the default names will be in existence so they can be set by program control in the model definition section. Other uses of these names (*e.g.*, defining MAXT as a state variable) could cause conflicts.

Runtime system symbols

A further level of names to be considered is the runtime system symbols listed in Appendix C. These are flags and numeric values that determine such things as plot format, print intervals, etc. Quantities such as the Y axis length (YINCPL) for line plots, grid line type (GLTPLT), etc. are under your control at runtime. All these symbols have default values. Access is by runtime SET command; *e.g.*,

```
SET YINCPL = 5.0
```

Precedence of names

Symbols in SET commands are searched for first in the user's dictionary; *i.e.* the dictionary created from the variable names in the program code. If that search is unsuccessful, the dictionary of ACSL system symbols is searched. Thus, if one of the system symbols is used as a name in the program, that use takes precedence over the ACSL system use. You should generally not duplicate system symbol names within the program since doing so means that you will not have the luxury of modifying the system default values.

1.6 Fortran subroutines

User-defined Fortran functions or subroutines can be used in an ACSL program by including them at the end of the program; *i.e.* immediately following the last ACSL END statement. The translator looks for the following:

```
REAL FUNCTION ...
INTEGER FUNCTION ...
LOGICAL FUNCTION ...
CHARACTER FUNCTION ...
DOUBLE PRECISION FUNCTION ...
SUBROUTINE ...
FUNCTION ...
PROGRAM ...
```

If one of these keywords is found, the translator assumes that all the subsequent lines are to be passed directly to the Fortran compiler without any changes, so Fortran format must be followed exactly; *i.e.*, start in column seven or beyond, etc.

First line It is important that the line following the ACSL model really contain one of the Fortran statements listed above. A Fortran comment (C in column one) is rejected since it could be the beginning of a valid ACSL line; *e.g.*,

```
c = k1*x
```

Tabs The first line cannot contain a tab since ACSL sees a tab as a space until it determines that it is in Fortran. Tabs are acceptable in the Fortran code itself.

TOO MANY ENDS If text following the ACSL model does not correspond to the beginning of a Fortran routine, the error message *Too many ends* is produced. A model definition with too many END statements causes the model file to close prematurely. The first statement after an extra END statement looks the same to the translator as an incorrect Fortran subroutine header.

$ for common blocks The only change the translator makes in transferring the Fortran code is to look for a dollar sign (`$`) in column one. This line is replaced by all the ACSL common blocks called `/ZZCOMi/` and type statements (which may run to many lines). Using this feature makes the names and values of an ACSL program available to the subroutine. The dollar sign should appear before any DATA or executable statements, but can be after type statements or comments.

Note that there is no selectivity in pulling in the common blocks; either all the symbols are obtained or none. It is easy to get name conflicts or violate the translator sorting algorithm by this action. We recommend using your own common blocks where necessary to make names from one program or subroutine available in another.

2. Language Elements

2.1 Introduction

Most of the basic elements of ACSL are defined as in Fortran. Table 2-1 lists the major differences. If you are familiar with Fortran, you could skip to the next chapter after reading the table; if not, you should be sure you understand the language elements described in this chapter.

2.2 Constants

Constants may be either integer, floating point (single or double precision), logical, or character.

Integer — An integer constant is a whole number, written without a decimal point, exponent, or embedded blanks. Positive numbers may be prefixed with a plus sign; negative numbers must be prefixed with a minus sign. The maximum length of an integer is machine dependent. Subscripts of arrays are generally limited to five digits. Examples of integer constants include:

```
0     +526     -63     476
```

Table 2-1. Major differences between ACSL and Fortran syntax

labels	Labels can be alphanumeric or numeric. Labels are separated from the statements to which they are attached by a colon; *i.e.*, `label: statement` Due to conflicts with macro expansions within labelled statements, we recommend that labels be used only with CONTINUE statements. See the section on labels.
names	Blanks are not allowed within names. Names may have up to thirty-one characters. Names should not be of the form Znnnnn or ZZaaaa where *n* is any number and *a* is any alphanumeric character.
types	Variables starting with I, J, K, L, M, or N are *not* automatically considered integer. All variables and functions are considered floating point (single or double precision depending on the library specified) unless typed explicitly otherwise.
coding	Free format; use any columns 1 through 80. The translator can be instructed to ignore columns 73-80 with switch `-tll` (see *How to Use ACSL* for your system).
continuation	An ampersand (`&`) at the end of a line indicates continuation onto the next line. A non-blank column six has no significance in free format.
comments	An exclamation point (`!`) indicates a comment. ACSL ignores all code from an exclamation point to the end of the line. A C in column one has no significance in free format.