SimGauss V2.0

by Matthew Ford

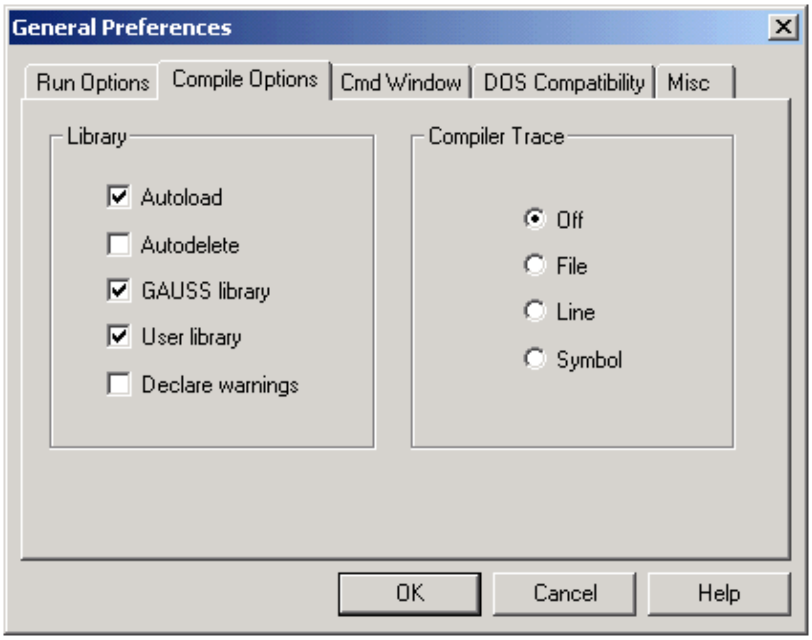
SimGauss V2.0 is a Non−Linear Simulation module for use with GAUSS Version 5.0 or higher. SimGauss has been designed to allow systems described by time dependent, non−linear differential equations and/or state space equations to be modelled using the GAUSS programming language. Both continuous and sampled data systems can be modelled.

SimGauss Installation

1. Unzip (or untar) the SimGauss files to your GAUSS directory. This will place the SimGauss examples in the **examples** sub−directory, the source files in the **src** sub−directory and create a **simgauss** sub−directory for your licence file.

2. Place your SimGauss licence in a file called **simgauss.lic**, in the **simgauss** sub−directory.

3. Set AUTOLOAD ON and AUTODELETE OFF. Use the GAUSS V5.0 **Configure** menu, choose **Preferences**, click on the **Compile Options** tab and tick Autoload and untick Autodelete.



Library
- [✓] Autoload
- [ ] Autodelete
- [✓] GAUSS library
- [✓] User library
- [ ] Declare warnings

Compiler Trace
- ● Off
- ○ File
- ○ Line
- ○ Symbol

SimGauss is designed to be run with AUTOLOAD ON and AUTODELETE OFF. It can be run with AUTODELETE ON but because most of SimGauss' procedures use global variables, if you make a typing mistake and GAUSS returns the error message 'Undefined Procedure' most of the SimGauss procedures will be deleted from the workspace. Then when you next compile a model SimGauss will be reloaded overwriting any variables you may have set up in the workspace.

4. Set the SimGauss and Publication Quality Graphics library in order to run the SimGauss tutorial examples and to use the SimGauss plotting procedures, SGXY etc.

```
library simgauss, pgraph;
```

This line can be added to your GAUSS **startup** file.

Installation Test

To test that the installation is correct run the Simple Model in Chapter 2 of this manual.

Documentation Version 281102

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Chapter 1

Introduction

This chapter will outline the features of SimGauss and the structure of this tutorial. It will also cover the customizing of SimGauss. To install SimGauss refer to the installation instructions at the front of this manual.

1.1 Introduction

SimGauss is a Non−Linear Simulation module for use with GAUSS Version 5.0 or higher. SimGauss has been designed to allow systems described by time dependent, non−linear differential equations and/or state space equations to be modelled using the GAUSS programming language. Both continuous and sampled data systems can be modelled. The results of the simulations can be plotted using the GAUSS Publication Quality Graphics module.

Some of the features of SimGauss are :−

- SimGauss is interactive. All the model variables are available for display and plotting, and their values can be changed from the GAUSS command mode or under program control.

- SimGauss is vectorized. In keeping with GAUSS' matrix based programming language, SimGauss can integrate state vectors directly from the state−space matrix equations. Alternatively vectors of parameters can be defined to evaluate the effect of different values in one run.

- SimGauss is fast. The interactive programming environment of GAUSS makes development and debugging of models quick and easy. You produce results faster. SimGauss' vector capabilities simplify the modelling of state−space systems and allow the fast investigation of parameter variations.

- SimGauss is extendible. Procedures are provided to model backlash, dead zones, tabular data, quantization, etc. The GAUSS programming language can be used to write other specialized procedures. Existing Fortran, C and Assembler procedures can also be linked to the simulation using GAUSS' Foreign Language Interface.

- SimGauss is flexible. SimGauss can handle both continuous and discrete systems, multiple and variable sampling rates. SimGauss also has a powerful user event facility which allows you to modify the simulation's conditions, at precise instants in time, while the simulation is running.

- SimGauss is savable. The entire SimGauss model, including the current state, can be saved using the GAUSS **SAVEALL** command. This allows you to come back to the simulation at a later time, just as you left it. The model in steady state condition can be saved and then various disturbances introduced, each time recalling the original steady state model before the next run. The results of the simulation can also be saved in a GAUSS data set for use by other GAUSS programs or for conversion to ASCII for use by external programs.

- SimGauss can be included in other procedures. Other GAUSS procedures can run the compiled SimGauss model. This allows model parameters to be optimized and two point boundary value problems to be solved.

- SimGauss has automatic step size selection. As well as the common 4th order Runge−Kutta integration algorithm, SimGauss has variable step size, variable order integration algorithms which automatically adjust their step size to keep the state errors within user definable bounds.

This tutorial will work through a number of examples illustrating various features of SimGauss. All the code for these models should have been copied to your GAUSS **examples** sub−directory as part of the installation procedure.

1.2 Customizing SimGauss

With AUTOLOAD ON and the correct library set, SimGauss can find and load any additional procedures your model uses. However some of these procedures such as the plotting routines are required for almost every simulation and take time to compile. SimGauss can be customized to have such commonly used procedures already included when SimGauss is loaded.

Before customizing SimGauss make sure you have installed it properly. (Refer to the installation instructions at the front of this manual.) **Only customize a copy of SimGauss, not the original distribution code.**

To customize SimGauss to include the XY plotting routines, clear out the GAUSS workspace, set the required libraries and load SimGauss using

```
new;
library simgauss, pgraph;
simgauss;
```

Just hit the **Enter** key in response to the SimGauss prompt for a model file name. This will terminate SimGauss. Then load the SimGauss XY plot procedure using

```
external keyword sgxy;
```

Any other files you wish to add to your SimGauss system, such as the **SGSAVEP** keyword should also be added now.

Then type

```
compile simgauss.csg;
```

to save the new SimGauss system. This will overwrite the old SIMGAUSS.GCC file so be sure you are working on a copy of SimGauss not the original.

As delivered, SimGauss uses approximately 230 global symbols and 100K bytes of workspace before a model has been compiled.

The Publication Quality Graphics require considerable memory and may not be able to run with SimGauss loaded on computers with limited memory. 'Saving and Loading the Plot Matrix' in Chapter 2, A Simple Model, explains how the Publication Quality Graphics can still be used to plot SimGauss results. This procedure can be speeded up by pre−compiling the Publication Quality Graphics and saving the workspace for use later.

The following commands will achieve this.

```
new;
library simgauss, pgraph;
external keyword sgloadp, sgxy, sglogx, sglogy, sgloglog;
saveall sgplot;
```

SGPLOT.GCC now contains the compiled Publication Quality Graphics routines which can be reloaded using

```
run sgplot;
```

This will overwrite the entire GAUSS workspace, so the current SimGauss workspace should be saved first using the GAUSS **SAVEALL** command.

Chapter 2

A Simple Model

This chapter presents an example that is the simplest, and most limited, form of SimGauss model that can be run. Using this model this chapter will show you how to compile and run the model and how to print and plot the results. This example should be used to test that SimGauss has been properly installed. Some of the common installation errors are covered in this section. Finally the structure of the code of this simple model is explained.

The standard installation process copies the tutorial example files into the **examples** sub−directory. The examples in this manual assume that your working directory is the GAUSS directory.

All the tutorial examples assume you are already in GAUSS and have a clear workspace. Just type

```
new;
```

before starting the next example if you have not exited GAUSS since running the last example. Refer to the SimGauss Reference Manual and the GAUSS Command Reference for a complete description of the SimGauss and GAUSS commands used here.

2.1 Spring System

This is a model of the mass−spring−damper system shown in Figure 2−1.

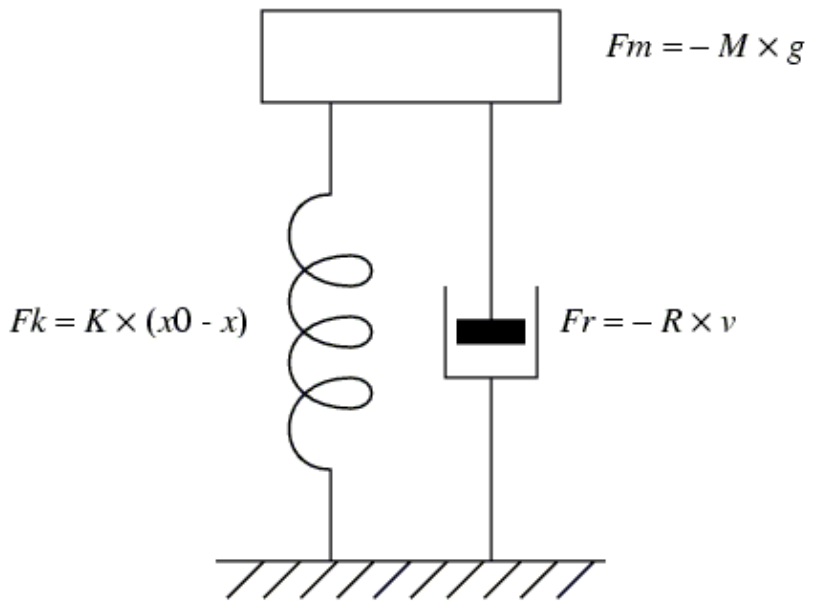


Figure 2−1

The system equations are :

$$M \times \frac{dv}{dt} = \Delta F = -M \times g + K \times (x0 - x) - R \times v$$
$$v = \frac{dx}{dt}$$

where

Fm	is the force due to gravity
Fk	is the spring force
Fr	is the damping force
K	is the spring constant (Kg/s2)
M	is the mass (Kg)
R	is the damping co−efficient (Kg/s)
x0	is the length of the spring at rest (m)

g	is the gravitational acceleration constant (m/s2)
v	is the velocity of the mass (m/s)
x	is the position of the mass (m)

2.1.1 Compiling the Model

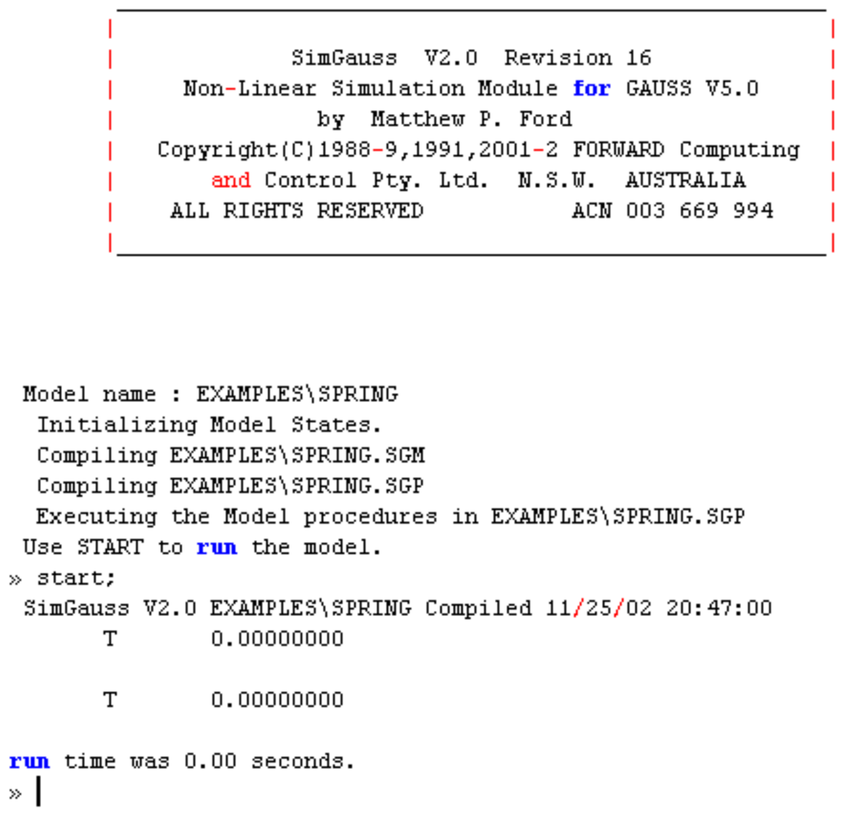
Having started up GAUSS, set the SimGauss and graphics' libraries and compile the model.

```
library simgauss, pgraph;
simgauss examples\spring
```

Then run the model using the SimGauss START command

```
start;
```

Your screen should then look like Figure 2−2 (the run times are for a Toshiba Laptop Statellite Pro 6100).



```
 ______________________________________________________
|                                                      |
|            SimGauss  V2.0  Revision 16               |
|   Non-Linear Simulation Module for GAUSS V5.0        |
|               by  Matthew P. Ford                    |
|  Copyright(C)1988-9,1991,2001-2 FORWARD Computing    |
|      and Control Pty. Ltd.  N.S.W.  AUSTRALIA        |
|   ALL RIGHTS RESERVED           ACN 003 669 994      |
|______________________________________________________|


 Model name : EXAMPLES\SPRING
  Initializing Model States.
  Compiling EXAMPLES\SPRING.SGM
  Compiling EXAMPLES\SPRING.SGP
  Executing the Model procedures in EXAMPLES\SPRING.SGP
 Use START to run the model.
» start;
 SimGauss V2.0 EXAMPLES\SPRING Compiled 11/25/02 20:47:00
       T       0.00000000

       T       0.00000000

run time was 0.00 seconds.
» |
```

Figure 2−2

2.1.2 Possible Installation Errors

If there were any errors go back and check that you have installed SimGauss correctly.

Some possible errors are :−

♦ There is no valid SimGauss licence file.

```
SG_TRANS.EXE (0) : SimGauss Translator error.
     Error could not open c:\gauss50\simgauss\simgauss.lic
```

Check that you have installed the simgauss licence file in the path indicated.

♦ There is no valid SimGauss licence.

```
SG_TRANS.EXE (0) : SimGauss Translator error.
     Error could not find valid SimGauss Key: in file
c:\gauss50\simgauss\simgauss.lic
```

Check that you have installed the correct SimGauss licence for this copy of GAUSS

♦ There is not enough GAUSS workspace memory to load SimGauss.

```
   (0) : error G0030 : : Insufficient workspace memory
```

Try exiting GAUSS and removing some of you memory resident programs, print spoolers, etc, or reduce the amount of main program space allocated, using the GAUSS **NEW** command. As delivered SimGauss requires approximately 100K bytes of workspace before a model has been compiled.

♦ There are not enough global symbols specified in the GAUSS workspace to load SimGauss.

```
   (0) : error G0026 : : Too many symbols
```

Use the GAUSS NEW command to increase the number of global symbols available. As delivered SimGauss requires approximately 230 global symbols before a model has been compiled.

♦ The AUTOLOAD feature of GAUSS is OFF, or the SIMGAUSS library has not been set.

```
   (0) : error G0025 : 'SIMGAUSS' : Undefined symbol
```

2.1.3 The Model Code

The model code required to implement this example is contained in the file **spring.sgm** All SimGauss model code files have the extension **.sgm**

```
1. /* spring.sgm   A simple mass/spring/damper system */
2.
3. DYNAMIC(0.1); /* Start of DYNAMIC section */
4.
5. /* Declare global constants */
6. DECLARE matrix M ! = 1; /* Kg mass */
7. DECLARE matrix K ! = 9.8; /* Kg/(s*s) spring const. */
8. DECLARE matrix R ! = 1; /* Kg/(s) damping co-effic. */
9. DECLARE matrix x0 ! = 5; /* length of spring at rest */
10. DECLARE matrix g ! = 9.8; /* gravit. acc. m/(s*s) */
11.
12. /* System equations */
13. d_v = (K.*(x0-x) - R.*v)./M - g;
14. d_x = v;
15. DECLARE matrix i_x ! = 5; /* mass initially 5 m off
16. :: the ground
17. */
18. END_DYN; /* End of DYNAMIC section */
```

There is only one section in this model, the DYNAMIC section which starts on line 3. The argument to the **DYNAMIC** procedure specifies the initial step size that the integration algorithm can take. In this model, because there is no DATALOG section or **DATASTEP** statement, the initial step size (0.1) is also used to set the data logging interval for the print and plot variables.

The **DECLARE** statements set up the necessary constants for the model. Since these assignments occur only at compile time they do not slow down the running of the model. Be sure to use either the ?= or != versions of the **DECLARE** statement or you will get a redefinition error when you try to translate the model for a second time.

The actual system equations are on lines 13 and 14 which define the derivatives of the mass' velocity, v, and its position, x.

The starting and stopping time of the integration is set by the SimGauss global variables, **i_t** and **stoptime**. The default values of these are zero.

2.1.4 Printing and Plotting Variables

Any matrix variable currently defined in the GAUSS workspace can be printed. The SimGauss keyword **PRTVARS** is used in the GAUSS command mode to set the variables to be printed. When SimGauss first starts up, the time, **t**, is the only variable printed. The format of the output is controlled by the GAUSS **FORMAT** command.

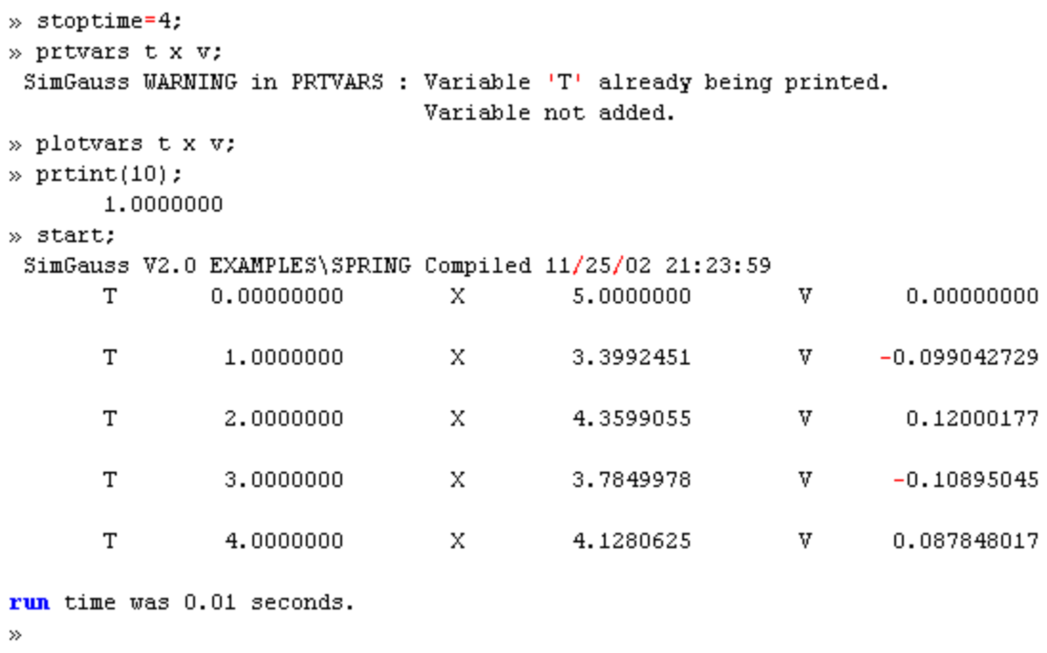
The SimGauss keyword **PLOTVARS** is used in the GAUSS command mode to set the variables to be saved for plotting. Only scalars or vectors (row or column) can be saved for plotting. Initially there are no plot variables specified.

The following commands extend the run time to 4 sec. and the set the print and plot variables to **`t`**, **`x`** and **`v`**.

```
» stoptime=4;
» prtvars t x v;
  SimGauss WARNING in PRTVARS : Variable 'T' already being printed.
                Variable not added.
» plotvars t x v;
» prtint(10);
       1.0000000
» start;
```

Also the default value for **PRTINT**, number of **DATASTEP**s between displaying the print variables, is 1. With the present **DATASTEP** interval of 0.1 sec. this would result in about 40 lines of output. To reduce this volume of printed output the print interval is set to display the print variables only every 10th **DATASTEP** interval.

Now re−run the model using **START**. Your screen should now look like Figure 2−3.



```
» stoptime=4;
» prtvars t x v;
 SimGauss WARNING in PRTVARS : Variable 'T' already being printed.
                               Variable not added.
» plotvars t x v;
» prtint(10);
      1.0000000
» start;
 SimGauss V2.0 EXAMPLES\SPRING Compiled 11/25/02 21:23:59
      T       0.00000000       X       5.0000000       V       0.00000000

      T        1.0000000       X       3.3992451       V     -0.099042729

      T        2.0000000       X       4.3599055       V       0.12000177

      T        3.0000000       X       3.7849978       V      -0.10895045

      T        4.0000000       X       4.1280625       V      0.087848017

run time was 0.01 seconds.
»
```

Figure 2−3

2.1.5 Saving and Loading the Plot Matrix

To plot the saved plot variables using GAUSS' Publication Quality Graphics routines you can use the SimGauss keywords **SGXY**, **SGLOGX**, **SGLOGY** and **SGLOGLOG**. Each of these keywords are followed by the variables to use as the X and Y axes. The Y axis variables are separated from the X axis variables by a comma.

The Publication Quality Graphics package requires considerable memory and may not be able to run with SimGauss loaded on computers with limited memory. In these cases the following commands can be used to save the plot matrix in a GAUSS data set then reload it for plotting using the Publication Quality Graphics. (Refer to the Section 1.2, 'Customizing SimGauss' in the Introduction for a fast way of loading the plot routines.)

```
sgsavep spring; /* save plot vars to a data set */
saveall spring; /* save the current workspace */
new;
library simgauss, pgraph;
sgloadp spring; /* reload the plot variables */
sgxy t,x;
```

which gives the graph in Figure 2−4.

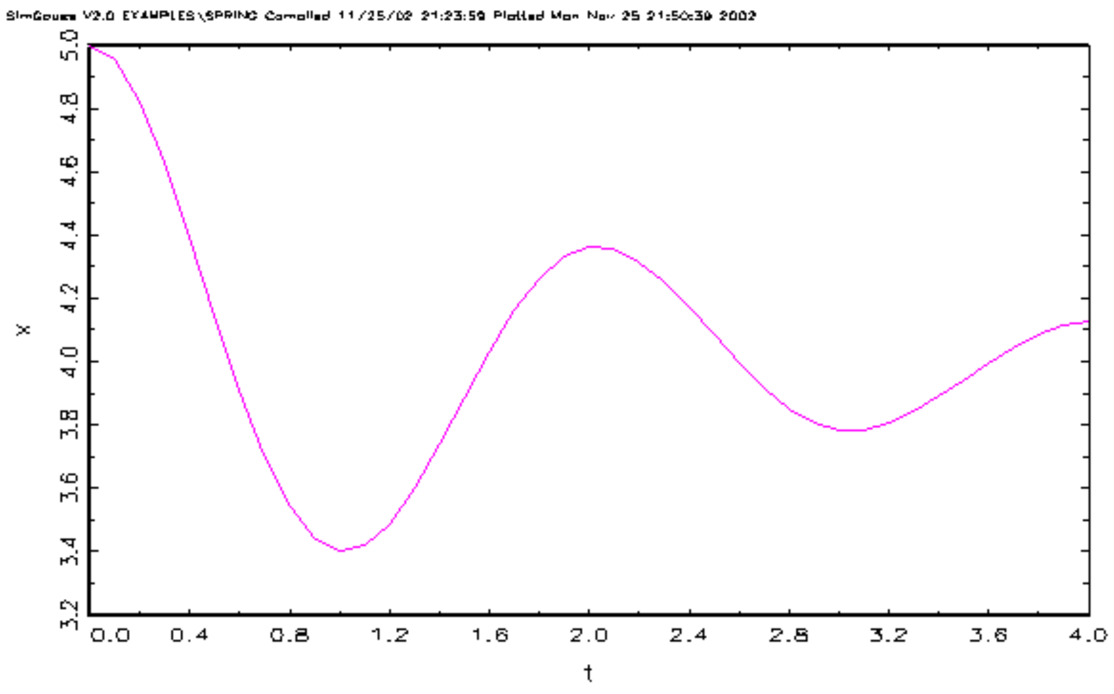


Figure 2−4

Chapter 3

Model Structure

This chapter will explain in detail the structure of a SimGauss model and the way in which each section is used in the simulation. The naming conventions used by SimGauss and the files produced by the translator will also be explained in this chapter.

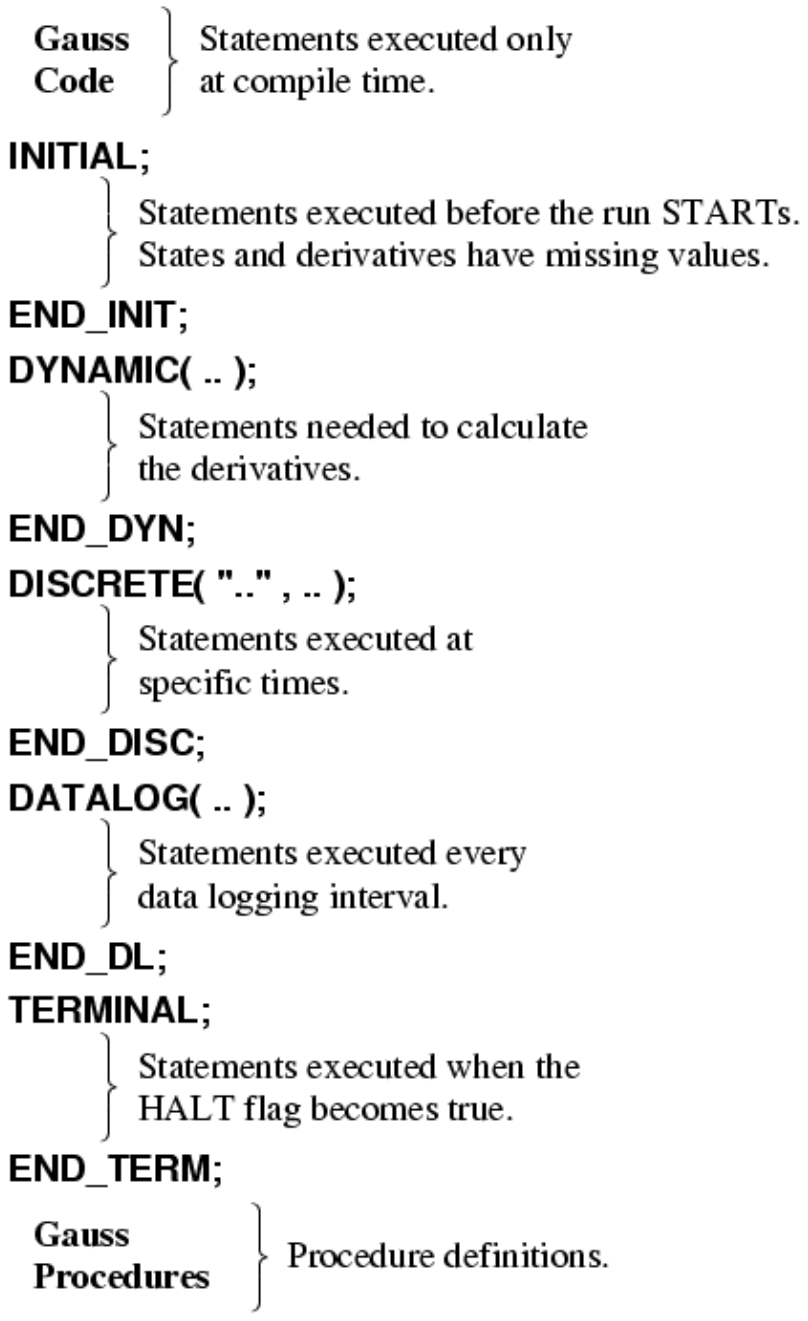
3.1 SimGauss Model Structure

The simplest SimGauss model has only a DYNAMIC section. The types of model that can be handled with this simple structure are very limited. SimGauss provides a larger, more flexible structure which includes

Gauss Code
INITIAL Section
DYNAMIC Section
DISCRETE Section(s)
DATALOG Section
TERMINAL Section
Gauss Procedures

These sections are shown in more detail in Figure 3−1.

All the sections, except the DYNAMIC section, are optional and there can be more than one DISCRETE section. However if any of the optional sections are present they must be in the order shown in Figure 3−1.



Gauss Code — Statements executed only at compile time.

INITIAL;

Statements executed before the run STARTs. States and derivatives have missing values.

END_INIT;

DYNAMIC(..);

Statements needed to calculate the derivatives.

END_DYN;

DISCRETE(".." , ..);

Statements executed at specific times.

END_DISC;

DATALOG(..);

Statements executed every data logging interval.

END_DL;

TERMINAL;

Statements executed when the HALT flag becomes true.

END_TERM;

Gauss Procedures — Procedure definitions.

Figure 3−1

The Gauss Code section of the model allows you to tailor the GAUSS environment that your model will run in, such as by setting load paths, libraries and the **stoptime**. This section of the model starts at the first line of code and ends at the first SimGauss section command. The Gauss Code section is **only** executed when the model is compiled, not when it is run.

The INITIAL section is executed every time the **START** command is used to run the model. It allows the initial conditions for the model to be calculated using any variables already defined in the GAUSS workspace. When this section is executed the state and derivative variables have missing values as the initial conditions are not transferred to the states until after the INITIAL section is executed.

The DYNAMIC section defines the state derivatives. Derivative statements are not allowed in any other section. The DYNAMIC code is called every time the integration routine wants to evaluate the derivatives at a given state.

The DISCRETE sections of the model are executed only at specific times. They can execute either repetitively, such as samplers for digital controllers, or only when **SCHEDULE**d, for example to change model's states abruptly.

The DATALOG section contains any code which is only needed for calculating output variables. This code is called at each data logging interval before displaying the print variables and saving the plot variables. The DATALOG section is not allowed to modify any of the model's variables. That is it can only assign values to variables that are not used in any other section of the model.

The TERMINAL section is called when the **`HALT`** flag is set, terminating the run. This section can be used to close output files, calculate performance values at the end of the run, etc.

The Gauss Procedures section is used to define any procedures used by the model. This section is never executed, only compiled, so any GAUSS code here which is outside a procedure definition is never executed.

An alternative method of including the procedures used by the model is to set up a GAUSS library and use the Gauss Code section to set the appropriate library file. When using SimGauss with AUTODELETE OFF any procedures referenced in the model code which do not appear in an active library need to have a **EXTERNAL** statement in the Gauss Code section of the model.

Each of the INITIAL, DYNAMIC, DISCRETE, DATALOG and TERMINAL sections of the model are translated into separate procedures which are called by the SimGauss procedures **START** and **GO**. This means you cannot branch outside these model sections using **GOTO** statements. Also **IF** and **DO** statements must not extend across section boundaries and procedure definitions are not allowed in these sections.

At present the SimGauss translator does not process **#INCLUDE** statements therefore these should not be used to include code that contains SimGauss section commands or derivative statements. Also, no GAUSS statements are allowed between the sections.

3.2 SimGauss Flow Chart

Two basic commands are provided to run a model. These are **START** and **GO**. The flow chart for these procedures is shown below (Figure 3−2).

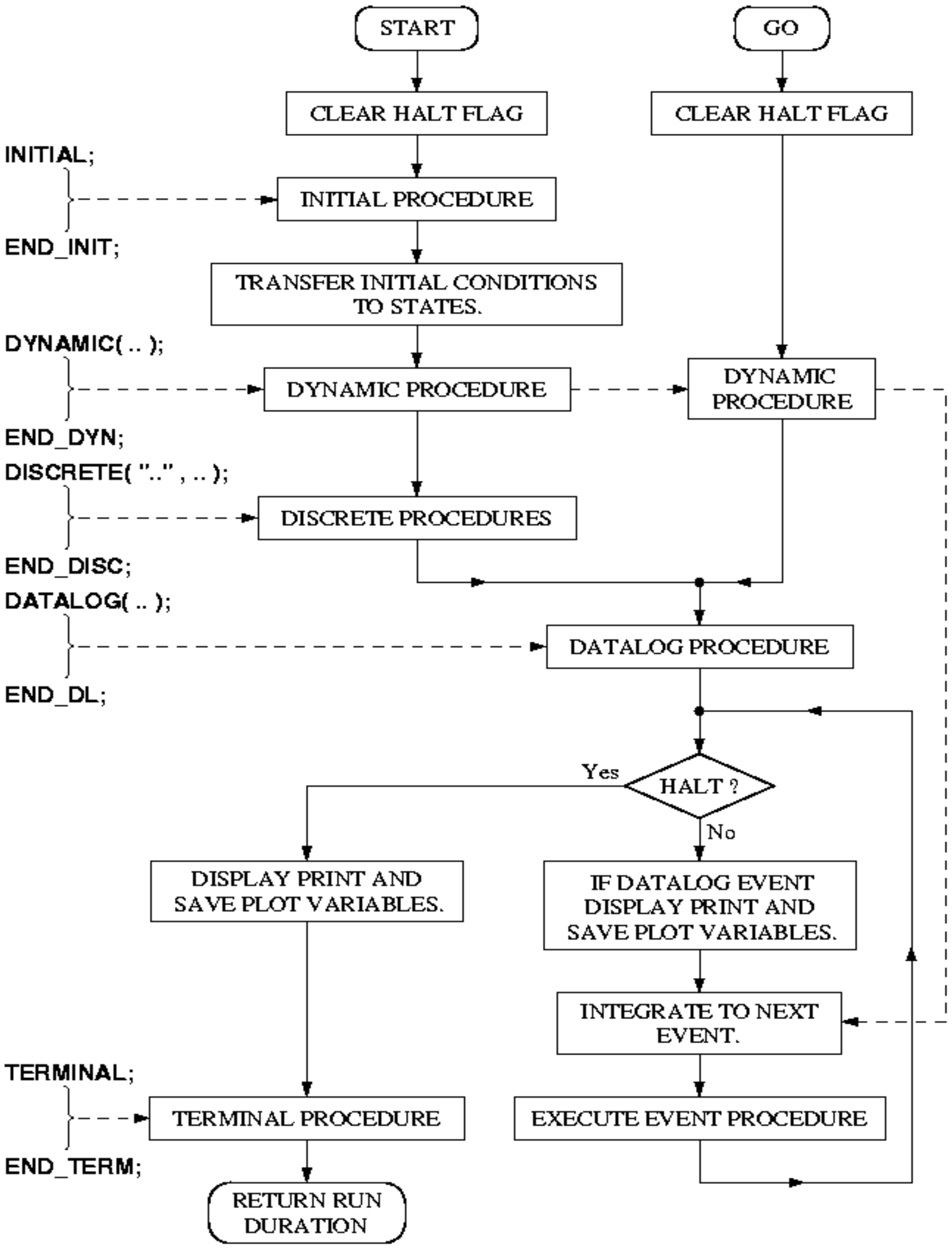


Figure 3−2

The **START** command clears the **HALT** flag then executes the code contained in the INITIAL section. At the end of this section the initial conditions are transferred to the states. Any states that do not have an initial condition defined are set to the scalar 0.

The DYNAMIC, DISCRETE and DATALOG sections are then executed in the order in which they appear in the model code. This ensures that all the variables are initialized ready for the first data logging and that all DISCRETE sections are properly **SCHEDULE**d. (The DYNAMIC and DATALOG sections are then executed again to update the model's variables to reflect any changes made by the DISCRETE sections)

The **`HALT`** flag is then checked to see if it is set.

If the **`HALT`** flag is not set, the plot variables are saved and the print variables displayed. Then the integration routine is called to advance the states to the next event time. This event could be either a DATALOG event, a DISCRETE event, a user event or the stoptime event. The event procedure is then executed and the **`HALT`** flag tested.

If the **`HALT`** flag is set (or the user has interrupted the run by pressing the **H** key) the final set of print variables are displayed and the plot variables saved before executing the TERMINAL section of the model. The **START** procedure returns a string containing the run time.

The **GO** command continues the run from where it was last halted. If the **stoptime** has not been changed the run will terminate without advancing the states. The **GO** procedure also returns a string containing the run time.

3.3.SimGauss Naming Conventions

SimGauss reserves several classes of identifier and file names for special uses.

3.3.1 Files

SimGauss uses three classes of file name. These are shown in Table 3−1. Only files with the extension **.sgm** are written by the user. The other two file types are generated automatically by the translator.

Table 3−1
SimGauss File Types

model.**sgm**	Files containing the user's model code.
model.**sgs**	Files generated by the SimGauss translator to define the model's states.
model.**sgp**	Files generated by the SimGauss translator to define the procedures needed to run the model.

3.3.2 Identifiers

SimGauss identifier classes are denoted by a prefix consisting of a letter(s) and an underscore. For example variables starting with **`d_`** denote the derivatives of states. **`d_x`** is the derivative of the state **`x`**. Table 3−2 lists the classes of identifier used by SimGauss.

Table 3−2
SimGauss Identifiers

`d`_*state*	The derivative of state
`i`_*state*	The initial condition of state
`m`_*state*	The maximum integration error for state
`r`_*state*	The relative integration error for state
`m`_*ctrl*	The absolute perturbation for control
`r`_*ctrl*	The relative perturbation for control
`SGP`_*name*	SimGauss model sections as procedures
`S`_*name*	SimGauss system variables and procedures

In keeping with GAUSS syntax there is no distinction between upper and lower case identifiers. So **`D_x`** and **`d_X`** are both the same variable. No user defined variables should start with **`'S_'`** or **`'s_'`** as these identifiers are reserved for SimGauss internal variables and procedures.

The states of a SimGauss model are determined from the statements which assign values to the derivative variables. These are called derivative statements. Derivative statements are only allowed in the DYNAMIC section of the model.

For example

```
d_x = v;
```

identifies **x** as being a state of the model.

A consequence of this naming arrangement is that state names can have at most 6 characters since SimGauss limits variable names to 8 characters. For a similar reason control variables are also limited to 6 characters.

At present the translator does not recognise derivative variables enclosed in ( ) or { }. The following valid GAUSS statements

```
( d_x1 ) = v1;
{ d_x2 } = velocity(v2);
```

will not add **x1** or **x2** to the list of model's states and so SimGauss will not integrate the states **x1** or **x2**.

3.3.3 Strings

Strings in SimGauss model files must have a closing pair of double quotes on the same line as the opening pair. Long strings can be continued on the following line by using "\ at the end of the first line of the string. Carriage returns and line feeds can be inserted into strings using the special character pairs \r and \n inside the double quotes.

3.3.4 Characters

The SimGauss translator forces all the model code characters into normal ASCII values. This means that any graphic characters appearing in the model are converted to normal text characters when they are copied to the **.sgp** file. To insert graphic characters into a string use the \ddd GAUSS codes as explained in the GAUSS manual.

3.3.5 Comments

SimGauss only allows comments delimited by /* and */. Any @'s found in the model code outside a string or a comment, delimited by /* */, will be flagged as an error.

Chapter 4

A Full Model

This chapter presents an example that includes all the possible model sections. It also illustrates the difference between a repetitive DISCRETE section and one that needs to be **SCHEDULE**d.

4.1 Digitally Controlled Spring System

This model is an extension of the Spring System of Chapter 2, A Simple Model. The system damping, R, is now controlled by a digital controller. The velocity, v, of the mass is sampled every 0.1 seconds and the absolute value used to increase the minimum value of system damping. In practice the sampling of the velocity and calculation of the new value of damping takes some time to complete. To model this sampling and calculation time, the output from the digital controller is delayed by 10mS (Figure 4−1). Initially the mass is released 1 meter above its steady state position.

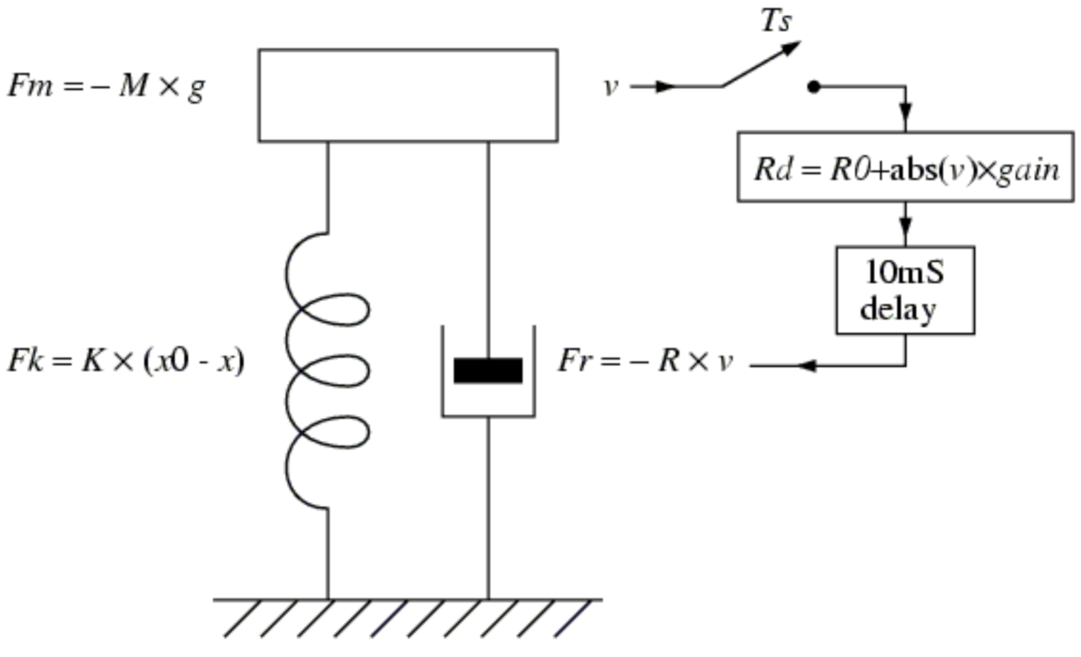


Figure 4−1

The system equations are :

$$M \times \frac{dv}{dt} = \Delta F = -M \times g + K \times (x0 - x) - R \times v$$

$$v = \frac{dx}{dt}$$

Every 0.1 seconds the velocity is sampled and Rd calculated from

$$Rd = R0 + abs(v) \times gain$$

10ms later the new value of damping is transferred to the system damping parameter, R, via

$$R = Rd$$

where

K	is the spring constant (Kg/s2)
M	is the mass (Kg)
R	is the damping co−efficient (Kg/s)
Rd	is damping co−efficient calculated by the digital controller
R0	is the minimum system damping co−efficient (Kg/s)
gain	is the gain of the digital controller
x0	is the length of the spring at rest (m)
g	is the gravitational acceleration constant (m/s2)
v	is the velocity of the mass (m/s)
x	is the position of the mass (m)

4.1.1 Model Code

```
/* ctrlsprg.sgm   A mass/spring/damper system
** with digital control of the damping. */

/* Set up Gauss environment */
library simgauss, pgraph;
graphset;
external proc plotxiri;
Ts = 0.1; /* sampling period */
calcT = 0.01; /* 10 mS calculation delay */
stoptime = 4.0; /* stop at 4 sec */
```

```

INITIAL;
DECLARE matrix R0 ! = 1; /* min. damping Kg/(sec) */
R = R0; /* initial value for damping */

DECLARE matrix M ! = 1; /* Kg mass */
DECLARE matrix K ! = 9.8; /* Kg/(s*s) spring const. */
DECLARE matrix x0 ! = 5; /* length of spring at rest */
DECLARE matrix x1 ! = 1; /* initial displacement */
DECLARE matrix g ! = 9.8; /* grav. accel. m/(s*s) */
i_x = x0−g./k + x1; /* steady state position + x1 */

output file = ctrlsprg.out reset; /* open output */
END_INIT;

DYNAMIC(0.1);
/* System equations */
d_v = (K.*(x0−x) − R.*v)./M − g;
d_x = v;
END_DYN;

DISCRETE("sample",Ts); /* sample every Ts sec */
DECLARE matrix gain ! = 4; /* controller gain */
Rd = R0+abs(v).*gain; /* damping varies with v */
SCHEDULE("control",T+calcT); /* SCHEDULE output */
END_DISC;

DISCRETE("control",0); /* Must be scheduled */
logdata(0); /* no print output */
R = Rd; /* transfer new control to system */
logdata(0); /* no print output */
END_DISC;

DATALOG(0.1);
Vi = V*3.2808; /* convert to imperial feet/sec */
Xi = X*3.2808; /* and feet */
Ri = R*0.4536; /* and pounds/(sec) */
END_DL;

TERMINAL;
output off; /* close output file */
END_TERM;

/* Gauss procedures */
Proc(0) = plotxiri;
_ptitlht = 0.3;
_paxht = 0.3;
title("Mass/Spring/Damper with Digital Damping "\
"Control");
begwind;
window(2,1,0);
sgxy "T,Xi";
```

```
63. title("");
64. nextwind;
65. sgxy "T,Ri";
66. endwind;
67. endp;
```

♦ Gauss Code

Lines 1 to 11 are the Gauss Code section of the model. They set up the GAUSS environment in which the model is to run. Line 5 sets the libraries needed for this model. Line 6 initializes the graphics module and 7 defines plotxiri as a procedure. Lines 8 and 9 set the sampling period and the calculation delay. Line 10 sets the initial stop time for the model. These variables could also have been set using DECLARE statements.

Refer to the section on Customizing SimGauss in the Introduction for instructions on how you can include the plotting routines in your SimGauss system so that they do not need to be compiled each time SimGauss is loaded.

♦ INITIAL Section

Lines 12 to 24 are the INITIAL section of the model. This code ensures that at the START of each run the initial value of the damping is set to R0 and that the initial condition for the position of the mass, **`i_x`**, is 1 meter above the steady state position. Performing this calculation here allows the values of the spring constant, K and the length of the spring, **`x0`**, to be changed from the GAUSS command mode and the simulation re−run with the same initial displacement from the steady state position. Line 23 resets the output file to capture the print variables.

Any state for which an initial condition variable is not defined is assigned an initial scalar value of 0. This is the case for the state **`v`** as no **`i_v`** is defined. Initial conditions need not be defined in the model code but can be defined from the GAUSS command mode.

♦ DYNAMIC Section

The DYNAMIC section of the model is lines 26 to 30. This contains the same system equations as were used in Chapter 2.

♦ DISCRETE Sections

The first DISCRETE section is defined in lines 32 to 36. This DISCRETE section is called **`sample`** and is executed repetitively every **`Ts`** seconds. This section implements the digital controller. It calculates the new damping co−efficient (line 34) and then **SCHEDULE**s the **`control`** DISCRETE section to transfer the output to the continuous system calcT seconds later (line 35). **SCHEDULE** statements should only appear in DISCRETE sections or from user events (see Chapter 7).

The second DISCRETE section is called **`control`** and runs from line 38 to 42. As it has a zero time interval specified (line 38) it will not execute unless **SCHEDULE**d by another DISCRETE section.

The **LOGDATA** statements on lines 39 and 41 force the DATALOG section to be executed and the plot variables to be saved. This gives a clean step in the plot of **`R`**, regardless of when **`R`** is updated. If the argument to the **LOGDATA** procedure is true (non−zero) then the print variables are also displayed at the same time. Line 40 transfers the new damping co−efficient to the continuous model.

When the **START** command is used to run the model the INITIAL, DYNAMIC, all DISCRETE sections and the DATALOG section are executed in the order in which they appear in the model. This ensures that all the variables are initialized ready for the first data logging and that all the DISCRETE sections are properly **SCHEDULE**d. In this case the DISCRETE section, **`sample`**, automatically re−**SCHEDULE**s itself to execute again **`Ts`** seconds after the **START**. It also **SCHEDULE**s the **`control`** DISCRETE section to execute **`calcT`** seconds after the **START**.

♦ DATALOG Section

Lines 44 to 48 are the DATALOG section of the model. The **DATALOG** statement on line 44 sets the data logging interval to 0.1 seconds. In this model the DATALOG section is used to convert the position, velocity and damping from metric to imperial units for printing and plotting.

♦ TERMINAL Section

The TERMINAL section is lines 50 to 52. It turns off the output file at the end of the run.

♦ Gauss Procedures

Lines 53 to 67 form the Gauss Procedures section of the model. In this model only one procedure is defined which will plot **`Xi`** and **`Ri`** on the same page with an appropriate title.

4.1.2 Results

Compile the model using

```
simgauss examples\ctrlsprg
```

Before using **START** to run the model, the required print and plot variables need to be specified. For this model the position and damping in imperial units will be printed and saved for plotting.

Type

```
prtvars xi ri;
```

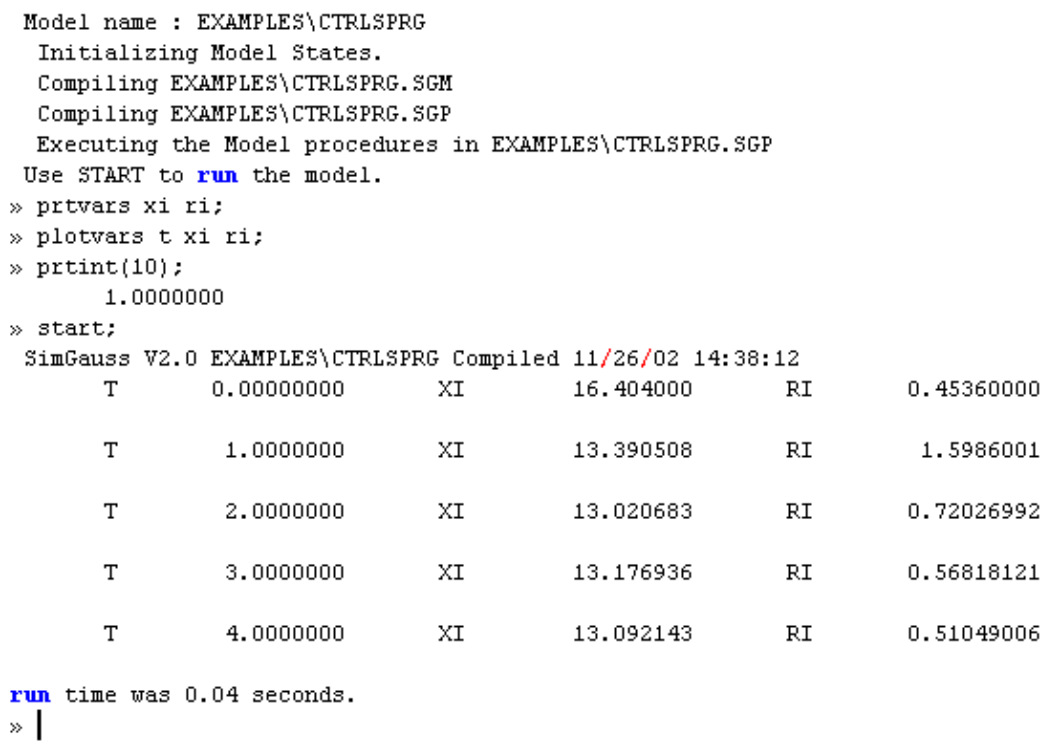
to set the print variables and

```
plotvars t xi ri;
```

to set the plot variables.

```
prtint(10);
```

is used to set the print interval to 10 times the data logging interval (once every second). After running the model using **START** your screen should look like Figure 4−2.



```
 Model name : EXAMPLES\CTRLSPRG
  Initializing Model States.
  Compiling EXAMPLES\CTRLSPRG.SGM
  Compiling EXAMPLES\CTRLSPRG.SGP
  Executing the Model procedures in EXAMPLES\CTRLSPRG.SGP
 Use START to run the model.
» prtvars xi ri;
» plotvars t xi ri;
» prtint(10);
      1.0000000
» start;
 SimGauss V2.0 EXAMPLES\CTRLSPRG Compiled 11/26/02 14:38:12
      T      0.00000000      XI       16.404000      RI       0.45360000

      T       1.0000000      XI       13.390508      RI        1.5986001

      T       2.0000000      XI       13.020683      RI       0.72026992

      T       3.0000000      XI       13.176936      RI       0.56818121

      T       4.0000000      XI       13.092143      RI       0.51049006

run time was 0.04 seconds.
» |
```

Figure 4−2

The plotting procedure defined in the model can now be used to plot the results.

Type

```
plotxiri;
```

to obtain the plot shown in Figure 4−3.

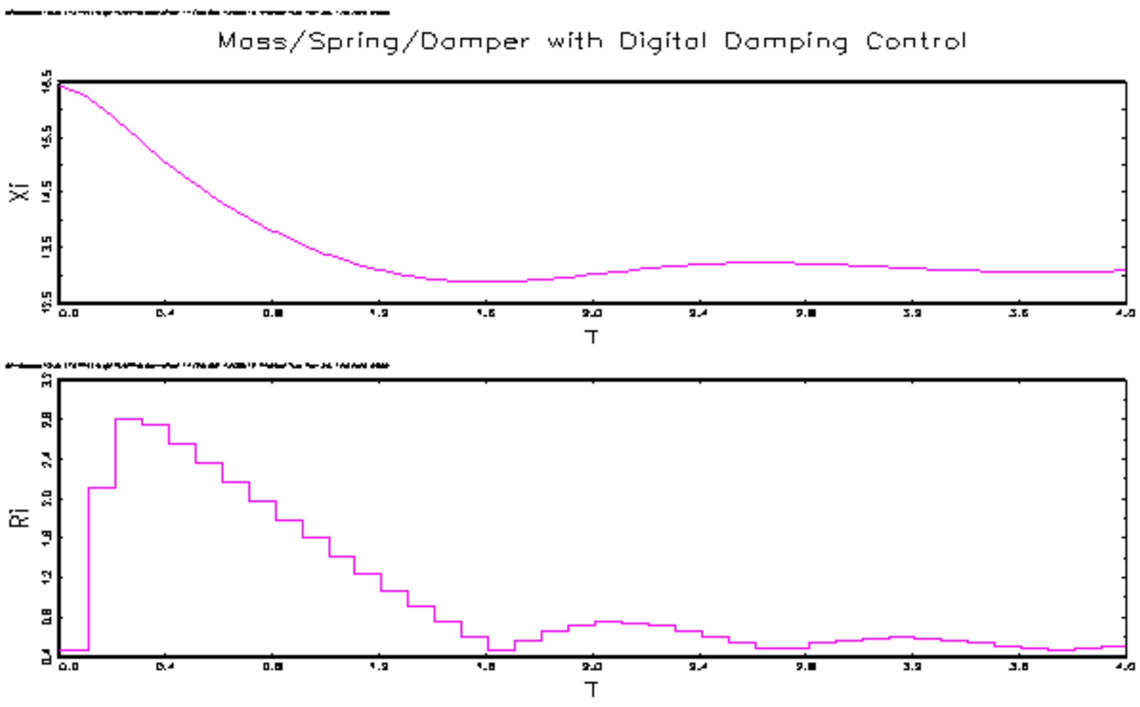


Figure 4−3

Chapter 5

State Vectors

One of the advantages of SimGauss over other simulation languages is the ease with which state vectors can be defined and used. In keeping with the matrix capabilities of GAUSS, SimGauss can integrate state vectors as easily as scalars. These state vectors can be generated in two ways, either by replacing the scalar derivative equation with one that uses vectors or replacing a group of scalar derivative equations with a single matrix equation.

This chapter discusses both these types of vectors, parameter vectors, and state vectors arising from state space systems. The use of vectors both speeds up the simulation and simplifies modelling.

5.1 Parameter Vectors

Parameter vectors allow fast evaluation of the effects of varying one or more model parameters. The digitally controlled spring model of Chapter 4, A Full Model, will be used to demonstrate the use of parameter vectors. An obvious parameter of interest in this system is the gain of the digital controller. Using vectors, various values of gain can be efficiently investigated in one run. In order to integrate the various gain models in one run the derivative statements must be able to return a vector of state derivatives corresponding to the vector of gains. SimGauss allows you to specify the dimensions of the various parameters of the model after it has been compiled, provided the model statements can handle the vectors correctly.

In general a scalar model can usually be converted to a vector model by replacing all the occurrences of / by ./ and * by .* However care must be taken with statements that use relational operators, such as **IF** statements, and with procedures that return or expect scalars. These types of statements are most simply handled by enclosing them in a **DO** loop which processes the elements of the vector one at a time. The model code for the digitally controlled spring, Section 4.1.1, uses element by element operators and has no relational operators.

The effect of five gains ranging from 0 to 10 will be investigated including the original scalar gain of 4. Having compiled the **ctrlsprg.sgm** model, the **`gain`** needs to be redefined as a vector and the dimensions of the model's states changed accordingly. SimGauss determines the dimensions of the states automatically from the dimensions of the initial conditions, so the initial conditions of the position, **`x`**, and the velocity, **`v`**, must be redefined as column vectors. SimGauss only allows scalars or column vectors for states.

Since the position's initial condition, **`i_x`**, is calculated in the INITIAL section of the model from

```
i_x = x0 − g./k + x1;
```

it cannot be redefined directly.

However by redefining **`x1`** as a column vector, **`i_x`** is forced to a column vector also. Equivalently either **`x0`**, **`g`** or **`k`** could have been redefined as column vectors.

To redefine the **`gain`** and the initial conditions type

```
gain = {0, 0.4, 1.5, 4, 10};
i_v = reshape(0,5,1);
x1 = reshape(x1,5,1);
```

The print and plot variables and the printing interval need to be set to the same ones used in Chapter 4, although **`xi`** and **`ri`** are now vectors.

```
prtvars xi ri;
plotvars t xi ri;
prtint(10);
```

Before SimGauss can run the model with these new dimensions, it has to regenerate and recompile the procedure files which depend on the dimensions of the model's variables. These are: the procedures that transfer the states to the model's variables and the printing and plotting procedures.

Each time the **START** command is used to run the model SimGauss checks the dimensions of the initial conditions after executing the model's INITIAL section. If any of the dimensions have changed since the last **START**, then SimGauss automatically recompiles the state transfer procedures and executes **START** again. If the state dimensions are correct SimGauss then executes the DYNAMIC section, all the DISCRETE sections and the DATALOG section to initialize all the variables for the first data logging operation.

SimGauss then checks the dimensions of first the print, and then the plot variables, against their old dimensions. If any of the dimensions have changed SimGauss recompiles the affected procedure and executes **START** again.

When **GO** is used to continue a run it also executes the DYNAMIC and the DATALOG sections before checking the dimensions of the print and plot variables and recompiling if necessary before executing **GO** again.

Run the vector model using **START** and compare the run time to that obtained for the scalar case. On this machine the vector run took about 0.04 seconds whereas five scalar runs would have taken about 4 seconds. The use of parameter vectors has produced the results approximately 4 times faster than the equivalent five scalar runs.

Figure 5−1 shows the response of **`xi`** for the five gains. It was generated by the following commands.

```
library simgauss, pgraph;
title("Digitally Controlled Mass/Spring/Damper System "\
"for 5 Gains");
_plegstr = " 0.0\000 0.4\000 1.5\000 4.0\000 10 ";
_plegctl = {1,5,3,14};
sgxy "T,Xi";
```

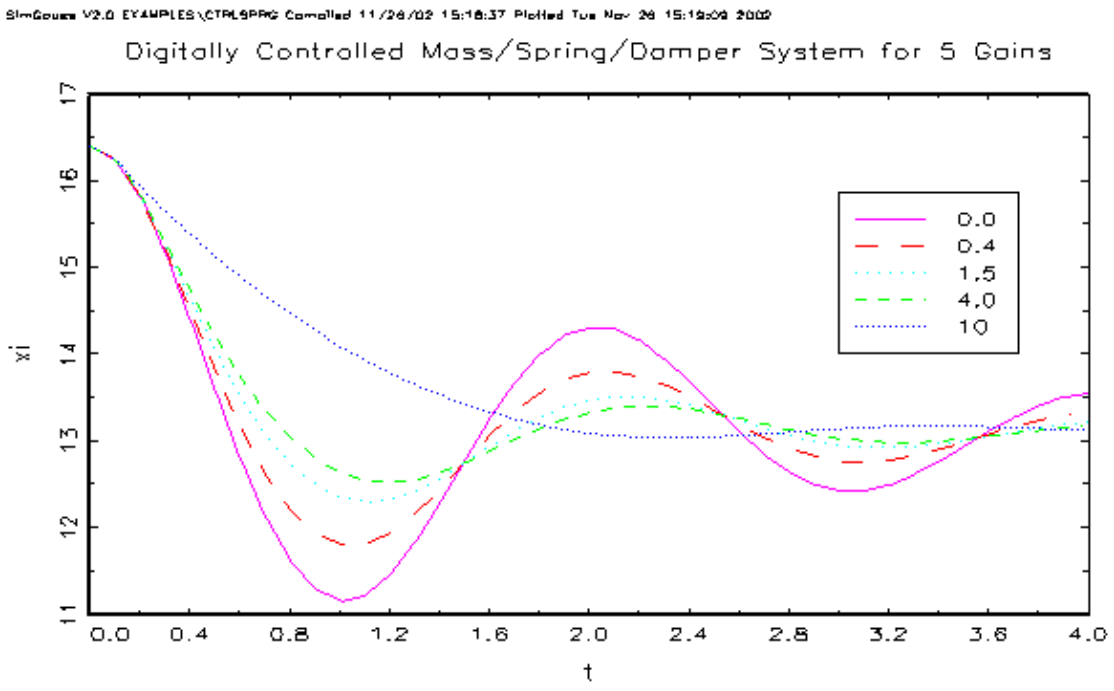


Figure 5−1

5.2 State Space Systems

Expressing a system of differential equations as a matrix differential equation is often very convenient and since GAUSS is optimized for matrix calculations it is also usually more efficient, unless the matrices have a lot of zero elements.

A continuous, linear, system of differential equations can always be expressed as

$$\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

where	*x*	is a vector of states
	u	is a vector of inputs
	y	is a vector of outputs and
	A, **B, C** and **D**	are matrices of the appropriate sizes

This state space representation of the system is discussed in the standard text books on control theory such as Takahashi et al. [1]. The SimGauss command **SGLIN** can be used to find the linearized A,B,C and D matrices of non−linear systems at the current state.

The simulation of state space systems will be illustrated using the paper−machine head box example described by Franklin and Powell [2]. In this example the inputs, **u**, are the air control and the stock control. The outputs, **y**, are the total head and the liquid level. The states, **x**, are the total head, the liquid level and the change in air valve opening.

Digital control of the system is obtained by sampling the states every 0.2 seconds and feeding them back to the inputs, via a gain matrix **K** (state feedback). For further details on the plant and the design of the feedback matrix refer to Franklin and Powell's book [2] which is an excellent introduction to digital control methods.

The code for the model is contained in the file STATEMAT.SGM

```
/* statemat.sgm  A state matrix simulation
** with digital state feedback control. */

/* Set up Gauss environment */
library simgauss, pgraph;
graphset;
_plctrl = −1; /* symbols only */
_plegstr = "total head\000liquid level";
_plegctl = {1,5,1.8,0.3};
Ts = 0.2; /* sampling period */
calcT = 0; /* no calculation delay */
stoptime = 3.0; /* stop at 3 sec. */
datastep(0.2); /* set data logging every 0.2 sec. */

/* Set up the state matrices and state gain matrix */
A = {−0.2 0.1 1, −0.05 0 0, 0 0 −1};
B = {0 1, 0 0.7, 1 0};
C = {1 0 0, 0 1 0};
D = {0 0, 0 0};
K = {5.85 −8.41 3.55, 0.986 5.22 −0.24};
```

```
21. i_x = {1, 0, 0}; /* initial state */
22. Iu = {0, 0}; /* initial control */
23.
24. INITIAL;
25. u = Iu; /* initialize inputs for time zero */
26. END_INIT;
27.
28. DYNAMIC(0.1);
29. /* System State Equations */
30. d_x = A*x + B*u;
31. y = C*x + D*u;
32. END_DYN;
33.
34. DISCRETE("sample",Ts); /* sample every Ts sec */
35. /* State Feedback */
36. ud = −K*x;
37. SCHEDULE("control",T+calcT); /* SCHEDULE output */
38. END_DISC;
39.
40. DISCRETE("control",0); /* Must be scheduled */
41. logdata(0); /* no print output */
42. u = ud; /* transfer new control to system */
43. logdata(0); /* no print output */
44. END_DISC;
```

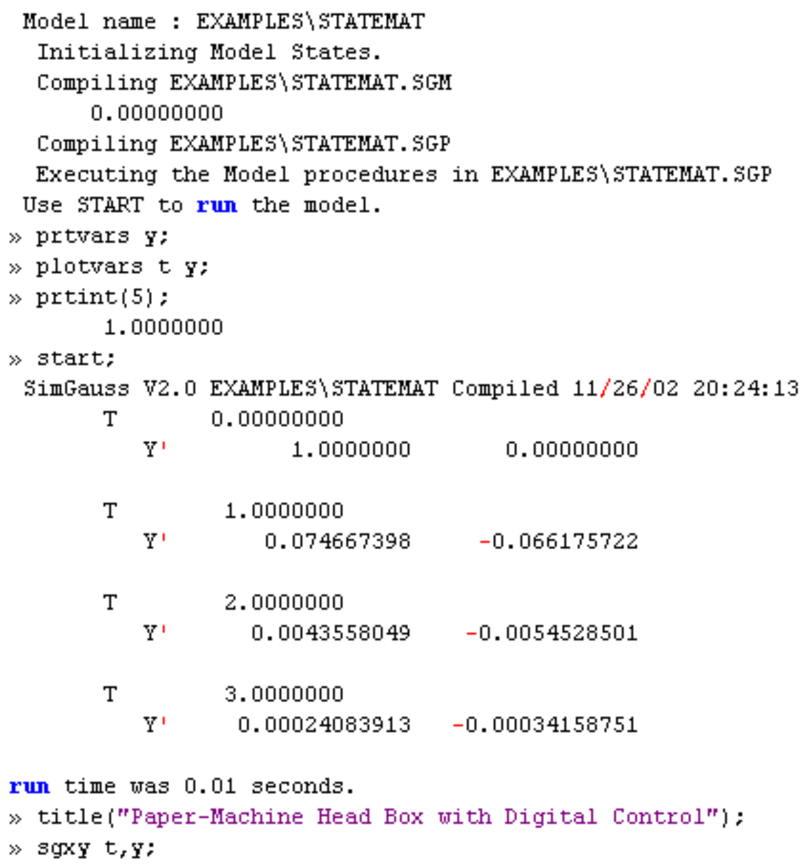
The GAUSS environment and the required matrices and vectors are defined in lines 5 to 22. Line 21 sets the initial state of the system and informs SimGauss of the dimension of the state vector. Since there is no DATALOG section in this model, line 13 sets the data logging interval to 0.2 seconds.

The state space equations are defined on lines 30 and 31, while the digital sampling is carried out every **Ts** seconds by the DISCRETE section **`sample`** (lines 34 to 38). Line 36 calculates the new control based on the sampled values of the states. This is then transferred to the plant by the **`control`** DISCRETE section (lines 40 to 44), after a calculation delay. Initially the calculation delay has been set to zero (line 11).

The model is run using the commands

```
» simgauss examples\statemat;
» prtvars y;
» plotvars t y;
» prtint(10);
» start;
```

Figure 5−2 shows the output generated from running this model.



```
 Model name : EXAMPLES\STATEMAT
  Initializing Model States.
  Compiling EXAMPLES\STATEMAT.SGM
      0.00000000
  Compiling EXAMPLES\STATEMAT.SGP
  Executing the Model procedures in EXAMPLES\STATEMAT.SGP
 Use START to run the model.
» prtvars y;
» plotvars t y;
» prtint(5);
       1.0000000
» start;
 SimGauss V2.0 EXAMPLES\STATEMAT Compiled 11/26/02 20:24:13
       T       0.00000000
          Y'         1.0000000        0.00000000

       T        1.0000000
          Y'       0.074667398     -0.066175722

       T        2.0000000
          Y'      0.0043558049    -0.0054528501

       T        3.0000000
          Y'     0.00024083913   -0.00034158751

run time was 0.01 seconds.
» title("Paper-Machine Head Box with Digital Control");
» sgxy t,y;
```

Figure 5−2

Figure 5−3 is a plot of the system output, y, sampled at 0.2 second intervals generated using the commands

```
» title("Paper−Machine Head Box with Digital Control");
» sgxy t,y;
```

The matrices and initial condition vectors used in this model are defined in the GAUSS section of the model, rather than in the INITIAL section. This allows you to change them from the GAUSS command mode, after the model has been compiled, to simulate other state space systems. For example setting **K** to a matrix of zeros will give the open loop response.

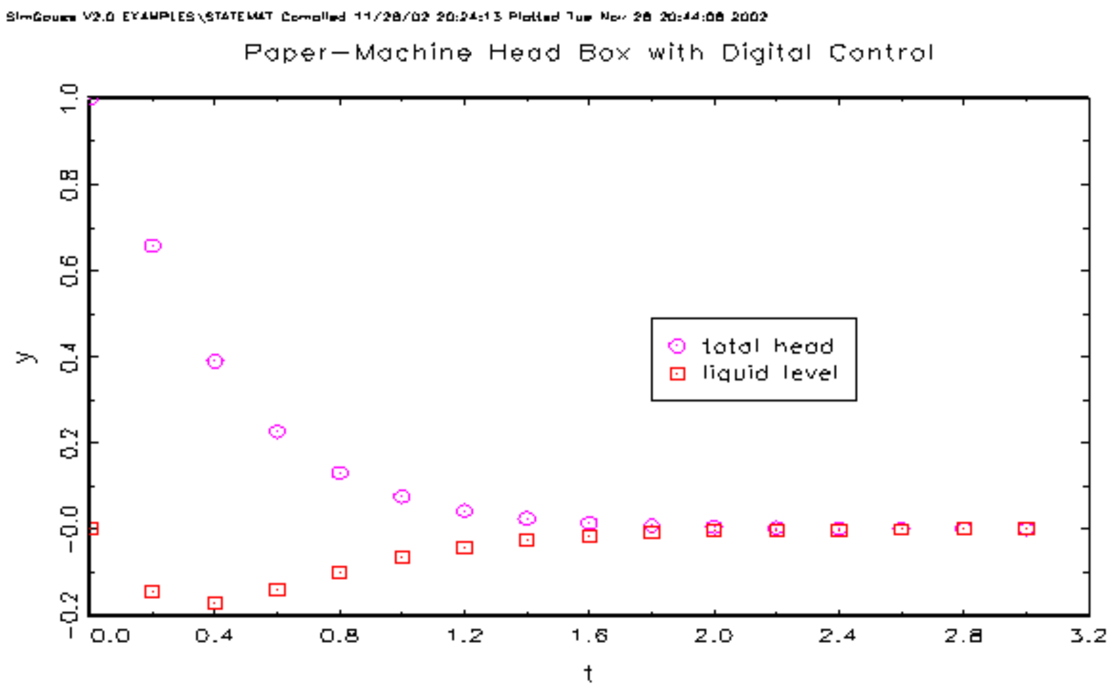


Figure 5−3

To change the control from digital to analogue, edit the model code to remove the DISCRETE sections and insert

```
u = −K*x;
```

after line 31 in the DYNAMIC section.

Another change that you may want to make is to replace the simple digital controller with a more general one using difference equations. This can be conveniently done using the discrete state space form of the difference equations. The file **discmat.sgm** contains the model code incorporating the necessary changes.

References

[1] Takahashi, Y.M., Rabins, J. and Auslander, D.M., Control and Dynamic Systems, Addison−Wesley, Mass. 1970.

[2] Franklin, G.F. and Powell, J.D., Digital Control of Dynamic Systems, Addison−Wesley, Mass. 1980, pp. 265

Chapter 6

Integration Algorithms

SimGauss provides eight integration algorithms. These are

Euler's method
2nd order Runge−Kutta
4th order Runge−Kutta
2nd/3rd order Runge−Kutta−Fehlberg
4th/5th order Runge−Kutta−Fehlberg
Richardson−Bulirsch−Stoer extrapolation method
Adams−Moulton predictor−corrector method
Gear's stiff method

The first three are fixed step size algorithms while the last five are variable step size algorithms. Initially SimGauss' integration algorithm is set to 4th order Runge−Kutta, not because this is always the best, but because it is the one most people are familiar with. This chapter will discuss the SimGauss integration process and the integration algorithms. The use and advantages of the variable step size algorithms will be illustrated by an example.

6.1 The Integration Procedure

SimGauss is event driven. This means that the integration proceeds from event to event and there is no mechanism in SimGauss for stopping the simulation between events. SimGauss maintains an ordered event queue. After each event, SimGauss compares the next data logging time and the **stoptime** to the first event on the queue and determines which is earlier. Data logging intervals and the **stoptime** are considered to be events but are not put on the event queue. It then calls the integration procedure to advance the model states to the next event time. If more then one of these events occurs at the same time the **stoptime** takes precedence followed by the data logging event then the event on the event queue.

After each event the SimGauss integration procedure first collects the current states and calls the DYNAMIC section of the model to evaluate the initial derivatives in case any of the states or variables have been changed by the last event. It then takes an integration step. At the end of the step the DYNAMIC section of the model is called again to update the variables and derivatives to the new state in preparation for the next integration step or the next event. If the next event time has not been reached further integration steps are taken until the next event time is reached.

For each integration step, one of SimGauss' integration algorithms is used to advance the states. The integration algorithm is selected by the **INTALG** keyword. The **INTALG** keyword can be use at any time during the simulation and will take effect from the next integration step even if the next event has not yet been reached.

6.2 Fixed Step Size Algorithms

The three fixed step size algorithms are: Euler's method, 2nd order Runge−Kutta and 4th order Runge−Kutta. When a fixed step size algorithm is selected using **INTALG**, SimGauss sets the integration step size to the initial step size. The initial step size is set by the argument to the **DYNAMIC** statement in the model code. It can be changed at any time using the **INITSTP** command. When the simulation is run the size of each integration step is determined by the minimum of, the time to the next event and the current step size.

For fixed step size algorithms, the integration step will stay at the initial step size unless the time to the next event is less than the initial step size. In that case a small step is taken to bring the time up to the next event time. This means that if **DATASTEP** is less than the initial step size then the time between data logging events sets the upper limit on the integration step size.

6.2.1 Euler's Method

Euler's method is not recommended for any real application but is included for pedagogical purposes and for error checking. It is a first order algorithm requiring one derivative evaluation per step. That is one call to the DYNAMIC section procedure, **SGP_DYN**, per step. Refer to any good numerical analysis text book such as Conte and de Boor [1] for further details.

6.2.2 Runge−Kutta Methods

The 2nd and 4th order Runge−Kutta algorithms make 2 and 4 derivative calculations across the interval and are 2nd and 4th order accurate respectively. A weighted combination of these derivatives is then used to calculate the new states. (See Algorithms 6.2 and 6.3 of Conte and de Boor's text [1].)

For the same accuracy the Runge−Kutta 2nd order algorithm requires a smaller step size than Runge−Kutta 4th order but for the same step size should run faster for non−trivial models. For models with **TABLE** lookups or hard non−linearities, such as **BACKLASH** and **DEADBAND**, Runge−Kutta 2nd order is recommended. This is because the higher accuracy of the 4th order method depends on the smoothness of the higher order derivatives.

The main problem with these fixed step size Runge−Kutta methods is that no error estimate is available. To check the errors in a practical simulation halve the step size, using **INITSTP**, and see how much the results change. The difference between results can then be found by using

SGPLTVAR to extract the required outputs after each run and then subtracting them.

When doing this check, the **DATASTEP** should be a negative integer so that data logging occurs a multiples of the integration step size and does not force small steps to be taken. The value of **INITSTP** also needs to be small enough to give reasonably accurate results for the system concerned.

Variable step size algorithms overcome this difficulty by calculating an estimate of the state error automatically and adjusting their step size accordingly.

6.3 Variable Step Size Algorithms

The variable step size algorithms available in SimGauss are the 2nd/3rd order Runge−Kutta−Fehlberg, the 4th/5th order Runge−Kutta−Fehlberg, the Richardson−Bulirsch−Stoer extrapolation method, the Adams−Moulton predictor−corrector method and Gear's stiff method. All these algorithms estimate the local truncation error at each integration step and adjust the step size to keep the local error less than the specified error.

It is important to remember that what is being controlled is the local truncation error at each step. The error in the states at the end of the run consists of both the local truncation error for the last step plus the cumulative effect of the truncation errors of all the previous steps.

The efficiency of all these variable step size methods relies on being able to take large steps over smooth areas of the simulation. Since the maximum step that the integration algorithm can take is limited by the time to the next event, if you wish to datalog at the finest intervals without limiting the step size then **DATASTEP(−1)** should be used to datalog at every integration step.

6.3.1 Runge−Kutta−Fehlberg Methods

The 2nd/3rd order Runge−Kutta−Fehlberg algorithm requires 3 derivative evaluations and is 3rd order accurate while the 4th/5th order algorithm requires 6 derivative evaluations and is 5th order accurate. See Thomas [2] for further details on these methods and their error control.

The 2nd/3rd order algorithm is recommended for most physical simulations. The 4th/5th order algorithm is for use with models that have continuous higher order derivatives and fewer discontinuities, **DEADBAND**s, **BACKLASH**, **TABLE** lookups, etc.

6.3.2 Richardson−Bulirsch−Stoer Method

The Richardson−Bulirsch−Stoer method divides each integration step into a variable number of sub−steps. It then uses Richardson's extrapolation method to estimate the new values for the state for a sub−step approaching zero length. See Press et. al. [3] for a more detailed explanation. For smooth systems this method can take much larger steps than other methods.

6.3.3 Adams−Moulton and Gear's Stiff Method

Adams−Moulton and Gear's stiff method are both variable step size, variable order, predictor−corrector methods. See Gear [4] for details. The Adams−Moulton method is suitable for differential equations whose solution is well approximated by a polynomial of moderate order. For other cases Richardson−Bulirsch−Stoer is often better.

Gear's stiff method is the method of choice when the system contains widely varying time constants. It allows larger integration step sizes without going unstable. It uses a subset of the **SGLIN** code to form a linearized system at the current state. The perturbations of the states used to linearize the system are determined from the current *desired_error* as defined below.

6.3.4 Step Size Control

When a variable step size algorithm is selected using **INTALG**, SimGauss does not change the current integration step size. When the simulation is run the size of each integration step is determined by the minimum of, the time to the next event, the current step size and the maximum step size. The variable step size algorithm then attempts to take a step of that size. If the local error is less then the desired_error then the algorithm updates the states and returns the maximum of the new integration step size and the minimum step size. Otherwise the step fails and the algorithm automatically reduces the current step size and tries again. This continues until either the smaller step size is successful or the minimum step size is reached.

The *desired_error* is calculated from the specified **MERROR**s and **RERROR**s for each state. The *desired_error* for each state is given by

$$\textit{desired\_error} = \max(\, \text{m\_}\textit{state} \,,\, \text{r\_}\textit{state} * \max(\text{abs}(\textit{state}))\,)$$

where max(abs(*state*)) is the maximum of the absolute value of that state so far, not the current value.

In SimGauss both the maximum and the relative error can be specified separately for each state. The variables **m_**state specify the maximum errors while the variables **r_**state specify the relative errors. These variables should be either scalars or have the same dimension as the corresponding state. However you are not forced to specify the errors for each state separately. The SimGauss commands **MERROR** and **RERROR** can be used to globally set the maximum and relative errors for all states which do not have a respective individual error specified or whose specified error is zero. The initial value for **MERROR** and **RERROR** is 1.0E−4.

If any of these variable step size algorithms attempts to reduce the integration step size below the minimum step size set by **MINT** then the error flag **errmint** is set to 1 and the DYNAMIC section of the model is executed once. If the **errmint** flag is still non−zero on return from the DYNAMIC section the run is stopped with an error message. Testing the **errmint** flag allows you to call **MINT** or change the integration algorithm to get over a singularity or some other difficult point in the simulation.

6.4 A Modified State Space System

A modification of the state space model of Chapter 5, State Vectors, will be used here to demonstrate the advantages of variable step size algorithms. Compile the file **statemat.sgm** using SimGauss and select the same print and plot variables as before (Figure 5−2). Then change the A[3,3] element to −30 and run the model and plot the results (Figure 6−1) i.e.

```
prtvars y;
plotvars t y;
prtint(5);
a[3,3] = −30;
start;
title("Modified Paper−Machine Head Box using RK4");
sgxy t,y;
```

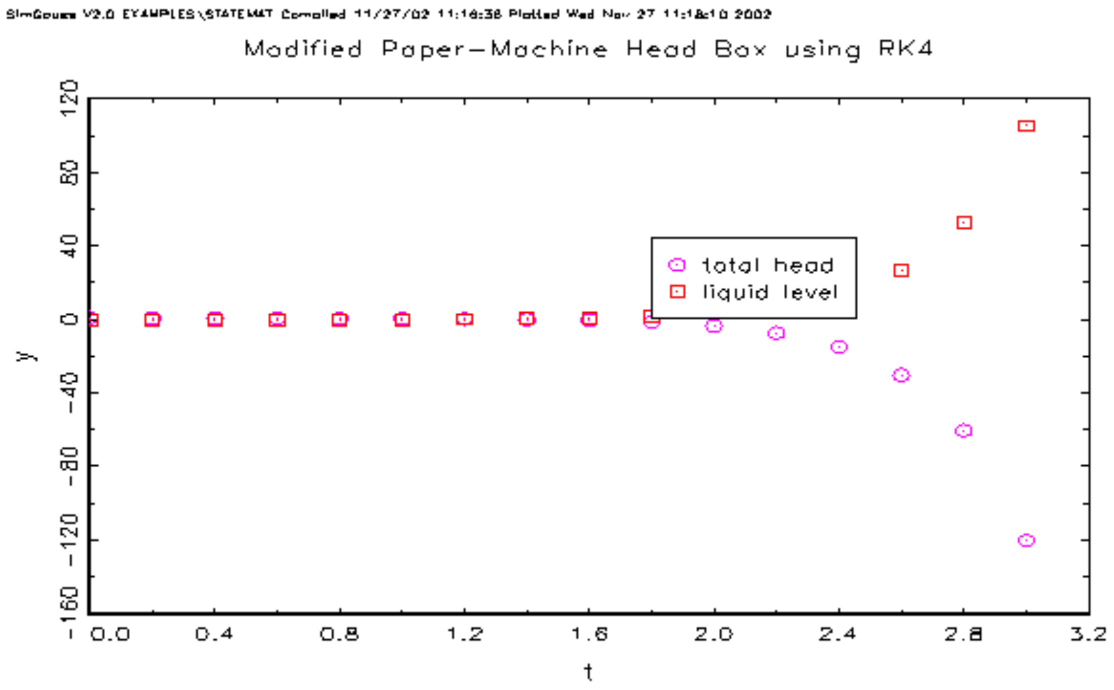


Figure 6−1

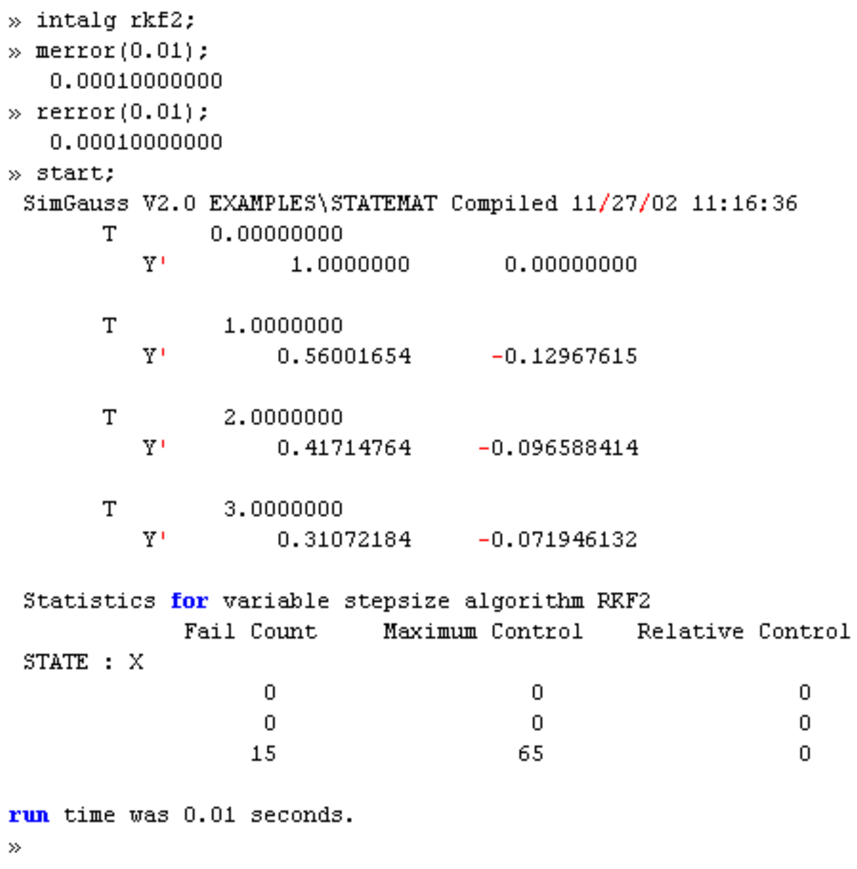
The integration method is unstable and if the simulation is run for a longer time an overflow error will occur. This is not due to an instability in the model but rather instability in the 4th order Runge−Kutta integration algorithm.

This type of instability can be removed by choosing a smaller integration step size using **INITSTP**, in this case approximately **INITSTP(0.03).** However this results in a longer run time. A better approach in this case is to use one of the variable step size algorithms.

Set the integration algorithm to 2nd/3rd order Runge−Kutta−Fehlberg and the maximum and relative errors to 1.0E−2. Then re−run the simulation.

```
intalg rkf2;
merror(0.01);
rerror(0.01);
start;
```

The results for this run are shown in Figure 6−2. Figure 6−3 shows the response of the output.



```
» intalg rkf2;
» merror(0.01);
   0.00010000000
» rerror(0.01);
   0.00010000000
» start;
 SimGauss V2.0 EXAMPLES\STATEMAT Compiled 11/27/02 11:16:36
       T       0.00000000
          Y'         1.0000000       0.00000000

       T        1.0000000
          Y'        0.56001654      -0.12967615

       T        2.0000000
          Y'        0.41714764     -0.096588414

       T        3.0000000
          Y'        0.31072184     -0.071946132

 Statistics for variable stepsize algorithm RKF2
             Fail Count     Maximum Control    Relative Control
 STATE : X
                   0                  0                   0
                   0                  0                   0
                  15                 65                   0

run time was 0.01 seconds.
»
```

Figure 6−2

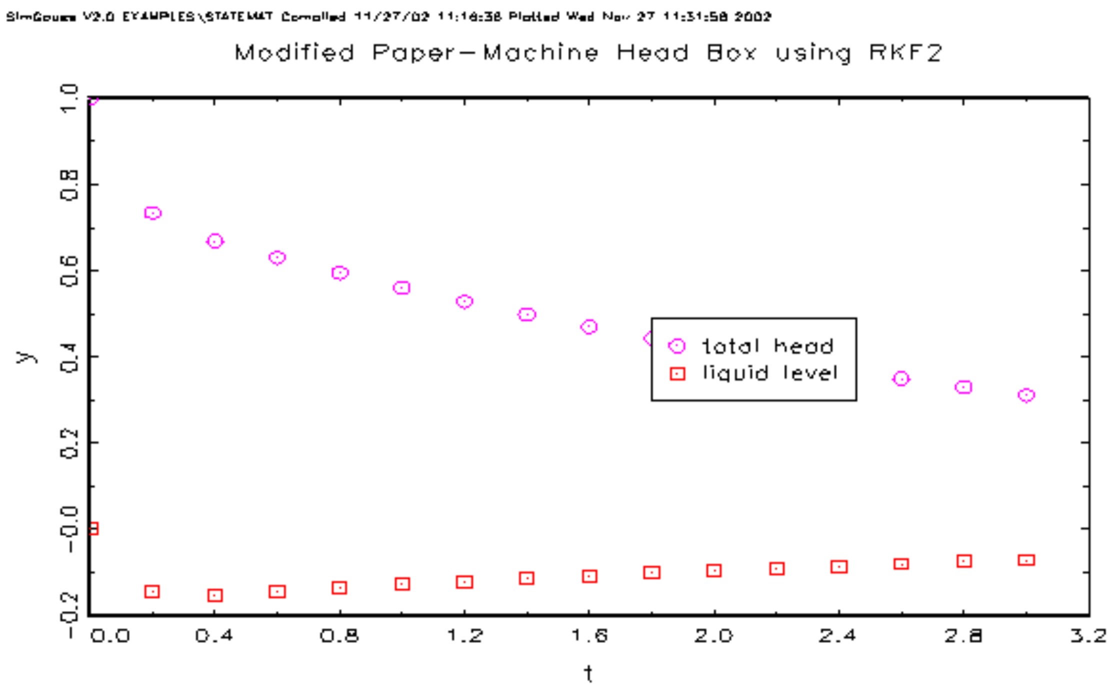


Figure 6−3

When a run terminates with a variable step size algorithm selected, statistics are printed out for each state. This output is suppressed if there are no print variables specified. For each state the number of steps that failed to meet the specified error and had to be retaken with a smaller step size is listed under the 'Fail Count'. Also printed are the number of steps for which the maximum and the relative error was the controlling error, 'Maximum Control' and 'Relative Control'.

For each step only the state with the largest error has its count incremented in either the 'Maximum Control' or the 'Relative Control' column depending on which error is controlling. If step had to be retaken the 'Fail Count' for that state is also incremented. The number of successful steps taken is the total number of steps in the 'Maximum Control' and the 'Relative Control' columns less the total of the 'Fail Count' column. In the case of Figure 6−2, there were 50 successful steps taken.

Since the maximum error must have units is it possible for the maximum error to swamp the relative error if state value never exceeds unity. A zero in the 'Relative Control' column indicates that the maximum error specification is dominating the error control for this state. This is not necessarily a defect but it is worth investigating.

In the above example the maximum error for the third element of the **x** state vector is dominating the error control but a plot of the third element shows its maximum value is about −0.2. Because this is less than 1 the maximum error always controls when it and the relative error have the same value. For this example a relative error specification of 0.055 is sufficient to make the relative error dominate.

References

[1] Conte, S.D. and de Boor, C., Elementary Numerical Analysis, 1972, McGraw−Hill.

[2] Thomas, B., The Runge−Kutta Methods, Byte, April 1986, pp. 191−210.

[3] Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T., Numerical Recipes, The Art of Scientific Computing, 1986, Cambridge University Press.

[4] Gear, C.W. "The Automatic Integration of Ordinary Differential Equations", Communications of the ACM, Vol. 14, No. 3, pp. 176−190, March 1971.

Chapter 7

User Events

In addition to the simulation event queue SimGauss also has a user event list. This is a very flexible and powerful mechanism for controlling the progress of the simulation. This chapter describes user events and illustrates how they can be used to introduce disturbances into the simulation.

7.1 User Event Keywords

There are two types of events in SimGauss. The first type is generated by the SimGauss translator from the model's DISCRETE sections. (Refer Chapters 3 and 4.) These DISCRETE event procedures can only be **SCHEDULE**d from within the simulation itself. The other type of event is the user event keyword.

The user event keyword, as the name suggests, is a keyword procedure written by the user. It can be added to the user's event list using the SimGauss **ADDEVENT** command. This command specifies at what time during the simulation the keyword is to be executed. When the simulation is run using either **START** or **GO**, each keyword on the user event list, whose specified time is later than the current time, is transferred to the simulation event queue. The keywords are then executed at the specified time in the same way as any other simulation events.

The **CLREVENT** command is provided to remove keywords from the user event list, to clear the user event list and to display the user event list.

Typical uses for user event keywords are to introduce changes into the simulation at particular times. These could be, for example, applying step changes to states, turning on and off disturbance forces, switching controller modes, changing integration algorithms, etc.

Since GAUSS keywords can not return results all modifications carried out by user event keywords must be via global variables, or by calling SimGauss or GAUSS commands. When a user event keyword is executed by SimGauss it will be passed an empty string as its argument.

This is a very powerful facility and can easily invalidate your model if you are not careful. For example changing the dimensions of any of the variables in your model is likely to cause trouble. Trying to assign values to the model's derivatives will also not work as the these will be re−calculated at the beginning of the next integration step. Also calling SimGauss or GAUSS commands that **RUN** files, such as **SIMGAUSS**, **PRTVARS**, **PLOTVARS** and **RUN**, will terminate the simulation and return you to the GAUSS command mode.

7.2 User Event Example

The Digitally Controlled Spring System of Chapter 4, A Full Model, will be used to illustrate how user event keywords can be used to introduce disturbances into the system. The disturbance will be in the form of a step change in the velocity of the mass at 1.6 sec. This is equivalent to giving the mass a sharp tap.

After compiling the **ctrlsprg.sgm** model, the following user event keyword is defined to produce a step change in the velocity. (If you define this keyword from the GAUSS command mode, press **Ctrl+Enter** keys at the end of the lines to prevent GAUSS from trying to compile an incomplete procedure. Press the **Enter** key after the **endp;** to compile the keyword procedure.)

```
keyword tap(str);
logdata(1);
v = v + 2;
logdata(1);
endp;
```

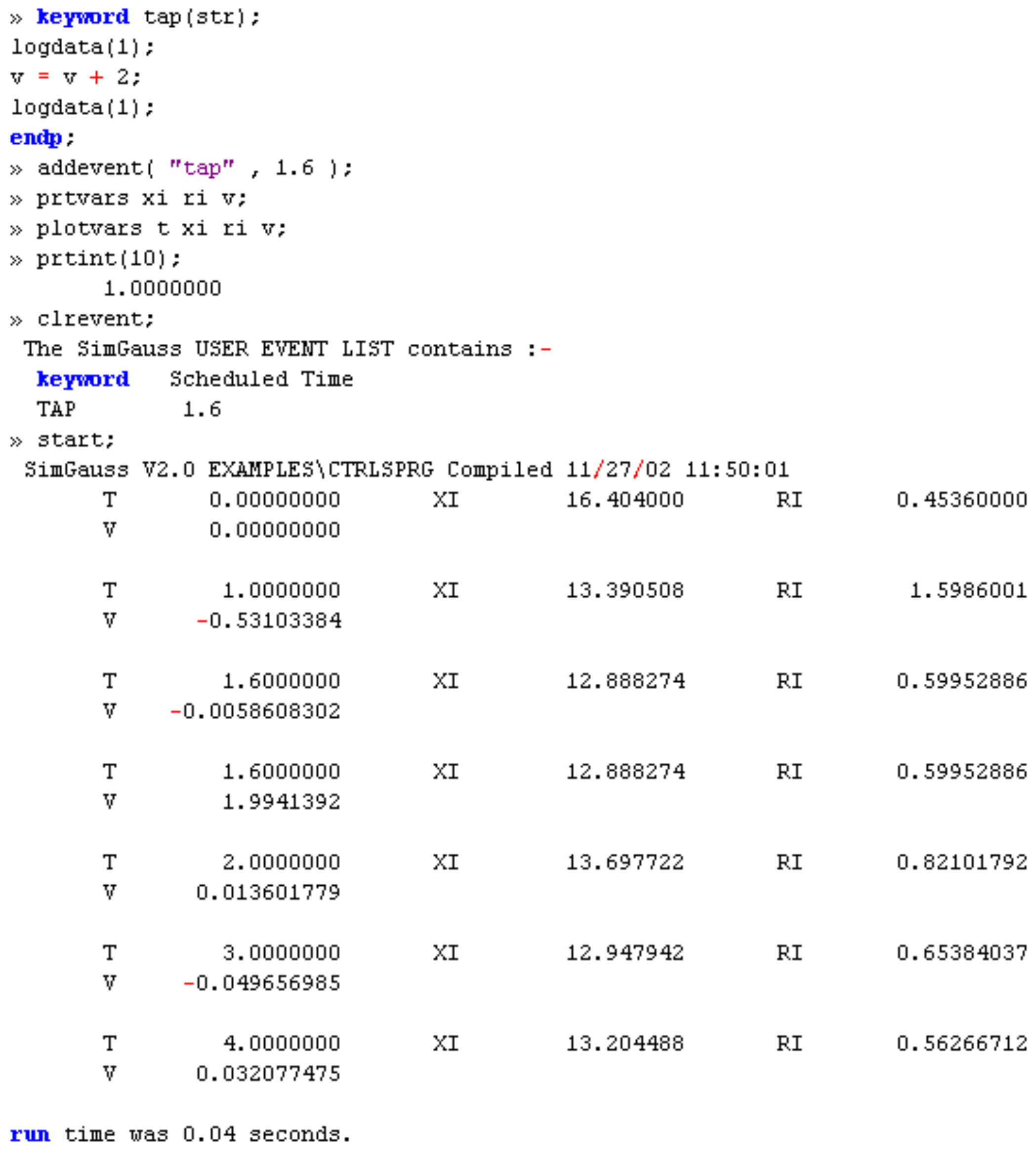
Having defined `tap`, **ADDEVENT** is used to put this keyword on the user event list to be executed at 1.6 sec.

```
addevent( "tap" , 1.6 );
```

The print and plot variables also need to be specified before **START**ing the run.

```
prtvars xi ri v;
plotvars t xi ri v;
prtint(10);
clrevent;
start;
```

The **CLREVENT** command has been used to display what is on the user event list prior to running the simulation. As usual **START** is used to run the simulation.



```
» keyword tap(str);
logdata(1);
v = v + 2;
logdata(1);
endp;
» addevent( "tap" , 1.6 );
» prtvars xi ri v;
» plotvars t xi ri v;
» prtint(10);
       1.0000000
» clrevent;
 The SimGauss USER EVENT LIST contains :-
  keyword   Scheduled Time
  TAP        1.6
» start;
 SimGauss V2.0 EXAMPLES\CTRLSPRG Compiled 11/27/02 11:50:01
       T       0.00000000       XI        16.404000       RI       0.45360000
       V       0.00000000

       T        1.0000000       XI        13.390508       RI        1.5986001
       V      -0.53103384

       T        1.6000000       XI        12.888274       RI       0.59952886
       V    -0.0058608302

       T        1.6000000       XI        12.888274       RI       0.59952886
       V        1.9941392

       T        2.0000000       XI        13.697722       RI       0.82101792
       V      0.013601779

       T        3.0000000       XI        12.947942       RI       0.65384037
       V     -0.049656985

       T        4.0000000       XI        13.204488       RI       0.56266712
       V      0.032077475

run time was 0.04 seconds.
```

Figure 7−1

Figure 7−2 shows a plot of the results using the **PLOTXIRI** procedure defined in the model code. Comparing this plot to Figure 4−3 clearly shows the effect of the sharp tap applied at 1.6 sec.

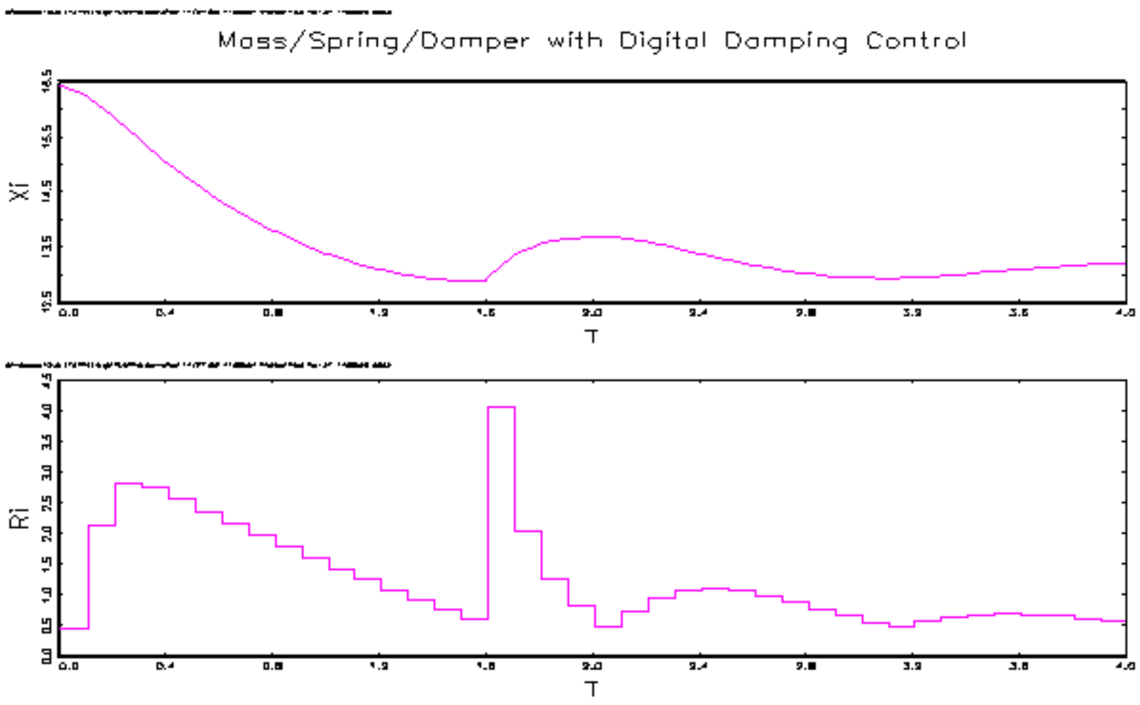


Figure 7−2

Finally, the user event list is not cleared by running the simulation, so running the model again will produce exactly the same results.

Chapter 8

Embedded Simulations

SimGauss models can be run from within other GAUSS procedures. This enables GAUSS' powerful non−linear equation solvers and optimization procedures to be used to solve non−linear differential equation problems and to optimize system parameters.

To include a simulation within another procedure is as simple as adding **START or GO** to that procedure's code. At the end of the run the results of the simulation are available in the model's global variables. Note however, if any of the dimensions of the states, or print or plot variables have changed since the last **START** or SIMGAUSS command, then the procedure in which **START** or **GO** is embedded will not be returned to when simulation run terminates. An initial **START** or **GO** must be executed from the GAUSS command mode to recompile all the necessary SimGauss procedures before starting the simulation proper.

This chapter contains two examples of embedded simulation. In the first example SimGauss is used together with the optional GAUSS Non−linear Simultaneous Equations module to solve a two−point boundary value problem. The second example makes use of the optional GAUSS Optimization module to identify the parameters of a chemical reactor.

8.1 Two Point Boundary Value Problem

The first example of embedding a simulation within a GAUSS procedure will solve the following two point boundary value problem [1] using the optional GAUSS Non−linear Simultaneous Equations module, NLSYS.

$$\frac{d^2y}{dx^2} = -(1+e^y) \qquad y(0)=0 \quad y(1)=1.0$$

To solve this problem using SimGauss, the problem needs to be reformulated into an initial value problem. This is done by assuming a value for y'(0) = dy/dt at t = 0, running the simulation to t = 1, and checking the resulting y(1). **NLSYS** adjusts the assumed value of y'(0) until y(1) has the desired value.

The model code required to solve this problem is contained in **twopoint.sgm**.

```
1. /* twopoint.sgm
2. ** Non-linear two point boundary value problem. */
3.
4. library simgauss,nlsys;
5. #include nlsys.ext;
6. nlset;
7.
8. stoptime = 1; /* stop run at 1 sec. */
9. /* define the conditions that are known */
10. y1 = 1; /* required y at t = 1 */
11. I_y = 0;
12.
13. DYNAMIC(0.1);
14. d_dy = -(1+exp(y));
15. d_y = dy;
16. END_DYN;
17.
18. proc fsys(x);
19. i_dy = x;
20. _sgkey = 0; call start; _sgkey = 1;
21. retp(y1-y);
22. endp;
23.
24. proc solve(dY0,tol);
25. local x,f,j,tcode;
26. stoptime = 1.0; /* stop run at 1 sec. */
27. _nlfvtol = tol;
28. __altnam = "i_dy";
29. __title = "Two Point Boundary Value Problem";
30. __output = 1;
31. output file = twopoint.out reset;
32. {x,f,j,tcode} = nlprt(nlsys(&fsys,dY0));
33. output off;
34. i_dy = x;
35. retp(x);
36. endp;
```

Lines 4 to 6 set up the required libraries and initialize NLSYS. Lines 8 to 11 set up the constants need for this simulation. Although unassigned initial conditions are assumed to be zero, **`i_y`** is explicitly set to zero here to ensure any old values left in the GAUSS workspace are not used by mistake.

Lines 14 and 15 define the derivative equations. **`d_dy`** is the second derivative of **`y`**.

Lines 18 to 22 define the **FSYS** procedure, which will be called by NLSYS to solve the two point boundary value problem. The procedure is passed one parameter, the initial value of **`dy`**. After setting **`i_dy`** to this value the simulation is run and the error between the value of **`y`** at 1 sec and the required value, **`y1`**, is returned. **START** is **CALL**ed to suppress the run time message which is normally returned. The SimGauss global variable **_sgkey** is set to zero before calling **START** to disable the keyboard check for the user interrupt key, **H**

The `solve` procedure (lines 24 to 36) is the main executable procedure which takes as arguments an initial guess for dy at t=0 and the required accuracy of the answer. It returns the value of `i_dy` required to give y(1)=1.

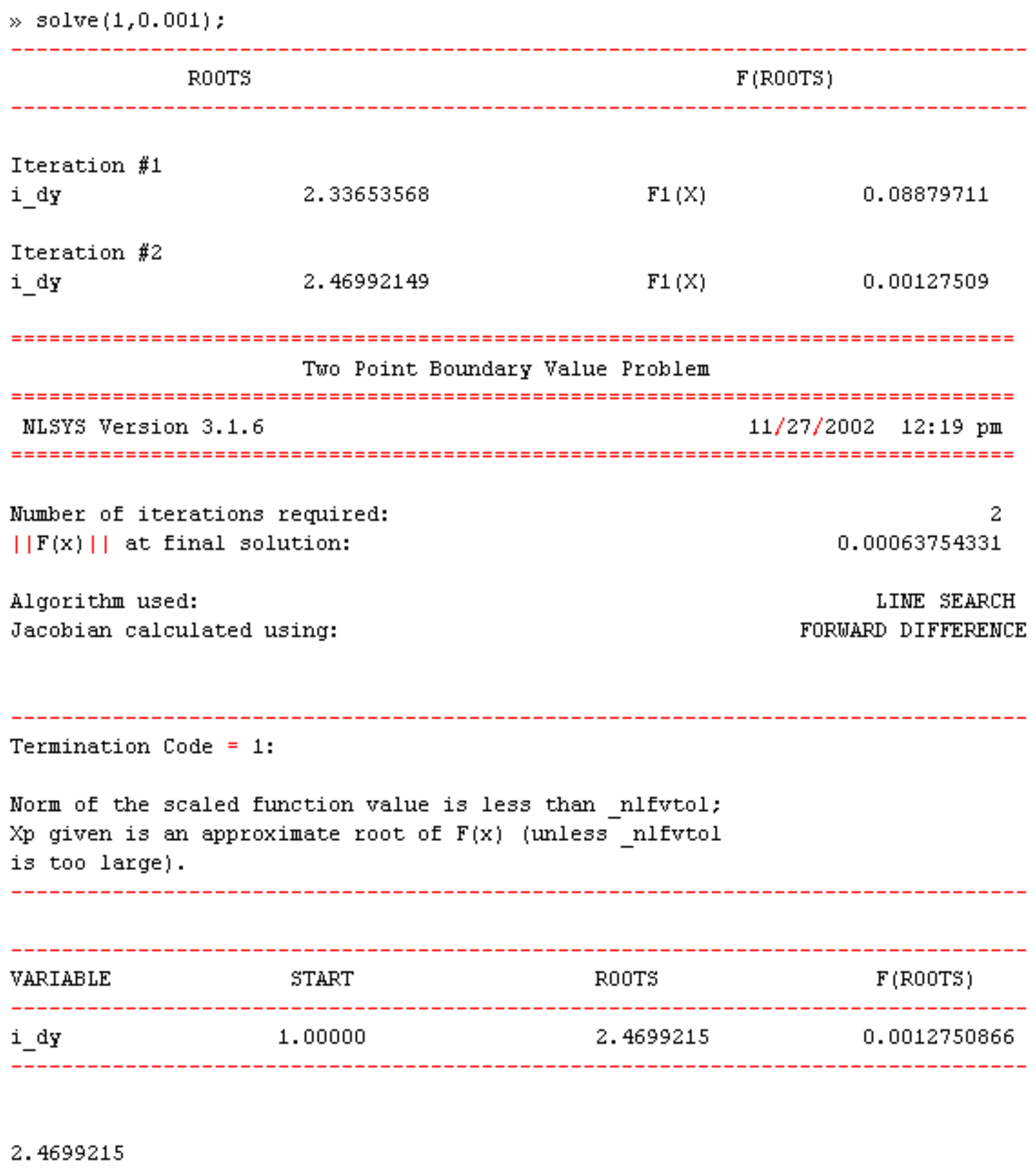
Line 26 sets **stoptime** to 1. This ensures that each simulation run stops at exactly t=1. Lines 27 to 30 set the NLSYS global variables. Line 31 sets the output file and line 32 calls NLSYS to solve for the initial condition. Line 34 assigns the result returned by NLSYS to `i_dy` before returning its value.

After translating the model and clearing the print variables the `solve` procedure is used to find the value of `i_dy`.

```
simgauss examples\twopoint;
prtvars clearall;
solve(1,0.001);
```

Figure 8−1 shows the results of the `solve` procedure using the initial estimate of 1.0 for `i_dy` and an accuracy tolerance of 0.001.

The **PRTVARS clearall** command was used to clear the print variables so that not even the default print variable, `t`, is displayed during each run.

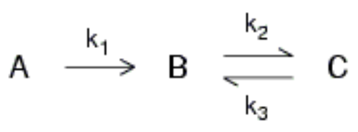


```
» solve(1,0.001);
--------------------------------------------------------------------------------
              ROOTS                                      F(ROOTS)
--------------------------------------------------------------------------------


Iteration #1
i_dy                   2.33653568                 F1(X)            0.08879711

Iteration #2
i_dy                   2.46992149                 F1(X)            0.00127509

===============================================================================
                       Two Point Boundary Value Problem
===============================================================================
 NLSYS Version 3.1.6                                    11/27/2002   12:19 pm
===============================================================================

Number of iterations required:                                              2
||F(x)|| at final solution:                                  0.00063754331

Algorithm used:                                                   LINE SEARCH
Jacobian calculated using:                                 FORWARD DIFFERENCE


--------------------------------------------------------------------------------
Termination Code = 1:

Norm of the scaled function value is less than _nlfvtol;
Xp given is an approximate root of F(x) (unless _nlfvtol
is too large).
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
VARIABLE             START                    ROOTS                 F(ROOTS)
--------------------------------------------------------------------------------
i_dy                 1.00000                  2.4699215             0.0012750866
--------------------------------------------------------------------------------


2.4699215
```

Figure 8−1

8.2 Identification of a Chemical Reactor

The second example identifies the reaction rates of a chemical reactor assumed to have the following model



Where A,B and C are the concentrations of the three reactants and k1,k2 and k3 are the three reaction rate constants. The reactor model can be expressed in differential equations as

$$\frac{dA}{dt} = -k_1 A$$
$$\frac{dB}{dt} = k_1 A + k_3 C - k_2 B$$
$$C = A_0 + B_0 + C_0 - A - B$$

The available experimental data for the concentrations is

Time	**A**	**B**	**C**
0	0.732	0.0	0.0
3	.	0.49	0.0957
6	0.0373	0.481	0.229
9	0.00835	0.399	0.345
12	0.00189	0.342	0.406
15	0.000419	0.276	0.467

Note that the measurement of concentration of A at 3 seconds is missing.

To identify the reaction rate constants using GAUSS' optional Optimization module two files are required. The first file contains the SimGauss model code which will predict the concentrations, A,B and C given a set of reaction rate constants. The second file contains the command file to run **OPTMUM** to do the estimation. Read the section on the Optimization in the GAUSS Applications Manual for the details of the **OPTMUM** procedure.

8.2.1 SimGauss Model File

The SimGauss code is contained in the file **chemr.sgm**.

```
/* chemr.sgm   Chemical Reaction Identification */

/* This is simulated data +/-3% normal errors */
data = { 0 0.732 0 0,
3 . 0.49 0.0957,
6 0.0373 0.481 0.229,
9 0.00835 0.399 0.345,
12 0.00189 0.342 0,
15 0.000419 0.276 0.467 };
/* Data must have unique times and */
/* be in increasing time order */

```

```
13. /* Actual values for k1 k2 k3 */
14. k1 = 0.5; k2 = 0.1; k3 = 0.03;
15. stoptime = 1000; /* must be greater than 15 */
16.
17. INITIAL;
18. first = 1; /* set flag */
19. i_A = data[1,2]; i_B = data[1,3]; i_C = data[1,4];
20. mdata = zeros(rows(data),cols(i_A~i_B~i_C));
21. END_INIT;
22.
23. DYNAMIC(1.0); /* start of DYNAMIC section */
24. C = i_A + i_B + i_C − A − B;
25. d_A = −k1*A;
26. d_B = k1*A + k3*C − k2*B;
27. END_DYN;
28.
29. DISCRETE("PROCESS",0); /* this is scheduled */
30. if first;
31. r_data = 1; /* set first row */
32. /* schedule first PROCESS */
33. schedule("PROCESS",data[r_data,1]);
34. first = 0; /* finished setup */
35. else; /* not first time */
36. mdata[r_data,.] = A~B~C;
37. r_data = r_data+1; /* update for next row */
38. if r_data > rows(data); /* finished */
39. call HALT(1); /* set halt flag */
40. else; /* schedule PROCESS for next row */
41. schedule("PROCESS",data[r_data,1]);
42. endif;
43. endif;
44. END_DISC; /* PROCESS */
```

The Gauss Code section, lines 1 to 16, sets up the experimental data matrix and initial reaction rate constants. The values for the reaction rate constants are in fact the true values, but any reasonable values could have been used. It also sets the stoptime out of the way as the length of the run of this model is set by the maximum time in the first column of the data matrix via the DISCRETE section, **PROCESS**.

The INITIAL section, lines 17 to 21, sets the flag **first**, the initial values of the concentrations and zeros out the model data matrix, **mdata**. Lines 23 to 27 are the DYNAMIC section of the model and calculate the varying concentrations over time.

The DISCRETE section **PROCESS**, lines 29 to 44, controls the saving of the simulation results and the stopping of the run. Each time the model is run using the **START** command, the INITIAL, DYNAMIC, all DISCRETE sections and the DATALOG section are executed in the order in which they appear in the model. This is to ensure that all the variables are initialized ready for the first data logging and that all DISCRETE sections are properly **SCHEDULE**d. In this case the INITIAL section sets the flag first so that when the DISCRETE section, **PROCESS**, is executed as part of this setting up procedure it initializes the row number of the **data** matrix to 1 and then **SCHEDULE**s itself to execute at the first time experimental data was collected and clears the **first** flag (line 34).

Subsequent calls to **PROCESS** occur at the times given in the first column of **data**. At each time the concentrations are saved in the model data matrix, **mdata**, (line 36) and the data row index, **r_data**, incremented. If the experimental data points have been exhausted **HALT(1)** is called to terminate the run, otherwise **PROCESS** is re−scheduled at the next experimental data time. Using this form of model allows more points to be added to the data matrix without having to modify the rest of the model.

This file is first compiled using the SIMGAUSS command. Then after checking it functions correctly some runs should be made for various values of **k1**,**k2** and **k3** to select some reasonable starting values for the optimization. Also as the simulation will be run many times during the optimization process it is worth selecting an integration algorithm, step size and error settings which minimize the run times consistent with providing the required accuracy in the results. To allow for the variations in the reaction rate constants that will occur as the optimization proceeds, a variable step size integration algorithm RKF4 was chosen. The default values for **MERROR** and **RERROR** of 1e−4 give acceptable accuracy for the results, while a **INITSTP** and **MAXT** of about 1 second gives the minimum run time.

8.2.2 OPTMUM Model File

At each iteration **OPTMUM** will select values for the constants **k1**,**k2** and **k3** and then run the simulation to generate the predicted values for the concentrations at the experimental measurement times. From these values, stored in **mdata**, the moment matrix of the deviation of the predicted values from the observed values will be computed. From the moment matrix the log−likelihood function will be computed.

Having defined the model file, a command file is needed to run OPTMUM. This file sets up the various constants needed by OPTMUM and defines the function which OPTMUM calls to calculate the log−likelihood for a given set of reaction rate constants. This procedure, call **ESTIMATE**, will call **START** to run the simulation model and then calculate the log−likelihood function from the results. The following code is contained in the file **chemr.opt**.

```
/*  chemr.opt   Chemical Reactor Identification */
library simgauss, pgraph, optmum;
#include optmum.ext;
optset;

/* calculate number of observations taking */
/* into account missing values */
nobs = data[.,2:4] .$/= miss(0,0);
nobs = nobs'*nobs;
```

```
10.
11. Proc(1) = estimate(p);
12. local sigma;
13. k1 = exp(p[1]); k2 = exp(p[2]); k3 = exp(p[3]);
14. _sgkey = 0; call START; _sgkey = 1;
15. sigma = moment(mdata−data[.,2:4],2)./nobs;
16. retp(ln(prodc(diag(sigma))));
17. endp;
18.
19. call maxt(1); call initstp(1); /* set steps sizes */
20. intalg rkf4; /* select variable stepsize algorithm */
21. /* set data logging and the stoptime out of the way */
22. stoptime = 1000; call datastep(1000);
23.
24. _opparnm = 0$+"ln(k1)" | 0$+"ln(k2)" | 0$+"ln(k3)";
25. __title = " Chemical Reactor Identification";
26.
27. x0 = ln(0.6)|ln(0.15)|ln(0.1); /* initial guess */
28. output file = chemr.out reset;
29. print "\r\l Initial values " exp(x0');
30. {x_fin,f_fin,g_fin,retcode}=optprt(optmum(&estimate,x0));
31. k1=exp(x_fin[1]); k2=exp(x_fin[2]); k3=exp(x_fin[3]);
32. print " k1 = " k1 " k2 = " k2 " k3 = " k3;
33. output off;
```

Lines 2 to 4 set the optimization library and reset **OPTMUM**'s global variables. Lines 8 and 9 calculate the number of observations, **nobs**, for each element of the moment matrix to take into account the missing observation at 3 seconds. If there were no missing data then **nobs** would be the number of rows in **data**.

The procedure **ESTIMATE** is defined in lines 11 to 17. It is this function which **OPTMUM** calls to calculate the log−likelihood of a particular set of parameters. **OPTMUM** passes the values for **ln(k1)**, **ln(k2)** and **ln(k3)** as a vector **p** to this procedure and expects a single return value, the value of the function to minimize. Since all the reaction constants must be positive the log of the constants is estimated. The procedure **ESTIMATE** first transfers the parameter values to the model's reaction rate constants (line 13) and then runs the simulation for these values. The SimGauss global variable `_sgkey` is set to zero before calling START to disable the keyboard check for the user interrupt key . This allows **OPTMUM** to be controlled from the keyboard during the optimization process.

The moment matrix of the deviation of the predicted values, **mdata**, from the observed values, **data**, is computed on line 15. From the moment matrix the log−likelihood function is computed and returned on line 16.

Normally line 16 would read **retp(ln(det(sigma)));** however in this case there are not enough observations to identify all the components of the moment matrix. Therefore only the diagonal elements of the moment matrix are used. This is the same as assuming that the model's errors are independent. The likelihood function being optimized here is the concentrated log−likelihood, i.e., the error covariance matrix has been factored out of the likelihood.

Lines 19 and 20 set the maximum and initial step sizes and the integration algorithm as determined during testing of the model. Line 22 sets the data logging interval and the **stoptime** out of the way. Lines 24 and 25 set the global parameters for the **OPTMUM** procedure.

The initial values for the reaction rate constants to be used as a starting point by **OPTMUM** are set on line 27 and **OPTMUM** is called on line 30.

8.2.3 Estimating the Parameters

Having translated the **chemr.sgm** file using the **SIMGAUSS** command, all the print variables are cleared to prevent SimGauss producing screen output while OPTMUM has control. Then the optimization is started by running the **chemr.opt** command file.

```
simgauss examples\chemr;
prtvars clearall;
run examples\chemr.opt;
```

At the end of the optimization **k1,k2** and **k3** are assigned the optimum estimates of the reaction rates (line 31). In this case k1 = 0.496, k2 = 0.104 and k3 = 0.0299.

8.2.4 Plotting the Results

To plot the results of the optimization, **SGXY** could be used as before, however the Publication Quality Graphics package requires considerable memory and may not be able to run with SimGauss and OPTMUM loaded on computers with limited memory. In these cases the following commands can be used to save the plot matrix in a GAUSS data set then reload it for plotting using the Publication Quality Graphics.

```
plotvars t a b c;
datastep(0.33);
start;
sgsavep chemr;
saveall chemr;
new;
library simgauss, pgraph;
sgloadp chemr;
title("Chemical Reactor (Optimized Parameters)");
_plegstr = " A\000 B\000 C";
_plegctl = 1;
load _psym = examples\chemrsym;
```

```
sgxy T,A B C;
```

After generating the plot matrix for the optimal parameters it is saved using **SGSAVEP**. Then, after save the current workspace, the workspace is cleared, the Publication Quality Graphics library is set and the plot matrix reloaded using **SGLOADP**. (Refer to the Section 1.2 'Customizing SimGauss' in the Introduction for a fast way of loading the plot routines.)

The original experimental data points are stored in **chemrsym.fmt** in a form suitable for assigning to the Publication Quality Graphics variable **_psym** which will plot the data points on the graph. Figure 8−2 compares the experimental data to the results of the model using the optimal rates.

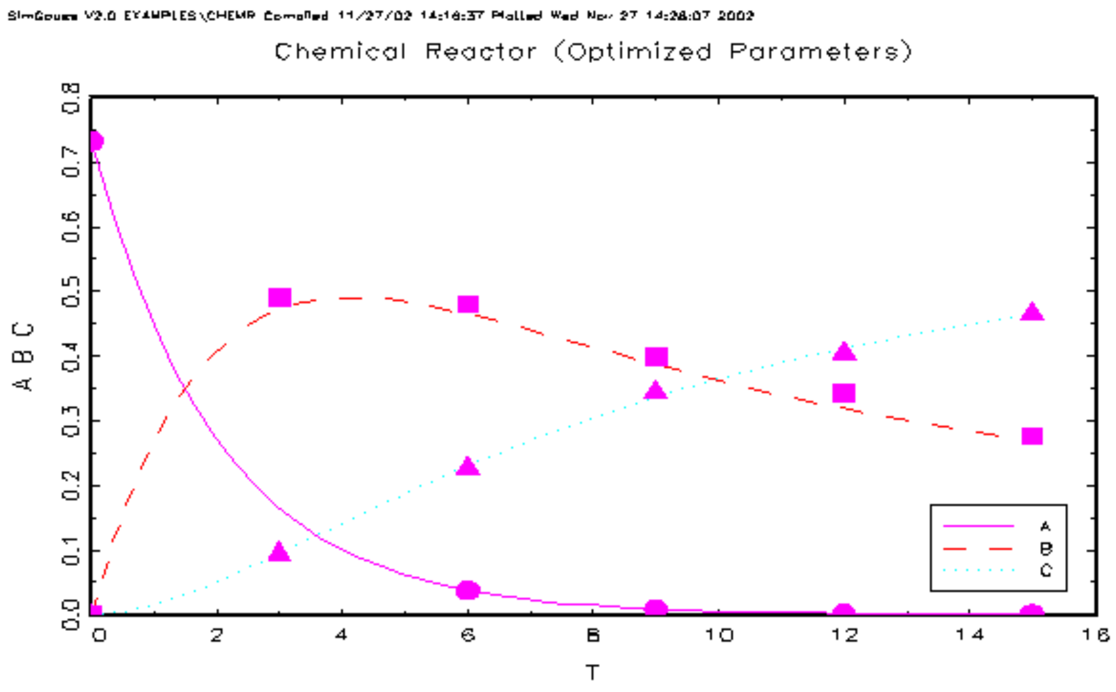


Figure 8−2

References

[1] The SCi Continuous System Simulation Language (CSSL), Simulation, December 1967, pp. 281−303

Chapter 9

Debugging Models

Debugging SimGauss models is not very different from debugging any other GAUSS program once you understand the steps involved in compiling a SimGauss model. Read the chapter on Error Handling and Debugging in the GAUSS Manual first.

There are three types of errors that can occur: translation errors, compile errors and execution errors. These will be discussed in the light of the steps SimGauss takes to compile and execute a model. The SimGauss debugger will also be discussed as well as the changes SimGauss makes to the GAUSS environment when it runs a model.

9.1 Translation errors

Before compiling the model, SimGauss scans the model code in the **.sgm** file to identify the states and to break the model up into the necessary procedures.

The **.sgm** file is not changed in any way by the translation process and the line numbers given in the translator error messages refer to the **.sgm** file. All translator error messages are written to the error file, **simgauss.err** and to the output screen.

The translator preforms a minimal amount of error checking on the model code. The translator is primarily looking for derivative statements, in order to determine the model states, and for SimGauss section commands, which define the start and end of the various sections of the model.

The SimGauss section commands are paired. Table 9−1 shows the section command pairs (also see Figure 3−1).

Table 9−1
SimGauss Section Command Pairs

INITIAL	END_INIT
DYNAMIC	END_DYN
DISCRETE	END_DISC
DATALOG	END_DL
TERMINAL	END_TERM

Only the **DYNAMIC**, **END_DYN** pair are compulsory. All the others are optional. However if they appear in the model they must appear in the order shown or the translator will abort with an error. The **DISCRETE**, **END_DISC** pair is the only pair of section commands that may be used more than once.

The translator ignores all comments delimited by /* and */ but treats comments as delimiters, just as if they where replaced by a space character. Any @ symbols found outside comments or strings are flagged as errors. The translator does not support the @ form of GAUSS comments.

The opening pair of double quotes for strings must be matched by a closing pair of double quotes on the same line or the translator will abort with an error message. Long strings can be continued on the following line by using "\ at the end of the first line of the string. Carriage returns and line feeds can be inserted into strings using the special character pairs \r and \n inside the double quotes.

SimGauss section commands must be at the beginning of a GAUSS statement in order to be recognized by the translator. A common error is not to have the previous statement terminated by a semi−colon. The translator then misses the section command altogether and generates an error when the next section command in encountered.

From the **.sgm** file the translator produces two other files. The **.sgs** file, which defines all the model's states and initializes them in the GAUSS workspace, and the **.sgp** file, which contains the procedures produced from the model code.

The procedures generated by the translator for each section of the model are given in Table 9−2.

Table 9−2
SimGauss Model Procedures

Gauss Code	None
INITIAL Section	**SGP_INIT**
DYNAMIC Section	**SGP_DYN**
DISCRETE Section(s)	Procedure name given in **DISCRETE** statement
DATALOG Section	**SGP_DL**
TERMINAL Section	**SGP_TERM**
Gauss Procedures	Names unchanged

The Gauss Code section of the model is not transferred to the **.sgp** file except for comments and **#LINESON;** and **#LINESOFF;** statements. No procedures are generated by the translator for this section of the model.

The model code in the INITIAL, DYNAMIC, DATALOG and TERMINAL sections is not changed in any way. All the code, except the SimGauss section commands and the DECLARE statements, is transferred unchanged to the body of the respective procedure. The **DECLARE**s are filtered out to prevent redefinition errors when the three files are compiled by SimGauss.

The DISCRETE sections are handled a little differently. Part of the **DISCRETE** statement contains the name to be used for the procedure written by the translator. The translator generates a procedure of that name whose body is the model code in that DISCRETE section. The last line of the procedure is the original **DISCRETE** statement which re−**SCHEDULE**s the procedure to execute at its next repetitive time. If the repetition time is zero when this **DISCRETE** statement is executed the procedure will not be re−**SCHEDULE**d.

The Gauss Procedures section of the model is not copied to the **.sgp** file. No addition procedures are generated by the translator for this section of the model.

In addition to generating the **.sgs** and **.sgp** files, which are opened in the same directory as the **.sgm** file, the translator uses some temporary files. These temporary files are opened in the current default directory. The temporary files are deleted when the translator terminates.

9.2 Compilation Errors

If the translator finds no errors in the **.sgm** file, SimGauss then **RUN**s the three files, **.sgs**, **.sgm** and **.sgp** in that order, to compile the model ready to be run with the **START** command.

9.2.1 .sgs File

When the **.sgs** file is **RUN** it defines all the state variables as global variables in the GAUSS workspace. All the state variables are then set to GAUSS' missing value code.

The **.sgs** file also defines all the initial value, maximum error and relative error variables as globals in the GAUSS workspace. Any of these variables that were not already initialized are set to zero.

If any of these variables already exist in the GAUSS workspace but are defined as other than scalars or column vectors, then an appropriate error message is printed and SimGauss returns to the GAUSS command mode.

9.2.2 .sgm File

The **.sgm** file, containing the user's model code, is then **RUN**. Since the model sections in this file are not enclosed in procedures, all the model variables are compiled as globals in the GAUSS workspace. They are then accessible from all other procedures and also interactively from the GAUSS command mode.

SimGauss does not attempt to sort the model code, so all variables must be defined before they are referenced, just as for normal GAUSS programs. In general this means that variables must appear on the left−hand side of an equals sign or in an EXTERNAL statement before they can be used in an equation or procedure call.

The exceptions to this are the state variables, the initial conditions and the maximum and relative error variables. These variables have all been set up in the GAUSS workspace by the **.sgs** file prior to compiling the model code contained in the **.sgm** file. However the state variables contain missing values until **after** the INITIAL section of the model has been executed, so do not use them on the right−hand side of equations in the Gauss Code or INITIAL section of the model.

If SimGauss is being run with AUTODELETE OFF then any procedures defined in the **.sgm** file need to have an associated **EXTERNAL** statement in the GAUSS code section of the model.

When the GAUSS compiler finds an undefined symbol it assumes it is a procedure. With AUTODELETE OFF after the compilation has terminated, GAUSS still has the symbol identified as a procedure so the next compilation will also fail unless the symbol is cleared from the workspace.

This can be done with either **NEW** or **DELETE** but in either case the GAUSS workspace will be completely overwritten with the SimGauss code the next time SIMGAUSS is executed. This is because the SimGauss procedures with global references are also deleted when DELETE is used and so SimGauss needs to be reloaded.

If AUTODELETE is ON, **DELETE** will be executed automatically by GAUSS at the end of the compilation, if there were any compilation errors. However with AUTODELETE ON any typing error is also likely to result in an undefined procedure error which will delete all procedures with global references, i.e. most of the SimGauss procedures and your model procedures. This will require you to reload SimGauss and re−compile your model. AUTODELETE ON is the default setting for GAUSS but for use with SimGauss it is more convenient to set it to OFF using the GAUSS menu Configure, Preferences − Compile Options.

After compiling the **.sgm** file GAUSS executes it. This executes any Gauss Code section of the model that appears at the top of the **.sgm** file. Any execution errors in the Gauss Code section are reported by GAUSS at this time and SimGauss will return to the GAUSS command mode. If a **#LINESON;** statement appears at the beginning of the model code, then GAUSS will report the line number of the offending statement. Since this code is a GAUSS program no active procedure name message will be give after the error message. Execution time error messages caused by any other section of the model will give the active procedure name shown in Table 9−2.

Certain commands in the Gauss Code section will cause SimGauss to terminate and return to the GAUSS command mode without completing the compilation of the model. These are any commands which **RUN** files, such as **PRTVARS** and **PLOTVARS**, also any **PRINT** commands which do not perform a Carriage Return before the end of the Gauss Code section, and any **LOCATE** commands.

PRINT;; and **LOCATE** statements will cause SimGauss to abort the compilation with an error message. The restriction on the use of **RUN** statements applies to all sections of the model code, but **PRINT** and **LOCATE** statements can be used freely in any section of the model other than the Gauss Code section.

When execution reaches the **INITIAL** or **DYNAMIC** statement further execution of the **.sgm** file is aborted and the **.sgp** file is **RUN**. This is why statements in the Gauss Procedures section of the model, which are outside procedures, are never executed.

9.2.3 .sgp File

The **.sgp** file contains the model code as procedures which are **CALL**ed by SimGauss during the simulation. Since the model sections which are enclosed by SimGauss section commands are translated into procedures, these sections should not contain any procedure definitions or any **GOTO**s which branch outside the section. Also **IF** and **DO** statements must not extend outside the section.

After compiling the **.sgp** file SimGauss evaluates the INITIAL section of the model and transfers the initial values to the model's states. SimGauss then executes the DYNAMIC section of the model, all the DISCRETE sections and the DATALOG section, to initialize all the model's variables. No DISCRETE sections are **SCHEDULE**d during this phase of the execution. The DISCRETE sections are executed strictly in the order they appear in the model.

The dimensions of the states and the derivatives are checked after executing the DYNAMIC section and after each of the DISCRETE sections are executed. If any of the dimensions have changed an error message is displayed indicating the state or derivative effected.

As the DISCRETE sections are also executed in this order each time the model is **START**ed you can check at this stage that the model will start in the correct state. For example if there are two DISCRETE sections, one to turn a switch on and one to turn it off, by arranging the order of the sections the model can be made to start in either state. Alternatively a user event to be executed at **i_t** can **SCHEDULE** the appropriate DISCRETE section to also execute at **i_t** to set the model to the correct state.

Finally the **.sgp** file generates and **RUN**s the state transfer procedure and returns to the GAUSS command mode with the message

```
     Use START to run the model.
```

9.3 Execution Errors

Execution errors in the Gauss Code section of the model will be reported when the **.sgm** file is **RUN** by the SimGauss compiler as explained above. Other execution time errors generated by the model code will occur when the respective procedures listed in Table 9−2 are executed.

The INITIAL, DYNAMIC, DATALOG and all the DISCRETE sections of the model are executed once with the model's initial conditions, as part of the SimGauss compilation process. Note that this means that the INITIAL section of the model has already been executed once before the first **START** command runs the model so care must be taken in the use of counters which are incremented in the INITIAL section.

The INITIAL section of the model can use any variables already defined in the GAUSS workspace, including those defined in a preceding Gauss Code section. Do not use the state variables themselves in the INITIAL section as the initial conditions are not transferred to the states until **after** the INITIAL section is executed. It should always be possible to write the initial conditions in terms of variables already defined.

Unlike the rest of the model's sections, the first time the TERMINAL section is executed is at the end of the first run. Although execution time errors in the TERMINAL section will cause the simulation to abort, you can usually recover the results of the simulation from the model's global variables. However it is wise to try a short run first to test out the entire model.

As all compilation of the model is done with line number tracking off, to assist with locating execution errors **#LINESON;** and **#LINESOFF;** statements may be put in the model code and these will be copied over to the **.sgp** file so that line numbers will be available for the model procedures. Any line numbers given for execution errors in the model's INITIAL, DYNAMIC, DATALOG, TERMINAL or DISCRETE procedures refer to the **.sgp** file. Look at that file to locate the model statement that has caused the error, then edit the **.sgm** model code file to correct the error and call SIMGAUSS to recompile the model.

The Gauss Procedures section of the model is not copied to the **.sgp** file so the line numbers for any errors in these procedures refer to the **.sgm** file.

Extensive error checking is performed in the SimGauss commands and you should receive a useful error message if you use them incorrectly. You should not normally encounter a GAUSS execution error which refers to an internal SimGauss procedure, i.e. a procedure whose name starts with **`'S_'`**. The exception to this is the **S_PLOT** routine. **S_PLOT** will terminate if the SimGauss plot matrix, where the plot variables are saved, exceeds the maximum size allowed for a GAUSS matrix.

Also since user event keywords can change any of the model's global variables you can easily generate GAUSS errors such as

```
    (0):error G0047:Rows don't match
```
or
```
    (0):error G0036:Matrices are not conformable
```

Other than this, if you do get an error message referring to an internal procedure, please report it and how it was generated.

9.4 SGDEBUG

A debugging procedure, **SGDEBUG**, is provided to give some information on what SimGauss is doing internally. The purpose of **SGDEBUG** is twofold, to assist the user in tracking his model's execution and to clarify possible SimGauss problems. Later releases will not necessarily provide the same information.

Large volumes of output are generated by the debugger so it is advisable only to turn it on when required. For example

```
sgdebug(t>1.5 and t<1.6);
```

will turn the debugger on when the time is between 1.5 and 1.6. This statement is best placed in the DYNAMIC section.

The SimGauss debugging is turned off quietly each time a model is compiled.

9.5 SimGauss Defaults

9.5.1 Print Format

The SimGauss print format used to output data at each data logging interval is controlled by the GAUSS **FORMAT** command.

9.5.2 Other Defaults

When SimGauss is loaded the default integration algorithm is Runge−Kutta 4th order, there are no plot variables specified, the time, **t**, will be printed each data logging interval and the user interrupt key, **H**, is enabled.

SimGauss Reference

Command Summary

Model Sections

INITIAL	start of the INITIAL section.
END_INIT	end of the INITIAL section.
DYNAMIC	start of the DYNAMIC section.
END_DYN	end of the DYNAMIC section.
DISCRETE	start of a DISCRETE section.
END_DISC	end of a DISCRETE section.
DATALOG	start of the DATALOG section.
END_DL	end of the DATALOG section.
TERMINAL	start of the TERMINAL section.
END_TERM	end of the TERMINAL section.

Printing and Plotting

DATASTEP	set the interval between data logging.
LOGDATA	force data logging.
PLOTVARS	specify variables to be saved for plotting.
PRTVARS	specify variables to be printed.
PRTINT	specify the number of data logging intervals between printing outputs.
SGXY	xy plot of variables from plot matrix.
SGLOGX	logx plot of variables from plot matrix.
SGLOGY	logy plot of variables from plot matrix.
SGLOGLOG	loglog plot of variables from plot matrix.
SGPLTVAR	extract variables from plot matrix.
SGSAVEP	save plot matrix to data set.
SGLOADP	load plot matrix from data set.
KEEPPLOT	prevent plot matrix from being over written each run.

Simulation and Integration

SIMGAUSS	translate and compile model.
START	run the model from its initial conditions.
GO	continue the model run from where it was halted.
REINIT	set initial conditions to the current states.
INTALG	select integration algorithm.
MAXORD	set the max. integration order.
INITSTP	set the initial integration step size.
MAXT	set the max. integration step size.
MINT	the min. integration step size.
MERROR	set the maximum state errors.
RERROR	set the relative state errors.
i_t	initial time to integrate from. 5
stoptime	final time to integrate to.
_sgkey	controls scanning of keyboard for user interrupt.
errmint	error flag for variable step size algorithms.

Trimming and Linearization

SGLIN	linearize model around the current state.
OBSERV	set the observable variables for the linearized model.
CONTROLS	set the controllable variables for the linearized model
STATES	display the model's states.
SGTRIM	trim the model for steady state conditions.
FREEZE	remove states from the trim process.

Event Handling

ADDEVENT	add user event keyword to list.
CLREVENT	clear user event keyword from list.
EVENTQUE	display the event queue.
HALT	set or test the halt flag
SCHEDULE	schedule a DISCRETE section.

Utilities

BACKLASH	backlash or hysteresis.
BOUND	bound input.
DEADBAND	dead band around zero.
DELAY	delay input.
IMPLICIT	solve algebraic equations.
LIMINT	limit a state variable.
PHYLIMIT	limit a second order system.
QUANTIZE	quantize the input.
SETEPS	set SimGauss' epsilon.
SGDEBUG	turn debugging on or off.
TABLE	look up a tabulated function.

_sgkey

Purpose

Controls scanning of the keyboard for user interrupts.

Format

_sgkey	flag. If non−zero the keyboard is scanned for user interrupts. If zero the keyboard is not scanned.

Remarks

When **_sgkey** is non−zero, SimGauss checks the keyboard at each event for the user interrupt key, **H**. In the process it consumes all the pending key strokes in the keyboard buffer. If the model is being run by another GAUSS program such as **OPTMUM** setting **_sgkey** to zero prevents SimGauss from taking the key strokes intended to control **OPTMUM**.

Globals

_sgkey

ADDEVENT

Purpose

Adds a user event keyword to the user event list.

Format

ADDEVENT*(keyword , evnttime);*

Inputs

keyword	string, contains the name of the user event keyword. User event keywords will always be passed an empty string when executed by SimGauss.
evnttime	scalar, the time at which the user event keyword is to be executed.

Remarks

At each **START** and **GO** the events in the user's event list, whose specified time is later than the current time, are transferred to the simulation's event queue for execution at the specified times.

The procedures formed from the model's DISCRETE sections cannot be added to the user's event list. They can only be **SCHEDULE**d from within the model.

Example

```
addevent("tap",1.0);
```

This example adds the **tap** keyword to the user's event list to be executed when **t** reaches 1.0

See Also

CLREVENT, EVENTQUE, SCHEDULE, SETEPS

BACKLASH

Purpose

Implements backlash or hysteresis.

Format

BACKLASH(*y* , *x* , *bklsh* **);**

Inputs

y	string, contains the name of the output variable, an Nx1 vector. This variable must have been assigned a value before the first call to **BACKLASH**.
x	Nx1 vector or scalar, half the width of the backlash.
bklsh	Nx1 vector, the input to the backlash.

Outputs

None. The output specified by *y* is updated using **VARPUT**.

Example

```
backlash("xout",x,0.5);
```

In this example a backlash of 0.5 is put between **x** and **xout**.

```
backlash("xout", x−(up−down)./2, (up+down)./2);
```

The second example implements an unequal backlash. The output, **xout**, will move when the **x** is greater than **xout + up** or less than **xout − down**. Note that element by element division has been used to allow for vector arguments.

Source

sg_utils.src

See Also

DEADBAND

BOUND

Purpose

Bounds the input.

Format

y = **BOUND(** *x* , *botlim* , *toplim* **);**

Inputs

x	Nx1 vector or scalar, the continuous input.
botlim	Nx1 vector or scalar, the bottom limit.
toplim	Nx1 vector or scalar, the top limit.

Outputs

y	Nx1 vector, the bounded output.

Remarks

Each element of *y* is calculated from the respective elements of the input by

y = botlim	when *x botlim*
y = x	when *botlim < x < toplim*
y = toplim	when *x toplim*

BOUND should not be used to limit a state variable as the integrator will continue to wind up the state, use **LIMINT** instead.

Example

```
y = bound(5|3, 6|4, 3.5)
y
  5
  3.5
```

Source

sg_utils.src

See Also

LIMINT, PHYLIMIT

CLREVENT

Purpose

Clears event keyword from user's event list.
OR
Displays the user's event list.

Format

CLREVENT *keyword;*
CLREVENT clearall;
CLREVENT;

Inputs

keyword	the name of the user event to be cleared. The event keyword of that name with the largest time is cleared. If the keyword is **clearall**, then the entire user's event list is cleared. If the keyword is not given OR the keyword is not found in the user's event list, then the current list is displayed.

Example

```
» addevent("tap",4.5);
» addevent("tap",2.5);
» clrevent tap;
» clrevent;
The SimGauss USER EVENT LIST contains :-
Keyword   Scheduled Time
2.5

» clrevent clearall;
Clearing user event list.CODE>
```

See Also

ADDEVENT, EVENTQUE

CONTROLS

Purpose

Sets the controllable variables for **SGLIN**.

Format

CONTROLS *variables***;**
CONTROLS clearall *variables***;**
CONTROLS;

Inputs

variables	a list of the global numeric column vectors, separated by spaces, to be added to the existing controllable variables. The names of control variables may not exceed 6 characters. If the list contains the name **clearall**, then the current control variables are cleared. Variables following **clearall** are then added to the list of controllable variables. If the list is empty the current controllable variables will be displayed. Indexing of elements is not allowed. The entire vector is controllable.

Remarks

If there are no controllable variables then **SGLIN** will return missings for the B and D matrices. Initially there are no controllable variables.

The order of the list of controllable variables displayed by **CONTROLS** is the order of the columns of the B and D matrices returned by **SGLIN**. Model states may **not** be controllable variables.

Example

```
controls u v;
```

This example adds the variables **u** and **v** to the current controllable variables.

See Also

SGLIN, STATES, OBSERV

DATALOG

Purpose

Indicates the start of the model's DATALOG section and sets the initial data logging interval.

Format

DATALOG(*interval* **);**

Inputs

interval	scalar, if positive, it is the time interval between data logging. If negative, it is rounded to give the number of integration steps between data logging.

Remarks

The **DATALOG** section must not modify either directly or indirectly any variables that appear in any other model sections. To change the data logging interval from the GAUSS command mode or from within the model code use the **DATASTEP** command.

If the interval is positive, it must be greater than SimGauss' epsilon which can be changed with the **SETEPS** command. The default epsilon is 20 times the machine's epsilon.

If there is no DATALOG section in the model and **DATASTEP** has not been used to set the data logging interval, then the initial data logging interval is set to the *initstep* argument of the **DYNAMIC** statement.

Since the model code in the DATALOG section is put into a separate procedure (**SGP_DL)** by the SimGauss translator, you cannot branch out of this section using **GOTO** statements and procedure definitions are not allowed in this section. Also **IF** and **DO** statements must not extend outside this section. This procedure is called at each data logging interval before displaying the print variables and saving the plot variables.

Example

```
DATALOG(0.01);
```

See Also

END_DL, DATASTEP, LOGDATA, SETEPS, INITIAL, DYNAMIC, DISCRETE, TERMINAL

DATASTEP

Purpose

Sets the interval between data logging.

Format

interval = **DATASTEP(** *interval* **);**

Inputs

interval	scalar, the interval between data logging. If the *interval* is zero the current setting is not changed. If positive, *interval*, sets the time interval between data logging. If negative, *interval* is rounded to give the number of integration steps between data logging.

Outputs

interval	scalar, the previous interval between data logging.

Remarks

The initial interval between data logging is set by the **DATALOG** statement in the model code (or the *initstep* argument of the **DYNAMIC** statement if there is no **DATALOG** or **DATASTEP** statement).

At each data logging time, the integration time is brought to within half SimGauss' epsilon of the exact time, with a short step if necessary, and the DATALOG section of the model executed before saving the plot variables and displaying the print variables. The display of the print variables is controlled by the **PRTINT** statement.

Example

```
dat_tmp = datastep(datastep(0)*2);
```

This statement doubles the current interval between data logging and saves the previous interval in **dat_tmp**.

See Also

DATALOG, PRTINT, LOGDATA, DYNAMIC

DEADBAND

Purpose

Implements a dead band around zero.

Format

y = **DEADBAND(** *x* , *botlim* , *toplim* **);**

Inputs

x	Nx1 vector, the continuous input.
botlim	Nx1 vector or scalar, the dead band in the negative direction.
toplim	Nx1 vector or scalar, the dead band in the positive direction.

Both *botlim* and *toplim* must be greater than or equal to zero.

Outputs

y	Nx1 vector, the output from the dead band.

Remarks

Each element of *y* is calculated from the respective elements of the input by

y = x + botlim	when x < −botlim
y = 0.0	when −*botlim* *x* *toplim*
y = x − toplim	when *x* > *toplim*

Example

```
y = deadband(5|−1, 0.3, 2)
y
      3
     −0.7
```

Source

sg_utils.src

See Also

BACKLASH

DELAY

Purpose

Models a pure time delay.

Format

y = **DELAY(** *x* , *dlytime* , *matname* , *maxrows* **);**

Inputs

x	Nx1 vector, the variable to be delayed.
dlytime	Nx1 vector or scalar, the time x is to be delayed.
matname	string, contains the name of the matrix used to store the past values. This is automatically initialized during the **START** procedure but if it already exists in GAUSS' workspace it must be a matrix.
maxrows	scalar, the maximum number of rows allowed in the delay matrix. The minimum value is 2.

Outputs

y	Nx1 vector, the delayed value of x.

Remarks

To calculate the output, the delay time is subtracted from the current simulation time and the delay matrix searched for two entries that bracket this time. Linear interpolation is then used to interpolate between these values. If the delay matrix does not have sufficient past values the simulation terminates with an error message. The output is equal to the initial value of the input variable until the time has advanced past **i_t** + *dlytime*.

DELAY only operates correctly in the DYNAMIC section of the model. Do not use the same delay matrix in more than one **DELAY** statement.

The value of *dlytime* can vary as necessary during the run but for best results it should vary slowly compared to the integration step size.

The maximum number of rows in the delay matrix can also vary during the model run but must be at least equal to the number of integration steps between the current time, **t**, and **t−** *dlytime*.

For variable step size algorithms, *dlytime*/**MINT**(0) is an approximate upper limit on the maximum number of rows needed.

For fixed step size algorithms, the number of rows needs is approximately
dlytime * (1/**INITSTP**(0) + 1/**DATASTEP**(0))

Example

```
y = delay(x1|x2|x3, dly1|dly2|dly3, "xdlymat",
xdlyrows);
```

In this example a column vector of three variables **x1, x2, x3** is delayed by the number of time units specified by the corresponding delay time vector. The matrix, **xdlymat**, is used to store the past values of the vector and the variable **xdlyrows** specifies the maximum number of rows the matrix may have. The output **y** is a column vector made up of the delayed values of **x1**, **x2** and **x3**.

Globals

The matrix specified by the string, *matname*.

Technical Notes

Each row of the delay matrix contains one past value of the input variable with the past time stored in the first column. **START** fills the matrix with the initial value of the input variable. The time of the first row is set to **i_t** and all the other times are set to a large negative number (approximately the largest negative number the computer can represent).

At the end of each integration step the current values are appended to the top of the matrix and the matrix trimmed to *maxrows*.

See Also

MINT, INITSTP, DATASTEP

DISCRETE

Purpose

Indicates the start of a DISCRETE section in the model code and sets the repetition rate for the execution of that section.

Format

DISCRETE(*procname , period* **);**

Inputs

procname	string, contains the name of the DISCRETE section. It must be a valid GAUSS identifier enclosed in quotation marks. Leading and trailing blanks are not allowed. A string variable is not allowed.
period	scalar, the period at which this DISCRETE section is to be executed. If the *period* is zero then the DISCRETE section is not executed, except at the **START**, unless **SCHEDULE**d.

Remarks

The SimGauss translator puts the code in a model's DISCRETE section into a procedure with the name specified in the string, *procname*. Each time a DISCRETE section is executed it is automatically re−**SCHEDULE**d to execute again at the current time + *period*, only if the period is greater than zero. Half SimGauss' epsilon is used to determine if the period is greater than zero.

When the **START** command is used to run the model, all DISCRETE sections are executed once in the order they appear in the model code and any DISCRETE sections with a non−zero period are **SCHEDULE**d. Any DISCRETE sections with a zero period are not automatically **SCHEDULE**d but are still executed once at the **START**.

The *period* may be varied during the simulation.

The **DISCRETE** statement does nothing when executed from the GAUSS command mode.

Example

```
DISCRETE("sample",ts);
```

This statement defines the beginning of a DISCRETE section called **sample** which is to be executed every **ts** time units during the simulation.

```
DISCRETE("control",0);
```

This statement defines the beginning of a DISCRETE section called **control** which is not executed (except at the **START**) unless **SCHEDULE**d by another DISCRETE section.

Technical Notes

Since the model code in a DISCRETE section is put into a separate procedure, you cannot branch out of this section using **GOTO** statements and procedure definitions are not allowed in this section. Also **IF** and **DO** statements must not extend outside this section.

See Also

END_DISC, SCHEDULE, START, INITIAL, DYNAMIC, DATALOG, TERMINAL, SETEPS

DYNAMIC

Purpose

Indicates the start of the model's DYNAMIC section and sets the initial step size for the integration algorithm.

Format

DYNAMIC(*initstep* **);**

Inputs

initstep	scalar, the initial integration step size.

Remarks

The SimGauss translator puts the code from the model's DYNAMIC section into a procedure called **SGP_DYN**. This procedure is then called by the integration routine to calculate the states' derivatives. This is the only section of the model which is allowed to contain derivative statements. Since the model code in the DYNAMIC section is put into a separate procedure, you cannot branch out of this section using **GOTO** statements and procedure definitions are not allowed in this section. Also **IF** and **DO** statements must not extend outside this section.

For fixed step size integration algorithms the integration step size is set by the **initstep** argument. This can be changed using the **INITSTP** command. If **MINT** has not been used the set the minimum step size, when the first model is run the minimum step size is set to **INITSTP(0)/100**. This limits the smallest step a variable step size integration algorithm may take. This default minimum step size can be changed using the **MINT** command, either in the model code or from the GAUSS command mode.

If there is no DATALOG section in the model and **DATASTEP** has not been used to set the data logging interval, then the data logging interval is set to the *initstep* argument of the first model run.

Example

```
DYNAMIC(0.01);
```

See Also

END_DYN, MINT, MAXT, SETEPS, INITIAL, DISCRETE, DATALOG, TERMINAL

END_DISC, END_DL, END_DYN, END_INIT, END_TERM

Purpose

Indicates the end of a model section in the SimGauss model code.

Format

END_DISC;
END_DL;
END_DYN;
END_INIT;
END_TERM;

Remarks

END_DISC; ends a DISCRETE model section.
END_DL; ends the DATALOG model section.
END_DYN; ends the DYNAMIC model section.
END_INIT; ends the INITIAL model section.
END_TERM; ends the TERMINAL model section.

See Also

DISCRETE, DATALOG, DYNAMIC, INITIAL, DATALOG, TERMINAL

errmint

Purpose

Flag which is set if the integration algorithm needs to reduce its step size below the minimum integration step size.

Format

errmint	flag, set to 1 if the last step failed and the next step of a variable step size integration algorithm would be less than the minimum integration step size.

Remarks

If a variable step size algorithm fails and needs to reduce the step size below the minimum integration step size, then the error flag **errmint** is set to 1 and the DYNAMIC procedure, **SGP_DYN,** is CALLed. If the **errmint** flag is still set when **SGP_DYN** returns, the run stops with a message.

The TERMINAL section is not executed before terminating, but the derivatives are accurate for the current state.

Testing the **errmint** flag allows you to call **MINT** or change the integration algorithm, using **INTALG**, for a few steps to get over a singularity or some other difficult point in the simulation.

You can also use **GO** repeatedly to continue the run using the smaller time step. The simulation will stop again after each step until the required integration step size is larger than **MINT**. This allows you to force the integration over a local discontinuity.

Example

```
if errmint and (1.0E−5 < mint(0));
     call mint(1.0E−5);
     errmint = 0; /* reset errmint */
endif;
```

In this example the **errmint** flag is tested in the DYNAMIC section to see if the integration algorithm wants to reduce the step size below the minimum. If **errmint** is set and **MINT** has not already been reduced to 1.0E−5 then the minimum step size is reduced. The second time **errmint** is set the simulation will terminate as **MINT** is already equal to 1.0E−5.

```
if errmint;
     /* change to fixed step size integ. alg. */
     intalg rk2; /* uses initial step size */
     init_tmp = initstp(mint(0));
```

```
          /* change back in 3 steps */
          schedule("RKF",t+3*mint(0));
          errmint = 0; /* reset errmint */
     endif;
          .....
     discrete("RKF",0);
          /* zero period, needs to be scheduled */
          /* change back to variable step algorithm */
          intalg rkf2;
          call initstp(init_tmp);
     end_disc;
```

In this example if errmint is set **INTALG** is called to change the integration algorithm to a fixed step size algorithm. **INITSTP** is used to set the step size and the previous value is saved in **init_tmp**. The **SCHEDULE** statement schedules a DISCRETE section to switch back to the variable step size integration algorithm after 3 steps of the fixed step size algorithm.

The DISCRETE section **RKF** switches back to the variable step size algorithm and restores the initial value of **INITSTP**.

Globals

errmint

See Also

MINT, INITSTP, INTALG, DYNAMIC

EVENTQUE

Purpose

Prints the current simulation event queue.

Format

call EVENTQUE;

Remarks

This is primarily a debugging statement to be inserted in the model code, most likely in a DISCRETE section or the DATALOG section. It will print a snapshot of the current simulation event queue to the screen each time it is executed.

Use **CLREVENT** to print the current user event list.

Example

```
» eventque;
At time 0.5
The SimGauss EVENT QUEUE contains :-
    Block Name        Scheduled Time
      SAMPLE                0.6
```

In this example the **EVENTQUE** command was executed from the GAUSS command mode when the run had stopped at **t=0.5** If **GO** is used to continue the run the event **SAMPLE** will execute at **t=0.6** The data logging events are not shown on the event queue as these are handled slightly differently to other events.

See Also

SCHEDULE, CLREVENT, ADDEVENT

FREEZE

Purpose

Removes states from the trim process.

Format

FREEZE states**;**
FREEZE clearall *states***;**
FREEZE;

Inputs

states	a list of the states, separated by spaces, to be frozen. The states may have optional numerical indices. If the list contains the name **clearall**, all the states except the time state, **t**, will be unfrozen. The time state, **t**, is always frozen and need not be specified. The states following **clearall** will then be added to the list of frozen states. If the list is empty the current frozen states will be displayed.

Remarks

Open loop integrators cannot be trimmed by **SGTRIM** and the associated state should be frozen using **FREEZE**. Open loop integrators are those states which do not effect their derivative value and have no effect on any of the other states' derivatives and whose derivative is not effected by any other state. These open loop integrators result in a zero row or column in the Jacobian and prevent NLSYS from finding a solution. The time state, t, is a common example of an open loop integrator and is always frozen.

If the dimensions of the any of the state vectors is changed or the model is recompiled, all states are automatically un−frozen.

Example

```
freeze x v[1 2];
```

This example freezes all the components of the state **x** and the first and second components of the state **v**.

See Also

SGTRIM

GO

Purpose

Continues the model run from where it was halted.

Format

runtime = **GO;**

Outputs

runtime	string, the duration of run.

Remarks

The **GO** command is used to continue a run from the state where it was halted. The INITIAL section of the model is skipped. The **stoptime** or halt condition must be changed or the run will halt immediately.

The **GO** command clears the HALT flag and transfers the user's event list to the simulation event queue. It then executes the DATALOG section of the model with the **SCHEDULE** and **LOGDATA** operators disabled and outputs the first set of print and plot variables for this run. If any of the dimensions or names of the print or plot variables has changed since the last run then the affected procedure is recompiled and **GO** is executed again.

Then the state equations in the DYNAMIC section are integrated from their current state until either the HALT flag becomes true, or there is a user interrupt. The TERMINAL section of the model is then executed before **GO** returns a string indicating the duration of the run.

If **_sgkey** is non−zero, a user interrupt can be initiated by pressing the **H** key.

If **GO** is called from within another procedure and any of the SimGauss procedures need to be recompiled, then **GO** will return to the GAUSS command mode at the end of the run and not to the procedure which called it. Also, in this case, the *runtime* string will not be returned.

See Also

START, stoptime, HALT _sgkey

HALT

Purpose

Sets or tests the HALT flag.

Format

halt_flg = **HALT(** *test* **);**

Inputs

test	any legal expression that returns a scalar. The HALT flag is set if the result is non−zero.

Outputs

halt_flg	scalar, the current setting of SimGauss' HALT flag.

Remarks

When the HALT flag is set the simulation run terminates after the next event, usually a data logging event. The HALT flag cannot be re−set within the run.

To terminate the run at a particular time use set the **stoptime** variable.

Example

```
call HALT(x>0.5);
```

In this example the HALT flag will be set and the simulation will terminate when the variable, **x** exceeds 0.5.

```
if HALT(0);
    ...
endif;
```

This is an example of testing the HALT flag and executing a code block if it is set.

See Also

stoptime, TERMINAL

i_t

Purpose

Sets the initial value of time.

Format

i_t	scalar, the initial value of time.

Remarks

When **START** is used to run the model, **i_t** sets the initial value of time. The default value is 0. **REINIT** stores the current time in this variable.

i_t is not overwritten when the model is recompiled unless it is explicitly defined in the model code.

Globals

i_t

See Also

stoptime

IMPLICIT

Purpose

Solves algebraic equations using Wegstein's iteration method.

Format

{ *x* , *errflg* **} = IMPLICIT(** *&fn_x* , *xz* , *maxerr* , *maxiter* **);**

Inputs

&fn_x	the address of a function or procedure which calculates x given a value for x. The function must have only one input argument, x, and one return, x.
xz	Nx1 vector, the initial value for x.
maxerr	Nx1 vector or scalar, the maximum error allowed.
maxiter	scalar, the maximum number of iterations allowed.

Outputs

x	Nx1 vector, the result of the iterations.
errflg	Nx1 vector, 0 if convergence obtained for that element of *x*. $x_i - fn\_x(x_i)$ if that element of *x* has an error greater than *maxerr* after *maxiter* iterations.

Remarks

IMPLICIT solves algebraic equations using the Wegstein's iteration method. The set of algebraic equations are contained in the procedure or function, *fn_x*. This function is iterated until either

$abs(x_i - fn\_x(x_i)) \leq abs(x_i .* maxerr)$ and
$abs(x_i - x_{i-1}) \leq abs(x_i .* maxerr)$ for $abs(x_i) > 1$
OR
$abs(x_i - fn\_x(x_i)) \leq abs(maxerr)$ and
$abs(x_i - x_{i-1}) \leq abs(maxerr)$ for $abs(x_i) \leq 1$

To maintain the vector capability of SimGauss the function or procedure, *fn_x(x)*, must be able to accept a column vector for *x* and return a column vector of the same size. The function or procedure can access any other model variable necessary to solve for *x*, but it should not modify any of them.

WARNING : Use of **IMPLICIT** is costly in time. It is worth the effort to solve the algebraic equations explicitly if possible. eg.

```
y = x − k .* y
```

can be expressed as

```
y = x ./ (1+k)
```

Also the convergence depends on the function and the starting value (See the Technical Notes below).

Often the last result is a good value for *xz* for the next call to **IMPLICIT**.

The *errflg* can be tested after the call to **IMPLICIT** to check on convergence. eg.

```
if not(0==errflg)
   print "IMPLICIT failed to converge!";
   print
   " The errors of non−convergent elements are";
   print (errflg);
   end;
endif;
```

Example

Solve 1.5*x − tan(x) = 0.1

```
/* define procedure for evaluating x given x */
Proc(1) = test(x);
     retp( (0.1 + tan(x)) ./ 1.5 );
endp;

{x, errflg} = IMPLICIT(0, 1e−6, 5);
x
     0.20592170
errflg
     0.00000000
```

Source

sg_utils.src

Globals

None

Technical Notes

The iterations may not converge because

i) *maxiter* is too small

ii) the starting value, *xz*, is too far away from the solution

iii) the *maxerr* is too small compared with the roundoff errors

iv) the solution value has a multiplicity greater than one

The iterations will also fail if

i) the secant has a slope of 1, either exactly or due to roundoff errors

ii) $x_i = x_{i-1}$ and x_i */= fn_x(*x_i*)* to machine accuracy. This can be caused by roundoff error or a very steep slope of the secant.

References

Numerical Methods for High Speed Computers by G.N. Lance, Iliffe, London, 1960, pp 134−138.

INITIAL

Purpose

Indicates the start of the model's INITIAL section.

Format

INITIAL;

Remarks

The SimGauss translator puts the code from the model's INITIAL section into a procedure called **SGP_INIT**. This procedure is then **CALL**ed to calculate the model's initial conditions each time **START** is used to run the model.

Technical Notes

Since the model code in the INITIAL section is put into a separate procedure, you cannot branch out of this section using **GOTO** statements and procedure definitions are not allowed in this section. Also **IF** and **DO** statements must not extend outside this section.

See Also

END_INIT, DYNAMIC, DISCRETE, DATALOG, TERMINAL

INITSTP

Purpose

Sets or displays the initial integration step size.

OR

Returns the current integration step size.

Format

initstep = **INITSTP(** *initstep* **);**

Inputs

initstep	scalar, the desired initial integration step size OR −1. If *initstep* is positive and less than SimGauss' epsilon, the current step size is not changed. If *initstep* is −1, the current integration step size is returned.

Outputs

initstep	scalar, the previous initial integration step size OR the current integration step size.

Remarks

The default initial step size is set by the **DYNAMIC** statement.

For fixed step size algorithms **INITSTP** sets the step size. Calling **INITSTP** to change the initial step size during a run will change the step size for fixed step size algorithms from the next integration step. Changing to a fixed step size algorithm will set the step size to **INITSTP**.

For variable step size algorithms **INITSTP** is only used when the run is **START**ed. Changing to a variable step size algorithm during a run will leave the current step size unchanged.

Example

```
step = initstp(−1);
```

This example returns the current integration step size. Add **step** to the print or plot variables see how the step size is changing.

See Also

MAXT, MINT, MAXORD, SETEPS, DYNAMIC

INTALG

Purpose

Selects or displays the integration algorithm.

Format

INTALG *algname;*
INTALG;

Inputs

algname name of the required integration algorithm.

The available algorithms are :−

euler	for Euler's first order algorithm (fixed step size)
rk2	for 2nd order Runge−Kutta (fixed step size)
rk4	for 4th order Runge−Kutta (fixed step size)
rkf4	for 2nd/3rd order Runge−Kutta−Fehlberg (variable step size)
rkf4	for 4th/5th order Runge−Kutta−Fehlberg (variable step size)
rbs	for Richardson−Bulirsch−Stoer (variable step size / variable order)
am	for Adams−Moulton predictor−corrector (variable step size / variable order)
gear	for Gear's stiff algorithm (variable step size / variable order)

If the *algname* is missing or not one of the valid algorithms then the list of available algorithms and the name of the current integration algorithm will be displayed.

When SimGauss starts up the default algorithm is 4th order Runge−Kutta.

Remarks

Eight integration algorithms are provided, three fixed step size algorithms, two variable step size algorithms and three variable step size, variable order algorithms. These are more fully described in Chapter 6 of the SimGauss Tutorial, 'Integration Algorithms'.

The recommended algorithms for most physical simulations are the variable step size 2nd/3rd or 4th/5th order Runge−Kutta−Fehlberg algorithms. These algorithms automatically vary the step size to maintain the specified state error per step. The 4th/5th order algorithm is for use with smoother simulations. That is ones which do not have many discontinuities, deadbands, table lookups, etc.

However the efficiency of variable step size methods relies on being able to take large steps over smooth areas of the simulation. Since the maximum step that the integration algorithm can take is limited by the time to the next event, if you want the finest data logging intervals without limiting the step size, then use **DATASTEP(−1)** to datalog at each integration step.

Euler's method is not recommended for any real application. It is included for pedogological purposes and for error checking.

For fixed step size algorithms **INITSTP** sets the step size. Calling **INITSTP** to change the initial step size during a run will change the step size for fixed step size algorithms from the next integration step. Calling **INTALG** to change to a fixed step size algorithm will set the step size to **INITSTP**.

For variable step size algorithms **INITSTP** is only used when the run is **START**ed. Calling **INTALG** to change to a variable step size algorithm will leave the current step size unchanged.

At the end of each integration step the DYNAMIC section is executed again to update the derivatives at the current state in preparation for data logging or the next integration step.

Example

```
intalg rkf4
```

This example selects the 4th/5th Runge−Kutta−Fehlberg variable step size algorithm. The current step size is unchanged.

Globals

errmint

See Also

INITSTP, MINT, MAXT, errmint, MERROR, RERROR

KEEPPLOT

Purpose

Prevents the plot matrix from being overwritten each run.

Format

KEEPPLOT(*boolean* **);**

Inputs

boolean	scalar, if zero the plot matrix is overwritten each run. This is the default when SimGauss is loaded. If *boolean* is non−zero, the plot matrix is kept and the next run appends to it.

Remarks

The plot matrix is only kept if the last call to **KEEPPLOT** had a non−zero argument and the names and dimensions of the plot variables have not changed since the last run.

Example

```
keepplot(1);
```

The next run, usually a continuation of the last run using the **GO** command, will append its plot values to the end of the current plot matrix.

See Also

PLOTVARS, SGPLTVAR, SGXY, LOGDATA, GO, START

LIMINT

Purpose

Limits a state variable.

Format

d_*x* = **LIMINT(** *x* , *x_dot* , *botlim* , *toplim* **);**

Inputs

x	the state variable to be limited. Must be a variable name not a constant or expression.
x_dot	Nx1 vector expression for the state derivative.
botlim	Nx1 vector or scalar, the bottom limit.
toplim	Nx1 vector or scalar, the top limit.

Outputs

d_*x*	the state derivative variable. The state defined by this derivative name must be the same state as the x input.

Remarks

The left hand side of this statement must be in the form **d_***state*, where *state* is the state variable name used for the *x* input argument.

The **LIMINT** statement can only be used in the DYNAMIC section of the model.

The state *x* will always penetrate the limit. Use **BOUND** to create an auxiliary variable which will not exceed the limits.

Example

```
d_pos = limint( pos, velocity, 5, −1 );
b_pos = bound( pos, 5, −1 );
```

In this example the state variable **pos** is limited to 5 and −1. **pos** is the integral of **velocity**. **b_pos** is the bounded auxiliary variable.

Source

sg_utils.src

Globals

The state, *x*, and the derivative, **d_***x*.

Technical Notes

Before the integrator can be limited, the state must exceed the bound. The smaller the integration step size and the lower the velocity, the less the state will exceed the bound.

To obtain a variable that never exceeds the bounds use **BOUND** to create an auxiliary variable. Do not assign the output of **BOUND** to the state variable itself as all the state variables are overwritten by the integration procedure before each call to the DYNAMIC section of the model.

See Also

BOUND, DYNAMIC

LOGDATA

Purpose

Forces data logging.

Format

LOGDATA(boolean **);**

Inputs

boolean	scalar, if non−zero the print variables are also displayed. This is a forced print regardless of the value of **PRTINT**. The format of the print variables is controlled by the GAUSS **FORMAT** command. In any case the DATALOG section is called and the plot variables saved.

Remarks

Data logging takes place at each data logging interval and on the termination of the run. Finer detail can be recorded by using this procedure.

It is often used to force a data logging just before and just after an event has occurred so that a subsequent plot shows the step change in the variable.

Example

```
logdata(1)
```

This statement will cause the DATALOG section to be executed, then the plot variables saved and the print variables displayed.

See Also

DATALOG, PRTVARS, PLOTVARS

MAXORD

Purpose

Sets or displays the maximum order for the AM and GEAR integration algorithms.
  OR
Returns the current order.

Format

maxorder = **MAXORD(** *maxorder* **);**

Inputs

maxorder	scalar, the desired maximum order for the AM and GEAR integration algorithms OR −1. If *maxorder* equals zero when rounded, the maximum order is not changed. If *maxorder* equals −1 when rounded, the current integration order is returned.

Outputs

maxorder	scalar, the previous maximum order OR the current integration order.

Remarks

The initial maximum order is 7.
The maximum order allowed for AM is 7 and for GEAR is 6.

Example

```
ord = maxord(-1);
```

This example returns the current integration order. Add **ord** to the print or plot variables see how the integration order is changing.

See Also

INTALG

MAXT

Purpose

Sets or displays the maximum integration step size.

Format

maxstep **= MAXT(** *maxstep* **);**

Inputs

maxstep	scalar, the desired maximum integration step size. If *maxstep* is less than SimGauss' epsilon the current step size is not changed.

Outputs

maxstep	scalar, the previous maximum integration step size.

Remarks

The maximum size of the next integration step is determined by
minc(maxt(0) | time_to_next_event | step_size **)**.

The default **MAXT** is 1e300.

Example

```
max_tmp = maxt(maxt(0)*2);
```

This example saves the previous maximum step size in the variable **max_tmp** and then doubles the maximum step size.

See Also

MINT, SETEPS, DYNAMIC

MERROR

Purpose

Sets the maximum errors for the states and controls.

Format

m_error = **MERROR(** *m_error* **);**

Inputs

m_error	scalar, the default maximum error. If *m_error* is zero, the current default maximum error is not changed. When SimGauss is loaded the default maximum error is 1.0E−4.

Outputs

m_error	scalar, the previous default maximum error.

Remarks

This command only affects variable step size integration algorithms and **SGLIN**.

The default maximum error is used for all states and controls which do not have an individual maximum error assigned, i.e. the associated **m_***state* or **m_***control* variable does not exist or is zero. It is also used as the maximum error for all elements of state and control vectors whose assigned maximum error is zero.

MERROR can be executed at any time to set the default maximum error and transfer the **m_***state* variables to the maximum error vector for use by the variable step size integration algorithms. Any changes made to the **m_***state* variables will not be recognised by the integration algorithms until **MERROR** has been executed.

MERROR(0) is automatically called each time the **START**, **GO** or **SGLIN** command is used.

The **m_***state* and **m_***control* variables can be either scalars or Nx1 vectors. If they are scalars, the scalar value is used as the maximum error for all elements of the associated state or control vector. If they are Nx1 vectors, the dimensions must match the dimensions of the associated vector. I.e. if **m_x** is a scalar 0 then the default maximum error set by **MERROR** is used as the maximum error for all the elements of **x**.

Example

```
merr_tmp = merror(merror(0)/2);
```

In this example the previous default maximum error is saved in **merr_tmp** and then halved and all the **m_***state* variables are transferred to the integration algorithm's maximum error vector. Any elements of this vector which are 0 are assigned the default maximum error.

Globals

m_*state*, **m_***control*

Technical Notes

Variable step size algorithms incorporate an equation which gives an estimate of the integration truncation error for each state at each step. If this error estimate for any state exceeds the *desired_error* for that state, then the integration algorithm reduces the step size. It then retakes the step again using the smaller step size and recalculates the error estimate. This process continues until either the step succeeds (i.e. the error estimates are less than the *desired_errors*) or the minimum integration step is reached.

The *desired_error* is calculated from the specified **MERROR**s and **RERROR**s for each state. The *desired_error* for each state is given by

desired_error = max(m_*state*, r_*state* * max(abs(*state*)))

where max(abs(*state*)) is the maximum of the absolute value of that state so far, not the current value.

SGLIN also uses the maximum errors to set the minimum absolute perturbation of each state and control element.

See Also

SGLIN, RERROR, MINT, INTALG, errmint

MINT

Purpose

Sets or displays the minimum integration step size.

Format

minstep = **MINT(** *minstep* **);**

Inputs

minstep	scalar, the desired minimum integration step size. If minstep is less than SimGauss' epsilon the current step size is not changed.

Outputs

minstep	scalar, the previous minimum integration step size.

Remarks

The minimum step size sets the limit on the smallest step a variable step size integration algorithm may take. If the algorithm needs to reduce the step size below *minstep* in order to satisfy the error criterion, it sets the flag **errmint** and calls the DYNAMIC procedure, **SGP_DYN**. If **errmint** is still set when **SGP_DYN** returns, the simulation terminates with a message. The minimum step size can be changed during a run using the **MINT** command.

If the time left to the next event is greater than SimGauss' epsilon but less than *maxstep*, a short step is taken to bring the time up to the next event. This step may be less than *minstep*.

If **MINT** has not been used to set the *minstep*, the initial *minstep* is set to **INITSTP(0)/100**.

Example

```
min_tmp = mint(mint(0)/2);
```

This example saves the previous minimum step size in **min_tmp** and then reduces it by half.

See Also

MAXT, SETEPS, INITSTP

OBSERV

Purpose

Sets the observable variables for SGLIN.

Format

OBSERV *variables***;**
OBSERV clearall *variables***;**
OBSERV;

Inputs

variables	a list of the global numeric column vectors, separated by spaces, to be added to the existing observable variables. If the list contains the name **clearall**, then all the current observable variables associated with the model will be cleared. The variables following **clearall** will then be added to list of observable variables. If the list is empty the current observable variables will be displayed. Indexing of elements is not allowed. The entire vector is observable.

Remarks

If there are no observable variables then **SGLIN** will return missings for the C and D matrices. Initially there are no observable variables.

The order of the observable variables displayed by **OBSERV** is the order of the rows of the C and D matrices returned by **SGLIN**. Model states may be observable variables also.

Example

```
observ x v;
```

This example adds the variables **x** and **v** to the current observable variables.

See Also

SGLIN, STATES, CONTROLS

PHYLIMIT

Purpose

Limits the position of a second order system.

Format

d_*vel* **= PHYLIMIT(** *pos , vel , accel , stiff , damping , botlim ,* **);**
d_*pos* = *vel* **;**

Inputs

pos	the position state variable to be limited.
vel	the velocity state variable.
accel	Nx1 vector expression for the system acceleration.
stiff	Nx1 vector or scalar, the additional spring stiffness when in limit.
damping	Nx1 vector or scalar, the additional velocity damping when in limit.
botlim	Nx1 vector or scalar, the bottom position limits.
toplim	Nx1 vector or scalar, the top position limits.

Outputs

d_*vel*	the velocity derivative variable. The state defined by this derivative must be the same as the *vel* input.
d_*pos*	the position derivative variable. The state defined by this derivative must be the same as the *pos* input.

Remarks

The Physical Limit function consists of two derivative statements. The first, **PHYLIMIT**, defines the velocity state, *vel,* while the second defines the position state, *pos*.

This simulates a second order system which has physical stops that limit its movement. A particular case is that of a mass/spring/damper system.

The equation of motion for this system is

$$m \times \frac{d^2x}{dt^2} + r \times \frac{dx}{dt} + k \times x = f(t)$$

where

> *m* is the mass
> *r* is the damping and
> *k* is the spring constant and
> the mass is constrained to move between *botlim* and *toplim*,
> that is *botlim x toplim*

To simulate this system, rewrite the equation of motion as

$$\frac{dv}{dt} = accel$$

where

$$accel = \frac{f(t) - r \times v - k \times x}{m}$$

and

$$v = \frac{dx}{dt}$$

When the mass reaches a limit the spring stiffness, *k*, and the damping, *r*, are increased. This corresponds to what happens physically and the equation of motion in the limits becomes

$$m \times \frac{d^2x}{dt^2} + (r + damping) \times \frac{dx}{dt} + (k + stiff) \times x = f(t)$$

The additional stiffness, *stiff*, determines how far the mass penetrates the limits, for a give velocity. The addition damping, *damping*, determines how much energy is dissipated in the limits. Adjusting these two values allows a wide variety of systems to be modelled.

A variable step size integration algorithm, such as **RKF2** or **RKF4**, should be used for the periods the system is at the limits as the higher stiffness and damping usually require much smaller integration steps.

Example

```
D_v = phylimit(x, v, (K.*(x0−x)−R.*v)./M − g,
               5000, 40, 0, 100);
D_x = v;
```

In this example the force, *f(t)*, on the mass/spring/system is due to gravity. 5000 is added to the spring stiffness and 40 added to the damping when the mass reaches ground level. This make the response quite bouncey but the damping causes a noticeable loss of energy on each bounce. The top limit is set out of the way at 100.

Source

sg_utils.src

Globals

The states, *pos* and *vel*, and the derivatives, **d_***pos* and **d_***vel*.

See Also

LIMINT, BOUND

PLOTVARS

Purpose

Specifies the variables to be saved for plotting.

OR

Displays the currently specified plot variables.

Format

PLOTVARS *varnames***;**
PLOTVARS clearall *varnames***;**
PLOTVARS;

Inputs

varnames	a list of names, separated by spaces, of the vectors or scalars to be saved at each data logging interval for plotting later. These variable names will be added to those variables already being saved. If the list contains the name **clearall**, all the current plot variables will be removed. Then the variables following **clearall** will be added to the list of plot variables. If the list is empty then the current plot variables will be displayed and recompiled if their dimensions have changed. Indexing of elements is not allowed. The entire vector is saved. Matrices cannot be saved directly.

Remarks

By default when SimGauss is first loaded no plot variables are specified.

Although the entire vector is saved in the plot matrix, individual elements can be retrieved by **SGPLTVAR, SGXY** etc. To save matrices assign each row or column to a variable and add those variables to the existing plot variables.

The **PLOTVARS** command should only be called from the GAUSS command mode as it terminates by **RUN**ning the file **s_plot.sg** if the variables or dimensions have changed.

Example

```
» plotvars t x;

» plotvars;
The current SimGauss plot variables are :-
    T          1 x 1
    X          1 x 1
*********************

» plotvars clearall;
Simgauss : Clearing plot variables.
```

Technical Notes

The plot variables are saved in the SimGauss plot matrix. There is one row for each time data is logged. Using a matrix is faster than storing the data in a file but limits the number of points that can be saved. If the storage limits of the matrix are exceeded during the simulation, then SimGauss will stop with the appropriate GAUSS error message. The currently active procedure will be **S_PLOT**.

See Also

SGPLTVAR, SGXY, PRTVARS

PRTINT

Purpose

Specifies the number of data logging intervals between printing the variables specified by **PRTVARS**.

Format

num = **PRTINT(** *num* **);**

Inputs

num	scalar, the number of data logging intervals between prints. If *num* is zero the current setting is not changed.

Outputs

num	scalar, the previous number of data logging intervals between prints.

Remarks

This command only effects the displaying of the print variables, the plot variables are always saved at each data logging interval.

The GAUSS **FORMAT** command controls the display format of the print variables.

Example

```
pint_tmp = prtint(prtint(0)*2);
```

This statement saves the previous print interval in **pint_tmp** and then doubles the time between displaying the print variables.

See Also

PRTVARS, PLOTVARS

PRTVARS

Purpose

Specifies the variables to be printed. OR Displays the currently specified print variables.

Format

PRTVARS *varnames***;**
PRTVARS clearall *varnames***;**
PRTVARS

Inputs

varnames	a list of names, separated by spaces, of the global variables to be printed at the data logging intervals. These variables will be added to those variables already being printed.
	If the list contains the name **clearall**, all the current print variables will be removed. Then the variables following **clearall** will be added to the list of print variables.
	If the list is empty then the current print variables will be displayed and recompiled if their dimensions have changed.
	Indexing of elements is not allowed. The entire vector or matrix is printed.

Remarks

By default, when SimGauss is first loaded, the time, **t**, is the only print variable. Initially the print variables are printed at every data logging interval. This print rate can be reduced by using **PRTINT**.

Turning the screen off using the GAUSS command **SCREEN OFF** will save a significant amount of run time. The GAUSS **OUTPUT** command can be used to capture the print variables in a file.

The **PRTVARS** procedure should only be called from the GAUSS command mode as it terminates by **RUN**ning the file **s_print.sg** if the variables or the dimensions have changed.

The GAUSS **FORMAT** command controls the display format of the print variables.

Use **PRTVARS clearall** to suppress all SimGauss output when running a model from within another program, such as **OPTMUM**.

Example

```
» prtvars; The current SimGauss print variables are
:−
    T           1 x 1
*********************
» prtvars x;
» prtvars;
The current SimGauss print variables are :−
    T           1 x 1
    X           1 x 1
*********************
» prtvars clearall;
SimGauss : Clearing print variables.
```

See Also

PRTINT, PLOTVARS

QUANTIZE

Purpose

Quantizes the input.

Format

y = **QUANTIZE(** *x* , *delta_x* **);**

Inputs

x	Nx1 vector, the continuous input.
delta_x	Nx1 vector or scalar, the quantization steps.

Outputs

y	Nx1 vector, the quantized output.

Remarks

The quantization is centred around 0 and the steps in the output occur at 0.5*delta_x. The output, *y*, is calculated from

```
y = round(x ./ delta_x) .* delta_x
```

Example

```
y = quantize(−0.6|−0.4|0|0.4|0.6 , 1)
y
  −1
   0
   0
   0
   1
```

In this example the input vector is quantized in steps of 1.

Source

sg_utils.src

Globals

None

REINIT

Purpose

Stores the current state vectors in the initial condition variables.

Format

REINIT;

Remarks

The current values of the states are written into the **i_***state* variables. The next **START** command will then run the model from this state.

Note however that the **DELAY** matrices and the current output of any **BACKLASH** and **IMPLICIT** statements are not saved.

The **REINIT** statement is best used to save the state of a model after it has been trimmed using **SGTRIM** or when it has reached the desired operating point to which disturbances will then be applied.

REINIT sets **i_t** since the time, **t**, is a state of the model. This means the next **START** command will start from the current time. **i_t** can be changed to another value if you wish. **i_t** is not overwritten when the model is recompiled unless it is explicitly defined in the model code.

REINIT sets **INITSTP** to the current step size.

An alternative method of saving the current state for later use is the GAUSS command **SAVEALL**. This saves the entire contents of the workspace to a file which can be reloaded later using the **RUN** command.

See Also

START, INITIAL

RERROR

Purpose

Sets the relative errors for the states and controls.

Format

r_error = **RERROR(** *r_error* **);**

Inputs

r_error	scalar, the default relative error. If *r_error* is zero, the current default relative error is not changed. When SimGauss is first loaded the default relative error is 1.0E−4.

Outputs

r_error	scalar, the previous default relative error.

Remarks

This command only affects variable step size integration algorithms and **SGLIN**.

The default relative error is used for all states and controls which do not have an individual relative error assigned, i.e. the associated **r_***state* or **r_***control* variable does not exist. It is also used as the relative error for all elements of state or control vectors whose assigned relative error is zero.

RERROR can be executed at any time to set the default relative error and transfer the **r_***state* variables to the relative error vector for use by variable step size integration algorithms. Any changes made to the **r_***state* variables will not be recognised by the integration algorithms until **RERROR** has been executed.

RERROR(0) is automatically called each time the **START**, **GO** or **SGLIN** command is used.

The **r_***state* and **r_***control* variables can be either scalars or Nx1 vectors. If they are scalars, the scalar value is used as the relative error for all elements of the associated state or control vector. If they are Nx1 vectors, the dimensions must match the dimensions of the associated vector. I.e. if **r_x** is a scalar 0 then the default relative error set by **RERROR** is used as the relative error for all the elements of **x**.

Example

```
rerr_tmp = rerror(rerror(0)/2);
```

In this example the previous default relative error is saved in **rerr_tmp** and it is then halved and all the **r_***state* variables are transferred to the integration algorithm's relative error vector. Any elements of this vector which are zero are assigned the default relative error.

Globals

r_*state*, **r_***control*

Technical Notes

Variable step size algorithms incorporate an equation which gives an estimate of the integration truncation error for each state at each step. If this error estimate for any state exceeds the *desired_error* for that state, then the integration algorithm reduces the step size. It then retakes the step again using the smaller step size and recalculates the error estimate. This process continues until either the step succeeds (i.e. the error estimates are less than the *desired_errors*) or the minimum integration step is reached.

The *desired_error* is calculated from the specified **MERROR**s and **RERROR**s for each state. The *desired_error* for each state is given by

desired_error = max(m_*state*, r_*state* * max(abs(*state*)))

where max(abs(*state*)) is the maximum of the absolute value of that state so far, not the current value.

SGLIN also uses the relative errors to set the relative perturbation of each state and control element.

See Also

SGLIN, MERROR, MINT, INTALG, errmint

SCHEDULE

Purpose

Schedules a DISCRETE section for execution at a later time.

Format

SCHEDULE(*blkname , blktime* **);**

Inputs

blkname	string, contains the name of the DISCRETE section to be added to the simulation's event queue. Leading and trailing blanks are not allowed.
blktime	scalar, the time at which the section is to be executed. It must be greater than or equal to the current time.

Remarks

Events with times earlier than the current time, **t**, by more than

epsilon * | t | + *epsilon* / 2

cannot be scheduled as the event time is considered to have been passed.

SCHEDULE should only be used within the model as the simulation event queue is cleared each time the **START** command is used to run the model. Use **ADDEVENT** to schedule user event keywords from the GAUSS command mode.

The **SCHEDULE** statement does nothing when executed from the GAUSS command mode.

Example

```
schedule("sample",t+0.1);
```

This example **SCHEDULE**s the DISCRETE section **sample** to execute in 0.1 time units from the current time.

See Also

DISCRETE, EVENTQUE, ADDEVENT, CLREVENT

SETEPS

Purpose

Sets the SimGauss epsilon.

Format

sgeps = **SETEPS(** *sgeps* **);**

Inputs

sgeps	scalar, the desired SimGauss epsilon. If zero the current value is not changed. If *sgeps* is greater than zero but less than 20 time the machine epsilon, eps, then SimGauss' epsilon is set to 20*eps. 20*eps is approximately 4.4e−15 for xx87 co−processors.

Outputs

sgeps	scalar, the previous value of SimGauss' epsilon.

Remarks

The SimGauss epsilon is used for determining if two times are the same.

The integration stops to execute the next event, such as data logging, if the current time is within half epsilon of the event time. Also events with times earlier than the current time, **t**, by more than

epsilon * | t | + epsilon / 2

cannot be **SCHEDULE**d as the event time is considered to have been passed.

For **87 co−processors the default epsilon is approximately 4.4E−15. This means that in a simulation spanning a 100 years, events within 1uS (1E−6 seconds) of each other are treated as occurring at different times.

Epsilon is also used in some procedures to test for arguments equal to zero.

Example

```
» seteps(1.0e−6);
  4.4408921E−015
```

This sets the SimGauss epsilon to 1.0E−6. This means times within 0.0001% of each other will be treated as the same time.

SGDEBUG

Purpose

Turns the SimGauss debugger on or off.

Format

SGDEBUG(*boolean* **);**

Inputs

boolean	scalar, if non−zero the debugging will be turned on. If zero, it will be turned off.

Remarks

When the SimGauss debugging is turned on SimGauss writes messages to the screen about the current integration algorithm and step size, the state of the simulation event queue, the time to the next event, etc. Exactly what information is provided will vary in different versions of SimGauss.

SimGauss turns the debugging off quietly each time it compiles a model.

For small integration steps or long runs, large volumes of output are generated. It is usually best to only turn the debugging on when necessary.

Example

```
sgdebug(t>1.5 and t<1.6);
```

In this example the debugging is only turned on when the simulation time is between 1.5 and 1.6. This statement is best placed in the DYNAMIC section of the model.

SGLIN

Purpose

Returns the state−space matrices of the model linearize around the current state.

Format

{ *a,b,c,d* **} = SGLIN(** *non−lin* **);**

Inputs

non−lin	The non−linearity measure above which warning messages are issued. If *non−lin* is 0 the default value of 0.1 is used.

Outputs

a b c d The state space matrices of the linearized model. Where

D_*states* = A * *states* + B * *controls*
*observ = C * states + D * controls*

Remarks

The order of the columns of A and B are order of the state variables as displayed by **STATES**. The order of the columns of B and D is given by **CONTROLS**. While the order of the rows of C and D is given by **OBSERV**. If there are no control variables specified then the B and D matrices will be missings. If there are no observable variables specified then the C and D matrices will be missings.

A central difference method is used to calculate the A,B,C and D matrices. Each of the elements of the state and control variables is perturbed both positively and negatively and the overall slope used to calculate the elements of the A, B, C and D matrices. The size of the perturbation of each element of the states and controls is controlled by maximum and relative error settings for that variable. E.g. for the variable x the positive and negative perturbation is given by

```
MAXC(M_X | ABS(R_X*X))
```

If **m_x** or **r_x** are undefined then the default values set by **MERROR** or **RERROR** are used. The initial default values are 1.0E−4

Only the DYNAMIC section of the model is used to calculate the A,B,C and D matrices, so the control variables must effect an equation in the DYNAMIC section and the observable variables must be calculated by the DYNAMIC section of the model. A column of zeros in B indicates that control variable has no effect on the derivatives when the DYNAMIC section of the model is executed. A row of zeros in C indicates that observable variable is not effected by any of the states when the DYNAMIC section of the model is executed.

SGLIN initially checks for repeatablity of the calculation of the derivatives and observable variables and issues a warning message if any of them differs in two successive calculations of the DYNAMIC section of the model. It also checks the linearity of the model at the current state by comparing the results obtained from the positive and negative perturbations with the initial values of the derivatives and observable variables. The initial values of the derivatives and observable variables should be the average of the perturbed results.

The difference between the initial values and the average of the positive and negative perturbed results is scaled by the overall difference between the perturbed results to produce a non−dimensional measure of non−linearity. A value of 0 indicates a linear system. A value of 0.5 commonly arises from a variable at a limit. If the measure exceeds the input argument value then a warning message is printed for that element of the matrix.

Example

```
call merror(0.1);
call rerror(0.01);
{a,b,c,d} = sglin(0.2);
```

This example sets the default minimum perturbation in each direction to 0.1 and for large values of the state and control elements the default perturbation will be 1%. The linearized model is then calculated with warning messages about non−linearity suppressed for non−linearity measures below 0.2. Those states and controls that have their own **m_** and **r_** variables defined will use those values instead of the defaults.

See Also

STATES, CONTROLS, OBSERV, MERROR, RERROR

SGLOADP

Purpose

Load the plot matrix from a GAUSS data set.

Format

SGLOADP *dataset*;
SGLOADP

Inputs

dataset	the filename of the GAUSS data set. If *dataset* is not given, the data set TEMP.DHT, TEMP.DAT will be loaded. TEMP.FST, if it exists, will also be loaded.

Remarks

This command is used to load SimGauss' plot matrix from a GAUSS data set. On systems with limited memory the Publication Quality Graphics may not run with SimGauss loaded. To overcome this memory limitation save the plot matrix using **SGSAVEP**, load the Publication Quality Graphics and then use **SGLOADP** to reload the plot matrix. The SimGauss plot commands **SGXY, SGLOGX, SGLOGY** and **SGLOGLOG** can then be used to plot the results.

GAUSS data sets saved by other programs, which could be loaded using **LOADD**, can also be loaded using **SGLOADP**. Using **SGLOADP** allows the SimGauss plot procedures to be used to plot a data set's contents and **SGPLTVAR** can be used to select various individual elements. The string in the .FST file, if it exists, is used to set the **_pdate** string for the plots.

Example

```
library simgauss;
sgloadp spring;
```

In this example the GAUSS data set SPRING.DHT, SPRING.DAT is loaded into the SimGauss plot matrix to allow selection of variables via **SGPLTVAR** or plotting via **SGXY** etc.

Source

sgloadp.src

Globals

SGLOADP(), LOADD(). All other globals are referenced by **VARGET** and **VARPUT**.

Technical Notes

The files produced by **SGSAVEP**, for loading with **SGLOADP**, are in standard GAUSS data set 8 byte format, except that for vectors the variable name is repeated in the .DHT file to identify the columns associated with that vector. **SGLOADP** not only reloads the plot matrix but also sets up the plot variable names and dimensions in a form suitable for the SimGauss plotting procedures. The file **sgloadp.src** illustrates a method of recovering vectors from the data set created by **SGSAVEP**. However it is often convenient just to use **SGLOADP** to reload the entire plot matrix and then use **SGPLTVAR** to extract the required elements.

Only data sets which can be loaded with **LOADD** can be loaded with **SGLOADP**.

See Also

SGSAVEP, SGXY, SGLOGX, SGLOGY, SGLOGLOG, SGPLTVAR, PLOTVARS, KEEPPLOT

SGLOGLOG, SGLOGX, SGLOGY, SGXY

Purpose

LogLog, LogX, LogY, XY plot of variables from the plot matrix.

Format

SGLOGLOG *xvars , yvars***;**
SGLOGX *xvars , yvars***;**
SGLOGY *xvars , yvars***;**
SGX *xvars , yvars* **;**

Inputs

xvars	a list, separated by spaces, of the X axis plot variable names, optionally with numerical indices.
yvars	a list, separated by spaces, of the Y axis plot variable names, optionally with numerical indices.

Remarks

These plotting KEYWORDS use the Publication Quality Graphics routines do the plotting. Both the **simgauss** and the **pgraph** libraries must be active and the Publication Quality Graphics must have been installed.

These commands also set the X and Y axis labels to the strings *xvars* and *yvars* respectively. As these commands are GAUSS KEYWORDs the argument is converted to upper case unless *xvars , yvars* is enclosed in inverted commas. These commands use **SGPLTVAR** to extract the variables from the plot matrix.

Only numeric indices are allowed, separated by spaces. Expressions or variables are not allowed for indices.

If *xvars* or *yvars* contains multiple elements, then symbols and lines will be displayed on the plot. This is done by setting **_plctrl** to 1 to display symbols on the various curves. This can be reset by calling the graphics library command **GRAPHSET**.

SimGauss does not need to be in memory to use these procedures. See **SGLOADP** for how to load a plot file from disk.

Example

```
library simgauss, pgraph;
sgloglog "t,x[1 2]";
```

In this example the log of the variables **x[1]** and **x[2]** are plotted against the log of the time, **t**.

The axes' labels are set to **t** and **x[1 2]** because the string following **SGLOGLOG** was enclosed in inverted commas.

Source

sgloglog.src, sglogx.src, sglogy.src, sgxy.src

Globals

SGLOGLOG(), SGPLTVAR(), LOGLOG(), SGLOGX(), LOGX(), SGLOGY(), LOGY(), SGXY(), XY(). All other globals are referenced by **VARGET** and **VARPUT**.

See Also

SGPLTVAR, SGLOADP, SGSAVEP, PLOTVARS, KEEPPLOT

SGPLTVAR

Purpose

Extracts variables from the plot matrix.

OR

Displays the plot variables stored in the plot matrix.

Format

y = **SGPLTVAR(** *vars* **);**

Inputs

vars string, the names of the plot variables, separated by spaces, optionally with numerical indices

OR

"SEQA"

A null string, "", or empty string, " ", will display the names of the plot variables saved in the plot matrix.

Outputs

y matrix, the columns selected by vars

OR

a column vector of indices if vars contains the string "SEQA"

If the input string is a null or empty, the plot variables are displayed and GAUSS reverts to command mode.

Remarks

Only numeric indices are allowed, separated by spaces. Expressions or variables are not allowed for indices.

The "SEQA" option is useful for plotting all the points of a variable against its row index. For variable step size algorithms using **DATASTEP(−1)** this shows the value at each step in the integration.

SimGauss does not need to be in memory to use this procedure. See **SGLOADP** for how to load a plot data set from disk.

_pdate is set to "\201SimGauss V2.0 Model *name* Compiled *date_time* Plotted ". This sets the date stamp at the top left of the Publication Quality Plots. This can be reset by calling the graphics library command **GRAPHSET**.

Since **SGXY**, **SGLOGX**, **SGLOGY** and **SGLOGLOG** all call this procedure these notes will also apply to those commands.

Example

```
library simgauss, pgraph;
xy(sgpltvar("t"),sgpltvar("x[1 2]"));
```

This example selects the columns which contain the values of **t**, **x[1]** and **x[2]** from the current plot matrix and passes the results to XY for plotting. The SimGauss command **SGXY** does a better job.

Source

sgpltvar.src

Globals

SGPLTVAR(). All other globals are referenced by **VARGET** and **VARPUT**.

See Also

SGLOADP, SGSAVEP, SGXY, SGLOGX, SGLOGY, SGLOGLOG, PLOTVARS, KEEPPLOT

SGSAVEP

Purpose

Save the plot matrix to a GAUSS data set.

Format

SGSAVEP *dataset***;**

Inputs

dataset	the filename of the GAUSS data set. If *dataset* is not given, the data set TEMP.DHT, TEMP.DAT will be created as well as TEMP.FST.

Remarks

This command is used to save SimGauss' plot matrix to a GAUSS data set and the model name to string file. On systems with limited memory the Publication Quality Graphics may not run with SimGauss loaded. To overcome this memory limitation save the plot matrix using **SGSAVEP**, load the Publication Quality Graphics and then use **SGLOADP** to reload the plot matrix. See **SGLOADP** for further details.

Example

```
sgsavep spring;
```

In this example the GAUSS data set SPRING.DHT, SPRING.DAT and SPRING.FST is created and the current SimGauss plot matrix and model name written to them.

Source

sgsavep.src

Globals

SGSAVEP(), SAVED(). All other globals are referenced by **VARGET** and **VARPUT**.

See Also

SGLOADP, SGXY, SGLOGX, SGLOGY, SGLOGLOG, SGPLTVAR, PLOTVARS, KEEPPLOT

SGTRIM

Purpose

Trim the model for steady state conditions.

Format

SGTRIM(0);
SGTRIM(*&fn* **);**

Inputs

&fn	The address of a procedure to executed at each iteration of the trimming process in addition to executing the DYNAMIC section of the model. This procedure must expect no arguments and return no results. If this address is 0 only the DYNAMIC section of the model is executed.

Remarks

The optional GAUSS Nonlinear Simultaneous Equations module is required to run **SGTRIM. NLSYS** is used to zero the derivatives of the model associated with the unfrozen states. Refer to the GAUSS APPLICATIONS manual for full details of the globals which control **NLSYS**. The **nlsys** library must be set prior to calling **SGTRIM**.

For models which use digital controllers implemented in DISCRETE blocks, the controller must be executed for each iteration of **NLSYS** if the correct steady state condition is to be found. The procedure address argument passed to **SGTRIM** allows this to be done (see the example below). If there is no controller or the controller is implemented in the DYNAMIC section of the model then **SGTRIM(0)** can be used.

If **SGTRIM** terminates with the error message
SimGauss ERROR in SGTRIM : Initial gradient matrix is singular.
The gradient matrix of the current unfrozen states will be displayed together with a list of the unfrozen states associated with each column. The rows of the gradient matrix are associated with the derivatives of the same states. A zero column in the gradient matrix indicates that the associated state has no effect of any of the derivatives and should be frozen using **FREEZE**. A zero row in the gradient matrix indicates that no states effect that derivative and the associated state should also be frozen.

After the model has been trimmed, **REINIT** can be used to save the trimmed states as the initial conditions. The other model variables will not be updated until the model is next run.

NLSYS allows for the scaling of the function arguments and results via use of the global variables **_nltypx** and **_nltypf**. Two procedures, which are not documented elsewhere, are provided to give access to suitable values for these variables, taking into account the frozen states. These global variables need to be updated each time additional states are frozen.

CRNT_ST takes no arguments and returns the current unfrozen states.

```
_nltypx = crnt_st;
```

DYN_FN(CRNT_ST) takes the current unfrozen states and returns the current values of the derivatives of the unfrozen states. It is these derivatives which **NLSYS** is trying to make zero.

```
_nltypf = dyn_fn(crnt_st);
```

Example

In this example the STATEMAT model from Section 5.2 of the SimGauss Tutorial will be trimmed (refer Section 5.2 for the model code). STATEMAT has a digital controller implemented by two DISCRETE sections, **sample** and **control**. To ensure the model's controller is executed every iteration of NLSYS, a procedure called **temp** will be created to call the **sample** and **sample** procedures created by the SimGauss translator from the DISCRETE sections. Note that these procedures must be called in the same order as they execute during the simulation.

```
library simgauss, pgraph, nlsys;
proc(0) = temp;
     sample;
     control;
endp;
sgtrim(&temp);
```

Source

sgtrim.src

See Also

REINIT, FREEZE, SGLIN

SIMGAUSS

Purpose

Translates and compiles a model.

Format

SIMGAUSS *model_name*

Inputs

model_name the name of the model to translate and compile.
If no name is given the program prompts for the name of the model.
If a file extension is given it must be **.sgm**

Remarks

If SimGauss is not already in the GAUSS workspace, the workspace is cleared and the **simgauss.csg.gcg** file is **RUN** to load SimGauss.

Pressing the **Enter** key in response to the model name prompt will terminate the **SIMGAUSS** command.

Typing a directory name including the final \ will list all the **.sgm** files on that directory. Use .\ to list the **.sgm** files on the current directory.

Refer to the section on Customizing SimGauss in the Introduction of the SimGauss Tutorial for examples of how **simgauss.csg.gcg** can be modified to include other procedures.

Example

See the SimGauss Tutorial for examples of this command and Chapter 9, 'Debugging Models' for a detailed description of the model compilation process.

See Also

START, GO

START

Purpose

Runs the model from its initial conditions.

Format

runtime = **START;**

Outputs

runtime	string, the duration of run

Remarks

The **START** command is used to start the first run of the model and any subsequent runs which are to execute the INITIAL section of the model code.

The **START** command clears the HALT flag and the event queue. It then transfers the user's event list to the simulation event queue and executes the INITIAL section of the model. At the end of the INITIAL section, the initial conditions are transferred to the states. If any of the dimensions of the model's states have changed since the last run the state transfer procedure is automatically recompiled and **START** is called again.

The model's DYNAMIC, DISCRETE and DATALOG sections are then executed in the order they appear in the model code to initialize all the variables for the first data logging and to **SCHEDULE** the repetitive DISCRETE blocks. Any **LOGDATA** calls are suppressed during this initial evaluation.

At this point the first output of print and plot variables can take place. If any of the dimensions or names of the print or plot variables has changed since the last run then the affected procedure is recompiled and **START** is called again. Then the state equations in the DYNAMIC section are integrated until the HALT flag becomes true or there is a user interrupt. Then the TERMINAL section of the model is executed before returning a string indicating the duration of the run.

If **_sgkey** is non−zero, a user interrupt can be initiated by pressing the **H** key.

See Also

GO, stoptime, HALT, _sgkey

STATES

Purpose

Displays the model's states.

Format

STATES;

Remarks

The order of the list of the state variables displayed by **STATES** is the order of the columns of the A and C matrices returned by **SGLIN**. The derivatives of these states, in the order displayed, are the rows of the A and B matrices. The time state, **t**, is technically always a state of the model but is not included in the matrices returned by **SGLIN**.

Example

```
states;
In addition to the time state, T, the model's states
are :-
X[1] X[2] X[3]
V[1] V[2] V[3]
```

This example displays the states of a model containing two state vectors each of dimension 3. The order of the states displayed indicates that the first three columns of the A and C matrices returned by SGLIN are associated with state elements **X[1], X[2]** and **X[3]**. The last three columns are associated with the state elements **V[1], V[2]** and **V[3]**. The six rows of the A and B matrices are associated with the derivatives of these states, i.e. **D_X[1], D_X[2], D_X[3], D_V[1], D_V[2]** and **D_V[3]**.

See Also

SGLIN, CONTROLS, OBSERV

stoptime

Purpose

Sets the end time for the integration.

Format

stoptime	scalar, the stop time for the run.

Remarks

When either **START** or **GO** is used to run the model, **stoptime** sets the time at which the run will halt. The default value is 0. **START** and **GO** will terminate immediately if the current time, t, is greater than or equal to stoptime.

If **stoptime** is equal to an event time, then **stoptime** takes precedence and simulation halts without having executed that event.

The integration can also be stopped prior to the **stoptime** by using the **HALT** procedure within the model or by pressing the **H** key if **_sgkey** is non−zero.

Globals

stoptime

See Also

i_t, HALT, _sgkey

TABLE

Purpose

Looks up the table of a function of 3 variables.

Format

fnzxy = **TABLE(** *tablemat* , *zvec* , *xvec* , *yvec* , *z* , *x* , *y* **);**

Inputs

tablemat	X*Z x Y matrix, contains the function's data points. This is defined by the user prior to calling **TABLE**.
zvec	Zx1 vector, the break points for z axis.−
xvec	Xx1 vector, the break points for x axis.
yvec	Yx1 vector, the break points for y axis. These vectors must be strictly monotonically increasing. That is the next break point must always be greater than the last one.
z	Nx1 vector or scalar, the index points for z co−ordinate.
x	Nx1 vector or scalar, the index points for x co−ordinate.
y	Nx1 vector or scalar, the index points for y co−ordinate.

Outputs

fnzxy	Nx1 vector, the value of the function at (*z*,*x*,*y*).

Remarks

The table of functional values is defined by the user before calling **TABLE**. Size of the table matrix, tablemat, is length (xvec)*length(zvec) rows by lenght(yvec) columns. This can be thought of as a number of matrices (length(zvec) matrices) each with length(xvec) rows and length(yvec) columns. That is, there is one matrix for each element in zvec and the matrices are stored by appending them row−wise.

For example

```
tablemat[1,1] = fn(zvec[1],xvec[1],yvec[1])
tablemat[1,2] = fn(zvec[1],xvec[1],yvec[2])
                    ....
tablemat[1,n] = fn(zvec[1],xvec[1],yvec[n])
                    ....
tablemat[2,n] = fn(zvec[1],xvec[2],yvec[n])
```

```
                    ....
tablemat[m,n] = fn(zvec[1],xvec[m],yvec[n])
                    ....
tablemat[1*rows(xvec)+m,n] = fn(zvec[2],xvec[m],yvec[n])
                    ....
tablemat[(k−1*rows(xvec)+m,n] = fn(zvec[k],xvec[m],yvec[n])
```

Linear interpolation is used to calculate the functional value. If any of the input arguments lie outside the range of the table, linear interpolation is used on the last two points in the table to calculate the result.

Example

A three dimensional function has the following functional values:

z	**x**	**y**	**f(zxy)**
2	−1	0	1
2	−1	1	2
2	0	0	3
2	0	1	4
3	−1	0	2
3	−1	1	4
3	0	0	6
3	0	1	8

For this function the vectors and table matrix are

```
» zv = {2,3};  xv = {−1,0};  yv = {0,1}
» a = {1 2,  3 4,  2 4,  6 8};
» a

1.000000 2.000000
3.000000 4.000000
2.000000 4.000000
6.000000 8.000000

» table(a,  zx,  xv,  yv,  2,  0|1,  0|0.5);
3.000000
5.500000
```

Here the second *x* index, 1, is outside the range of the table so the interpolated functional values for *fn*(2,−1,0.5)=0.5 and *fn*(2,0,0.5)=3.5 are used to extend the table to (2,1,0.5).

```
» a[1:2,.]
      1.000000 2.000000
      3.000000 4.000000
» table(a[1:2,.],0,xv,yv,0,0,1);
      4.000000
```

This is an example of a two dimensional table formed by taking the matrix for z=2. In the TABLE statement *zvec* and *z* are replaced with the same arbitrary constant value. If the index vector, *zvec*, is a scalar then the index value, *z*, must be the same scalar.

```
» a[1,.]
      1.000000 2.000000
» table(a[1,.],0,0,yv,0,0,2);
      3.000000
```

Here a one dimensional table is formed by taking the first row of a and replacing *zvec* and *z*, and *xvec* and *x* with arbitrary constant values.

TERMINAL

Purpose

Indicates the start of the model's TERMINAL section.

Format

TERMINAL;

Remarks

The SimGauss translator puts the code in the model's TERMINAL section into a procedure called **SGP_TERM**. This procedure is then **CALL**ed at the end of each run immediately prior to returning the run time.

Technical Notes

Since the model code in the TERMINAL section is put into a separate procedure, you cannot branch out of this section using **GOTO** statements and procedure definitions are not allowed in this section. Also **IF** and **DO** statements must not extend outside this section.

See Also

END_TERM, INITIAL, DYNAMIC, DISCRETE, DATALOG